SEICHE 2023
Intermediate Arduino Programming for IoT

Instructor: Paul Frommeyer

www.paulfrommeyer.com

Corporate Sponsor: DXC Technology

DXC TECHNOLOGY

YOUR INSTRUCTOR

Career

- Computer Infrastructure Engineer (30+ yrs)
  - Apple Computer
  - Sony
  - General Magic
  - Cisco Systems
  - HP/HPE/DXC [current]
- Linux/Unix Sysadmin (20+ years)
- C developer (professionally just entry level)
- BS Computer Science, BS Philosophy

Personal

- Ubergeek-at-large
- Computers, electronics, woodworking
  - Electronics since 6th grade
  - Computers since 7th grade

Class Format and Goals

“Understanding of Arduino programming using ESP8266 for IoT”

Goals

- Understanding of C preprocessor operations
- Understanding of variable internals and use of pointers
- Basic understanding of FAT and related filesystems
- Ability to use SPIFFS for file storage on ESP8266
- Basic understanding of HTML language and HTTP protocol
- Ability to create access points and web servers with ESP8266
- Basic understanding of network messaging with Arduino
- Basic usage of git and github for document version control

To Be Avoided

- Detailed electronics knowledge and physical wiring of components
- Processor internals and microcontroller internal architecture
- Extensive math, computer science, or EE
- Higher order web technologies like Java, JSON, and CSS

Lesson Plan Overview

Lesson 1 and 2: 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 3: 14Feb23 – More Gory Details

- Class Exercise: Call-by-reference variables
- Introduction to Complex Data Structures
- Using complex data structures
- Class Exercise: Declare and use complex data structures
- Introduction to linked lists
- Linked list usage example
- Different storage types
- Storage declarations and Arduino

Lesson 4: 21Feb23 – Fonts Redux

- 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 5: 28Feb23 – Intro to Visual Studio Code

- Introduction to VSC
- https://code.visualstudio.com/docs/introvideos/overview
- Installation

Lesson 6: 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 7: 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 8: 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 9: 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 10: 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 11:11Apr23 – 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 12:18Apr23 – Using VSC

- VSC plugin review
- Using VSC with Arduino
- Using VSC with Git

USB Flash Drive Inventory

DOCUMENTATION – Arduino Reference

- Programming with Arduino.pdf
- Arduino Cookbook-2ndEdition.pdf
- Arduino-For-Beginners-REV2.pdf
- IntroArduinoBook-AlanSmith.pdf
- IntroductionToArduino-Book-AlanGSmith.pdf
- Make_Getting_Started_with_Arduino_3E.pdf

DOCUMENTATION – Electronics Reference

- the-original-guide-to-boards-2021.pdf
- Basic Electronics-Semiconductors.pdf
- Grobs Basic Electronics 2010.pdf
- Instructables-Basic-Electronics.pdf
- Intro to Electronics-Noisemantra.pdf
- Make Electronics 2nd Edition by Charles Platt.pdf
- SPIE-TT107-PracticalElectronicsforOpticalDesignandEngineering-Chapter1.pdf

DOCUMENTATION – Internet of Things (IoT)

- Internet of Things With Arduino
- Building Arduino Projects For The Internet Of Things

DOCUMENTATION – ESP8266

- ESP8266 pinout diagram

DOCUMENTATION – Boards Guides

- Original Boards Guides from 2019-2022

DOCUMENTATION – Class Information

- SEICHE-BasicArduinoProgramming-Lesson1-V1.pdf

SOFTWARE – Visual Studio Code

- Windows x64
  - “User” non-elevated installer
  - “Normal” elevated installer
- Kubuntu x64
  - .deb package for use with Aptitude (apt-install)

Critical Documents

- Internet of Things With Arduino – Use this for reference and class excercises
- Building Arduino Projects For The Internet Of Things – Use this for advanced self-study

RETURN FLASH DRIVES AT END OF EACH LESSON

- I’ll load new content each week, at least that week’s lesson but also sketches when needed

Lesson 1 – Review, Addressing and Pointers

Part A

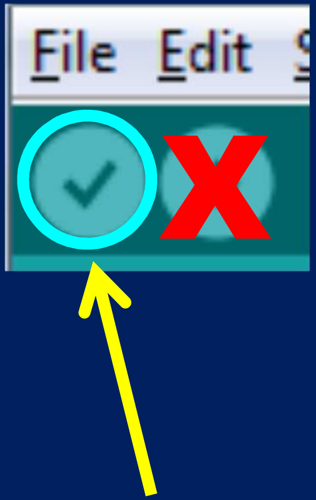
- Overview of lesson plan
- Inventory of USB Drives – New files
- Review of ESP8266 connection and software upload (Class Exercise)

Part B

- More About the Preprocessor
- Memory organization and variables
- Addresses and pointers
- Using pointers to variables
- Pointers and Arrays
- Dereferencing
- Call-by-value and Call-by-reference in function calls

Preparing a Sketch

- Launch the Arduino IDE by double clicking on the sketch SKETCH1A which has been copied to your hard drive (do not use the sketch on your USB drive!)
- Click on the "VALIDATE" check mark icon
- DO NOT click on the "right arrow" (UPLOAD) icon at this time
- This will *compile* the sketch and check for errors prior to upload
- This is normally an optional step, since selecting upload will always compile the code first.
- Whenever using a new board (or library) for the first time, it's a good idea to validate just to assure the software is happy before attempting and upload
- We are (re)validating that all your libraries, and particularly your board definitions, are still in place and working.



This check mark icon is for VALIDATE

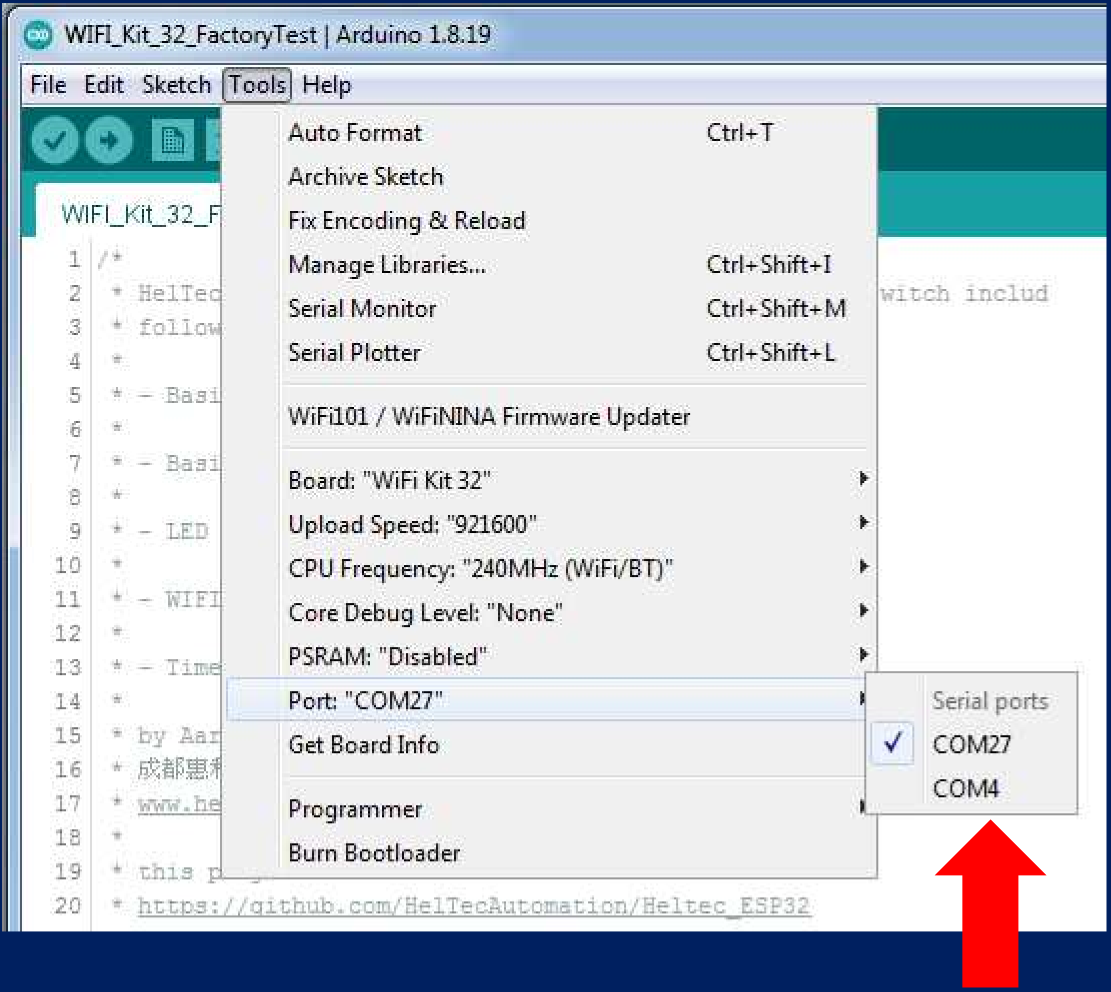
Finding and Setting the USB Port

WINDOWS

- Use Device Manager to identify the USB port
- The port will change with each connect/disconnect of the board

KUBUNTU

- Port should always be /dev/ttyUSB0
- The port will not change with connect/disconnect
- You can use Konsole to verify the port presence
  `ls /dev/tty*`

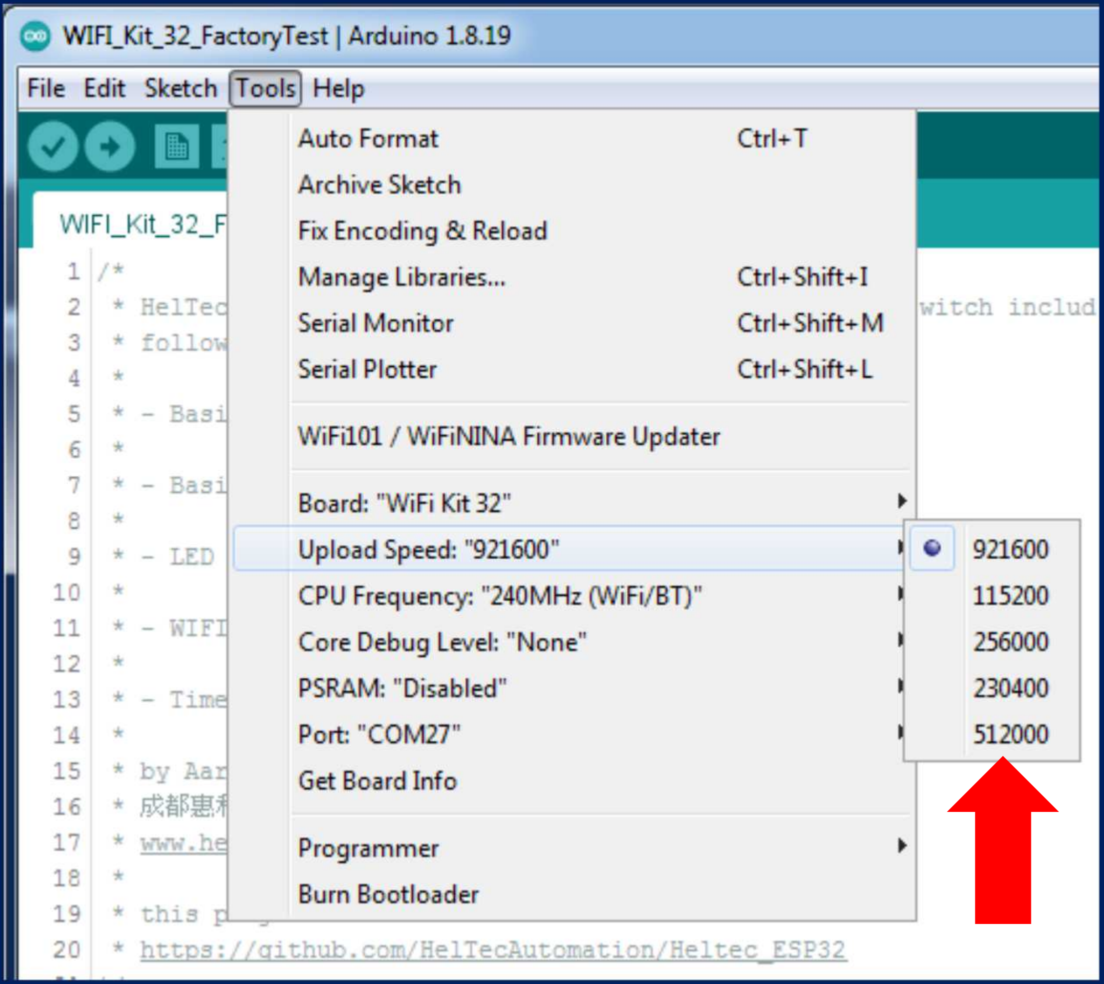

WIFI_Kit_32_FactoryTest | Arduino 1.8.19
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "WiFi Kit 32"
Upload Speed: "921600"
CPU Frequency: "240MHz (WiFi/BT)"
Core Debug Level: "None"
PSRAM: "Disabled"
Port: "COM27"
Get Board Info
Programmer
Burn Bootloader
Serial ports
✓ COM27
COM4

Adjusting Port Speed

Always start by using 921600, this works with nearly all USB chipsets (there are exceptions for the ESP32, which uses the Sil2102 driver– but that's not us!)

If you get an upload failure

- Recheck port setting
- Retry the upload with the next lowest speed
- Lather, rinse, repeat until success
- Suggested initial try order: 921600, 512000, 115200

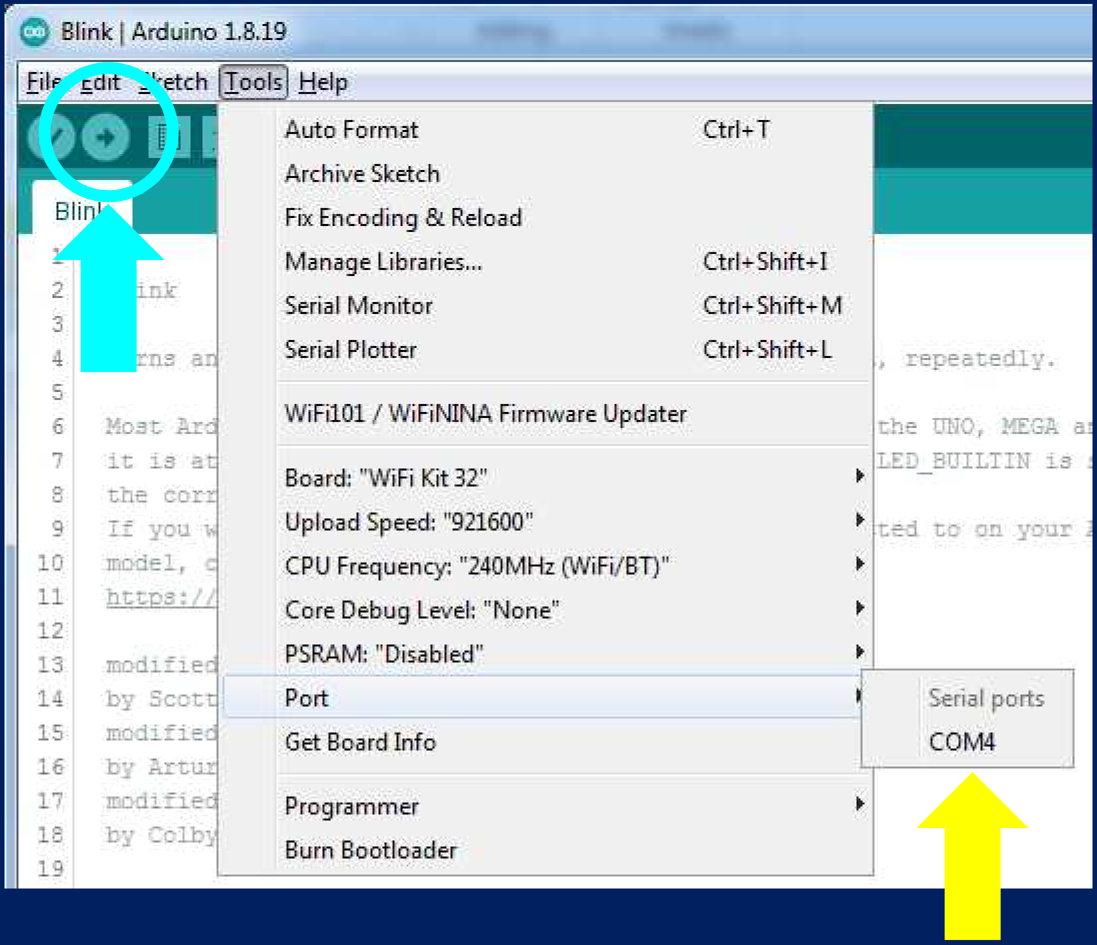

WIFI_Kit_32_FactoryTest | Arduino 1.8.19
File Edit Sketch Tools Help
WIFI_Kit_32_F
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Get Board Info
Programmer
Burn Bootloader
921600
115200
256000
230400
512000

Board: "WiFi Kit 32"
Upload Speed: "921600"
CPU Frequency: "240MHz (WiFi/BT)"
Core Debug Level: "None"
PSRAM: "Disabled"
Port: "COM27"

```
/*
 * HelTec
 * follow
 *
 * - Basi
 *
 * - Basi
 *
 * - LED
 *
 * - WIFI
 *
 * - Time
 *
 * by Aar
 * 成都惠
 * www.he
 *
 * this p
 * https://github.com/HelTecAutomation/Heltec_ESP32
```

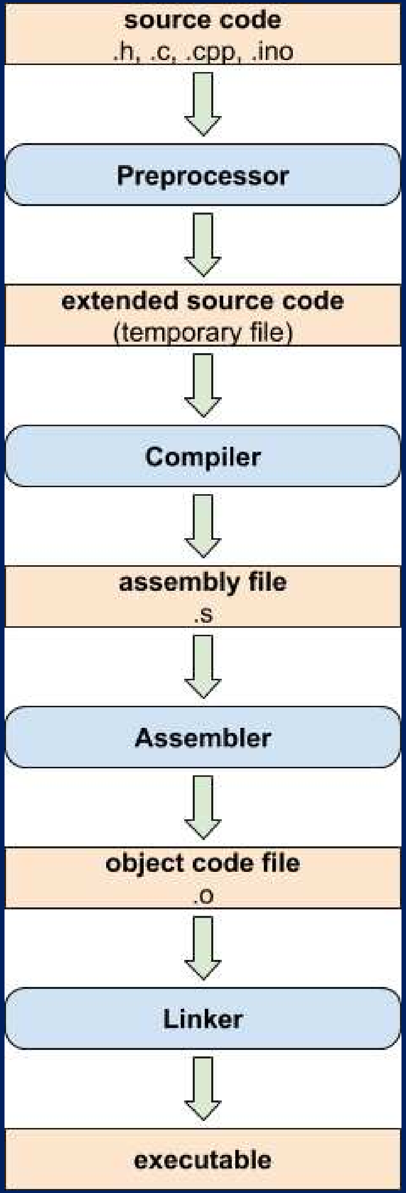
Uploading a Sketch

- **Ensure you have selected the correct port for your currently plugged in board**
- NOW click on the “UPLOAD” arrow icon
- Once upload is finished, the sketch will immediately run
- You should a message scroll on your displays
- As before, feel free to change the text message if you want



Arduino IDE Compile Sequence

Before your sketch can be uploaded, it must be *compiled* into a machine language file which the microcontroller can understand and execute. There are several steps involved in the compilation process; these can vary by langauge and/or IDE, but at left is the compiler flow for the Arduino IDE



More About The Preprocessor

Before the C/C++ code of a sketch is compiled into machine code, a preprocessor is run over the code. The preprocessor reads certain directives which are used to transform the sketch code.

- `/*` and `*/` – Any text between these comment delimeters is removed along with the delimeters
- `//` – Any text following this comment delimeter before another preprocessor directive or code statement is removed
- `#include` – specifies a file with additional code that should be directly included/incorporated into the sketch at the location where the #include occurs
- `#define` – Specifies a keyword variable which will be substituted for it’s defined text wherever it occurs throughout the code
- `#ifdef` – Executes conditional code if the specified keyword variable *has* been defined
- `#ifndef` – Executes conditional code if the specified keyboard variable *has not* been defined

The #include directive

`#include` is used to incorporate code from files “external” to the sketch it is used in

`#include “somefile.ino”`
Path specifier usage - includes a file from the same directory, or absolute directory path, as the sketch containing the directive

`#include <foo.h>`
Library usage – checks standard library locations for the file to be included. Most often used for header files to invoke a library to be used with the containing sketch. Some versions of C/C++ treat the .h suffix as optional when <> library specifier is used

The #define directive

`#define` is used to declare "substitute" text which will be globally replaced when used

```
#define FOO SEICHE2023
```

Will replace all occurences of "FOO" in the sketch with the text "SEICHE2023" wherever it occurs

```
#define TRUE 1
```

Will cause all usages of "TRUE" to be replaced with the numeral 1. This is actually a system default definition already made by the IDE

```
#define NULL 0
```

Will cause all usages of "NULL" to be replaced with the numeral 0. Here again, this is actually a system default definition already made by the IDE – it's already been defined for you.

Note: redefining a preprocessor variable will throw a warning message in the IDE, however the variable WILL be redefined.

The #ifdef and #ifndef directives

`#ifdef` is `#ifndef` are used to execute conditional code if a preprocessor variable has or has not been defined, respectively

```
#ifdef LASTYEAR
   char message[] = “SEICHE2022”
#endif
```

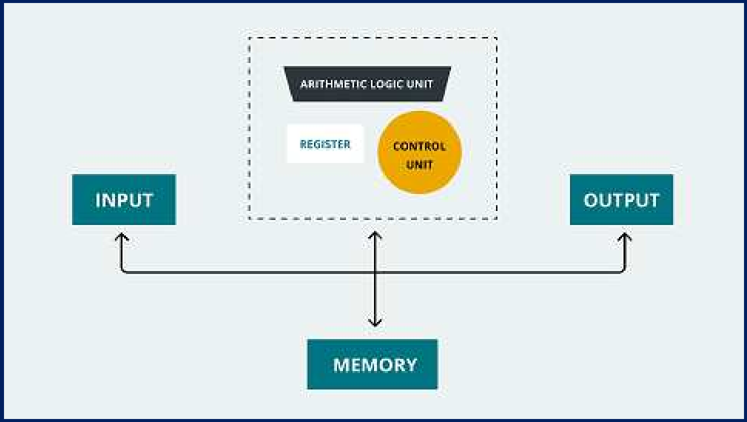
Will declare the string “message” with ASCII text “SEICHE2022” if preprocessor variable FOO has been defined. It does not matter *what* FOO has been defined *as*.

```
#ifndef LASTYEAR
    char message[] = “SEICHE2023”
#endif
```

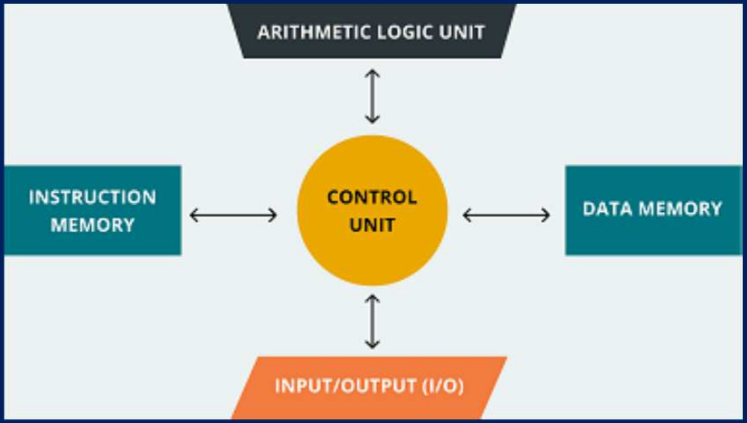
If LASTYEAR has not been defined, then declare “message” with “SEICHE2023”

Memory Organization and Variables
Memory Architectures

Von Neumann – A single monolithic memory (one big pool) is used for everything– program storage, predeclared variable storage, dynamic variable storage, input, output, the works.



Harvard– Discrete pieces of memory– usually physically separate chips– are used for the different operations of the computer. This architecture is preferred in microcontrollers, particularly those used by Arduino

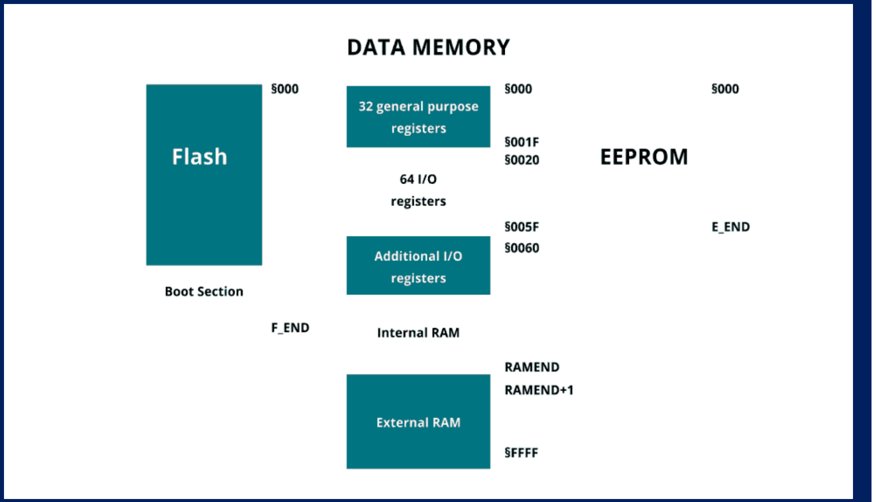


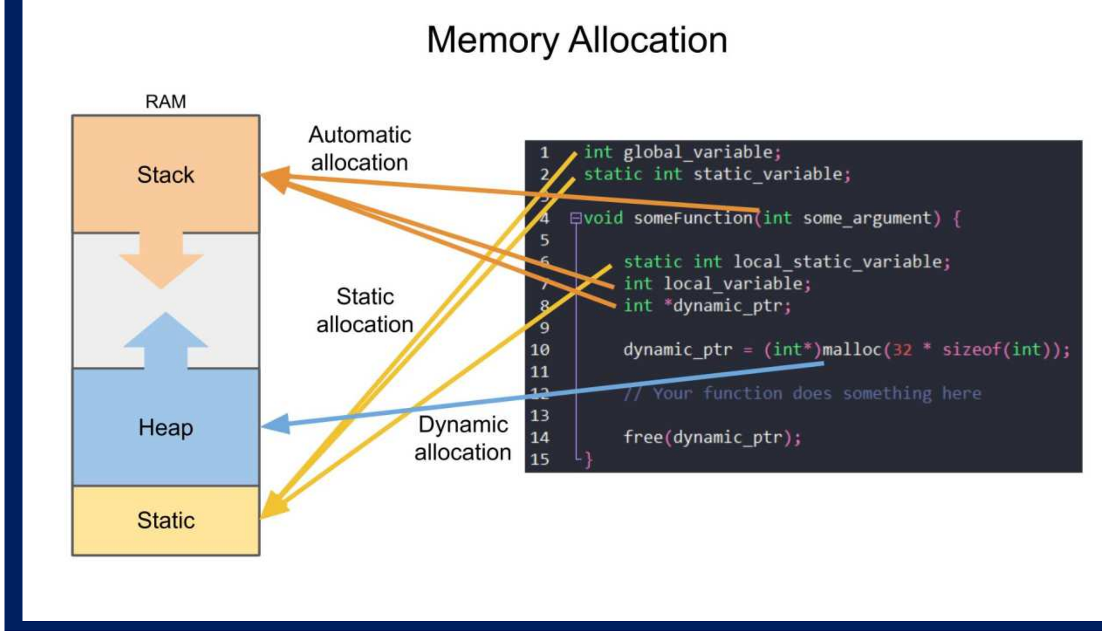
Memory Types

- **Flash** – This is where firmware is stored to be executed; this is where the compiled binary files from the IDE are stored.
- **RAM** – This is where temporary, or “run-time”, data is stored. RAM in microcontrollers is usually SRAM.
  - SRAM - “static RAM”, which means contents persist even if power is removed for a relatively short time. SRAM is usually slower than DRAM.
  - DRAM – “dynamic RAM”, requires constant refreshing by circuitry or data is lost. Even a 1 nanosecond signal fluctuation can corrupt DRAM. DRAM is usually faster than SRAM, which is why DRAM is the hardware of choice for laptops and servers.
- **EEPROM** - In microcontroller-based systems, Erasable Programmable Read-Only Memory, or EEPROM, is also part of its ROM; actually, Flash memory is a type of EEPROM. The main difference between Flash memory and EEPROM is how they are managed; EEPROM can be managed at the byte level (write or erased) while Flash can be managed at the block level.

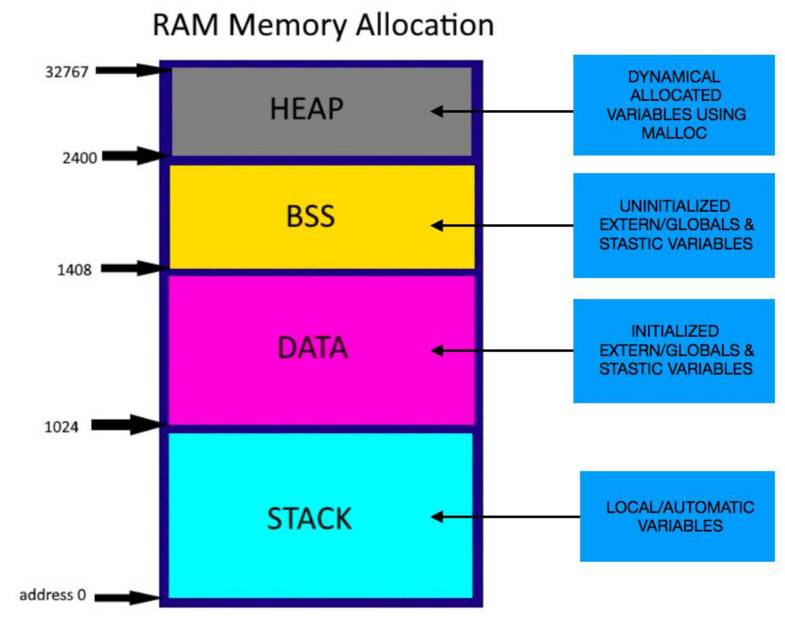
Memory Organization and Variables Arduino AVR Memory Model

How memory is organized in a computer is called a memory map. The memory map determines where program code and predeclared and dynamic variable data are stored in the computer's memory. Different types of memory can be used for the different types of data; this is especially true for microcontrollers.





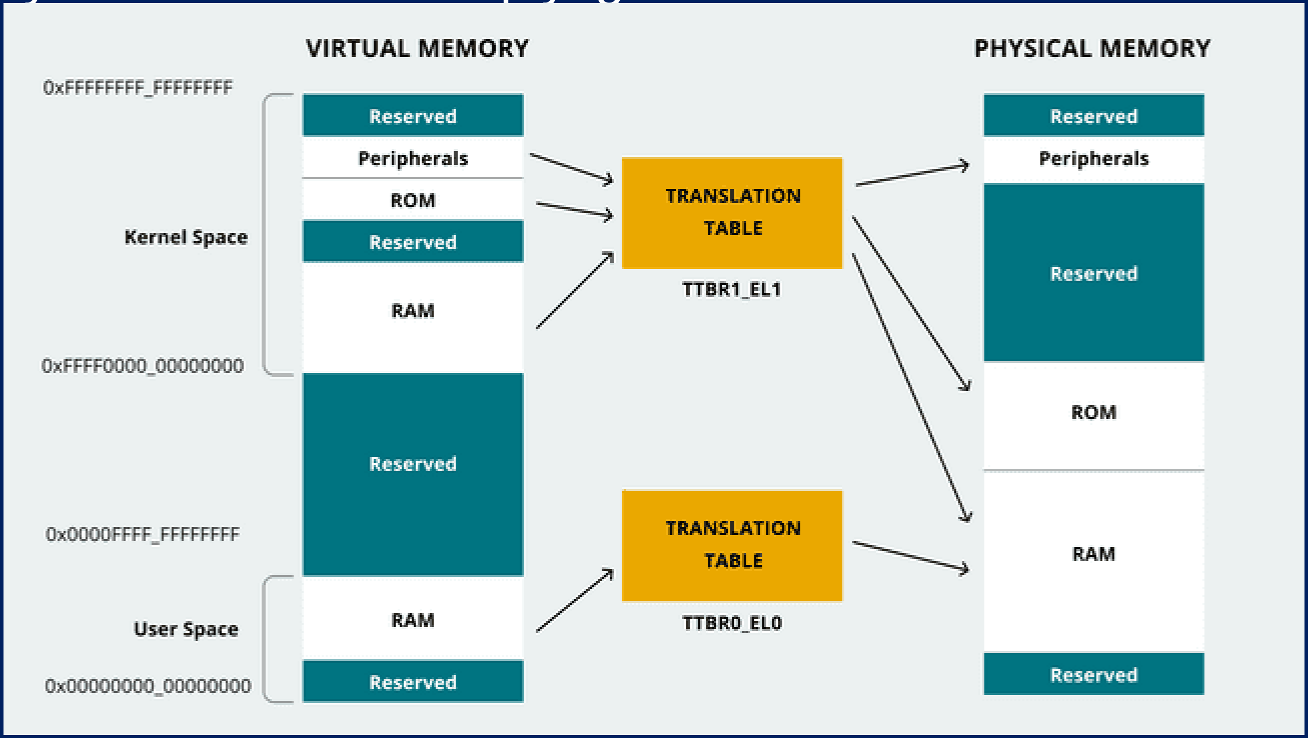
```
int global_variable;
static int static_variable;

void someFunction(int some_argument) {

    static int local_static_variable;
    int local_variable;
    int *dynamic_ptr;

    dynamic_ptr = (int*)malloc(32 * sizeof(int));

    // Your function does something here

    free(dynamic_ptr);
}
```


RAM Memory Allocation
32767
2400
1408
1024
address 0
HEAP
BSS
DATA
STACK
DYNAMICAL ALLOCATED VARIABLES USING MALLOC
UNINITIALIZED EXTERN/GLOBALS & STASTIC VARIABLES
INITIALIZED EXTERN/GLOBALS & STASTIC VARIABLES
LOCAL/AUTOMATIC VARIABLES

Memory Organization and Variables Arduino ARM Memory Model

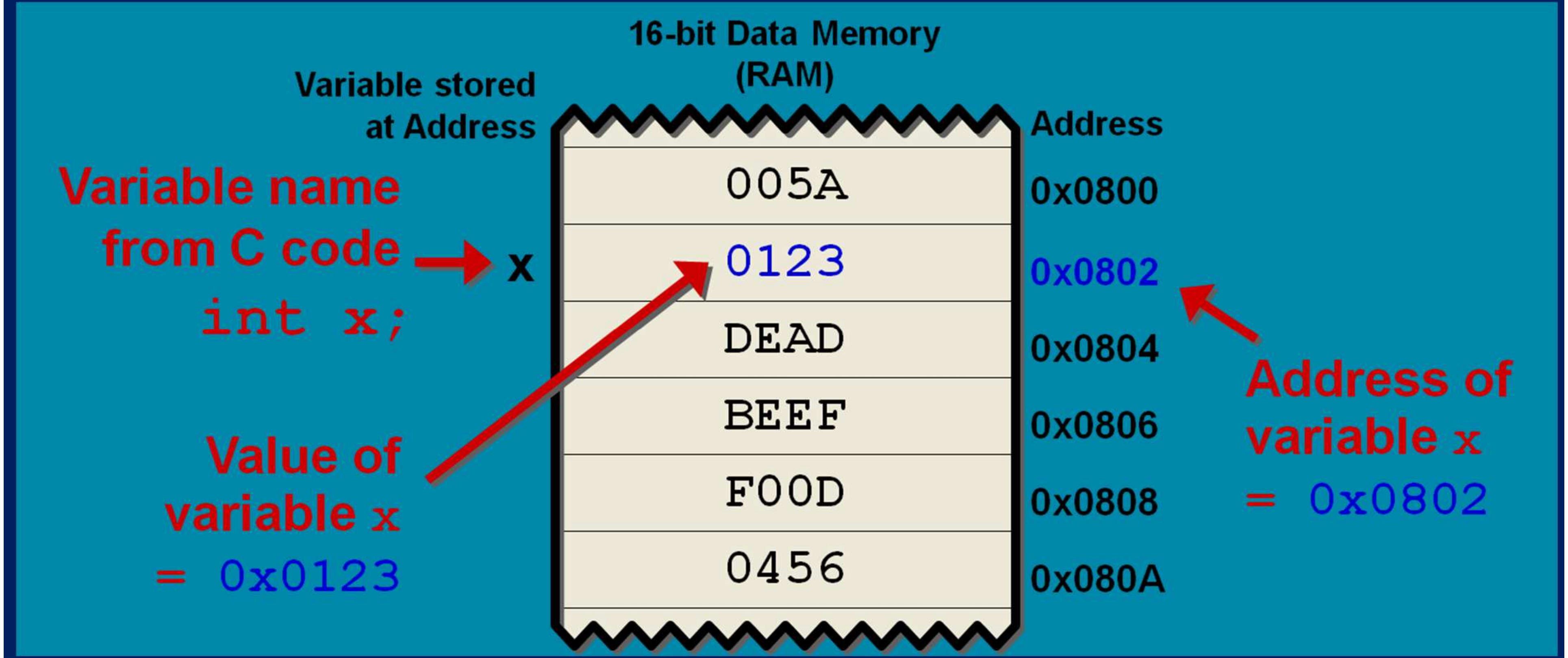
ARM and related microcontrollers utilize a scheme called virtual memory, where a small amount of physical memory is mapped (“translated”) into a vastly larger virtual memory space. Infrequently used virtual memory storage is “paged out” to slower storage media, such as flash, while unused virtual memory addresses are simply ignored.

While the original microcomputers of the 1970’s and early 1980’s all used physical memory, all modern laptop and server operating systems use virtual memory.



https://docs.arduino.cc/learn/programming/memory-guide

Memory Organization and Variables
Variable Data Storage

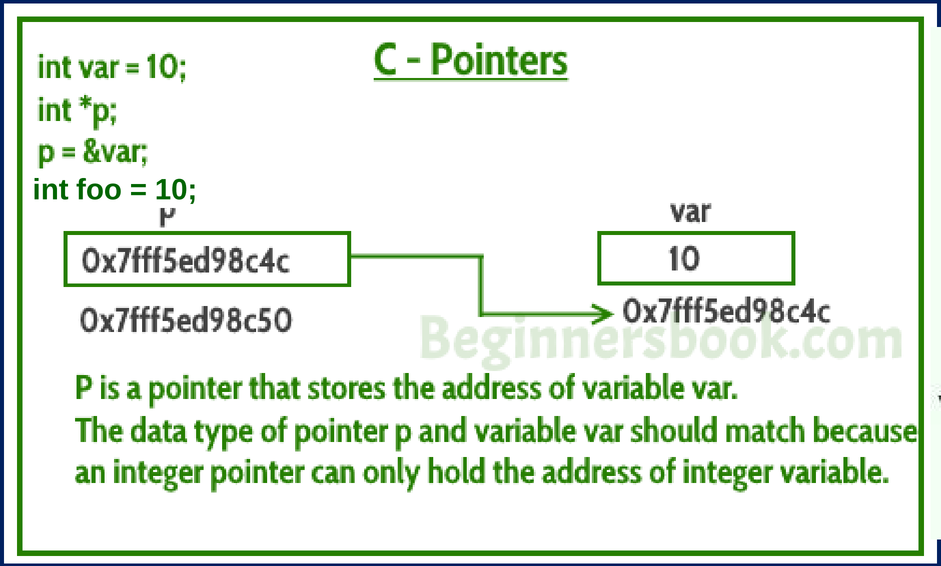

16-bit Data Memory (RAM)
Variable stored at Address
Address
Variable name from C code
x
int x;
005A
0123
DEAD
BEEF
F00D
0456
0x0800
0x0802
0x0804
0x0806
0x0808
0x080A
Address of variable x = 0x0802
Value of variable x = 0x0123

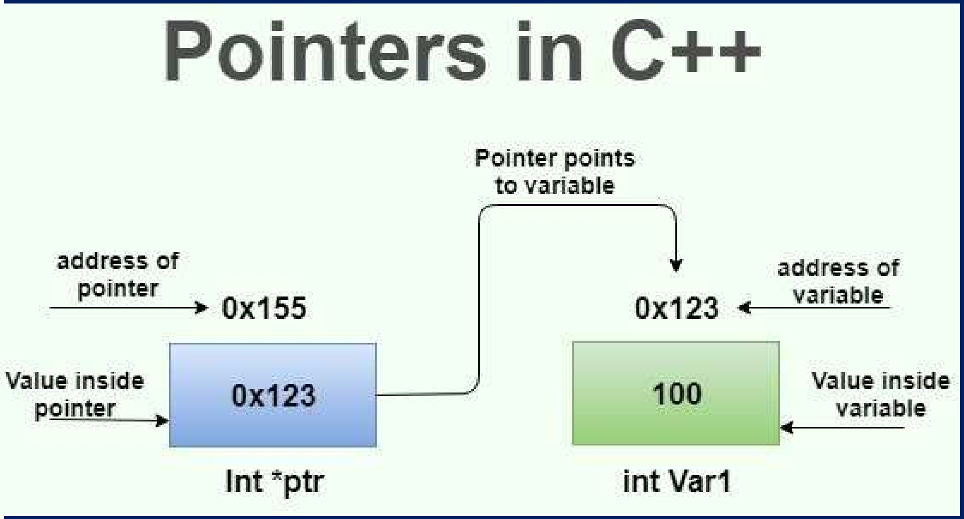
Variable data is stored at specific locations in memory; we touched on this last semester. Each variable has a “starting location” which is a specific address in memory. How many bytes are consumed at that location by the variable depends on its type, or in the case of strings, contents.

Addresses and Pointers

- **We’ve learned that a variable is data stored in memory, and that the contents of most variables can be modified (hence the name ☺)**
- The “contents” of a variable are the data stored at the *first* memory address for all the bytes *comprising* a given variable.
- As a shorthand, the phrase “address of a variable” refers to that first byte of the variable’s data structure
- **The address of a variable is itself data.** Depending on the architecture, the address may be 8, 16, 32, or even 64 bits in size
- It logically follows, then, that the *address* of a particular variable can itself be stored in another, *different* variable.
- A variable which is used to store the address of another variable is called a **pointer.**

Addresses and Pointers (cont.)

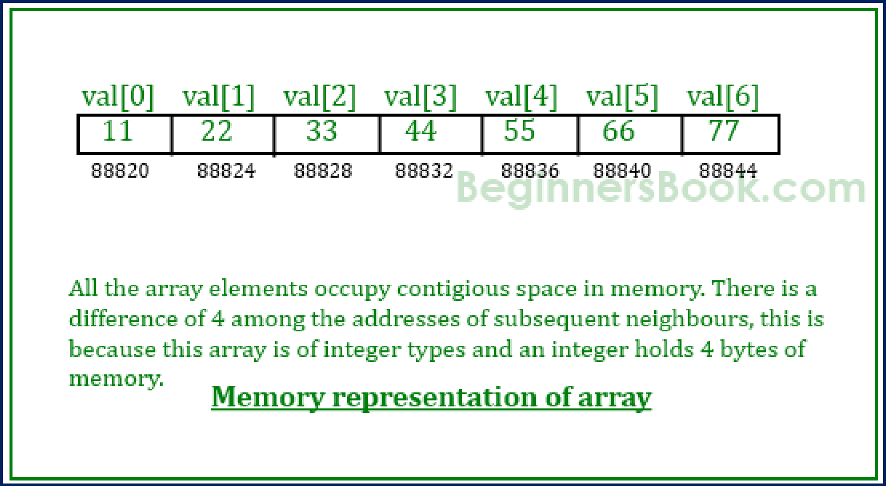

C - Pointers
int var = 10;
int *p;
p = &var;
int foo = 10;
p
0x7fff5ed98c4c
0x7fff5ed98c50
var
10
0x7fff5ed98c4c
Beginnersbook.com
P is a pointer that stores the address of variable var.
The data type of pointer p and variable var should match because an integer pointer can only hold the address of integer variable.



- The syntax for *declaring* a pointer is
  ```
  int *p;  // or
  uint64_t *p;
  ```
- And the syntax for accessing and assigning the address of a variable to a pointer is
  ```
  p = &foo;
  ```

Do not confuse *prefix* use of the asterisk (*) for a pointer with *infix* use as an arithmetic operator!

Using Pointers With Arrays

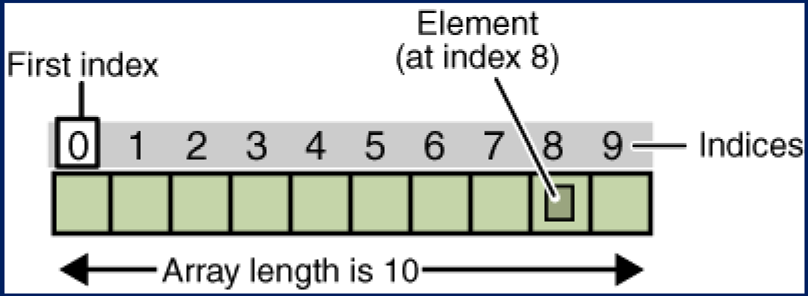


All the array elements occupy contigious space in memory. There is a difference of 4 among the addresses of subsequent neighbours, this is because this array is of integer types and an integer holds 4 bytes of memory.

Memory representation of array

- Recall that arrays are sequential data structures with variables of a given type accessed by index
  `char bar[4] = “foo”;`
- Each byte of the character array `bar` resides at a subsequent address in memory

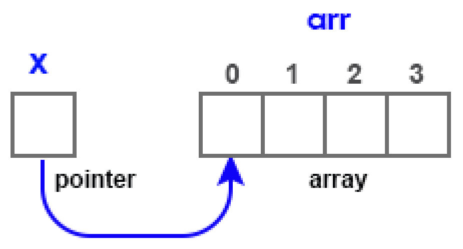
- As you may recall, the individual array members are “found” by the computer simply by adding the size of the array type to the base index of the array, which is zero.
- This process happens “behind” the scenes, but the way it is done is that the base “index” of zero is actually the address of the first member of the array.
- Then, to reach subsequent members, the address is incremented by the size of the array data type



Using Pointers With Arrays

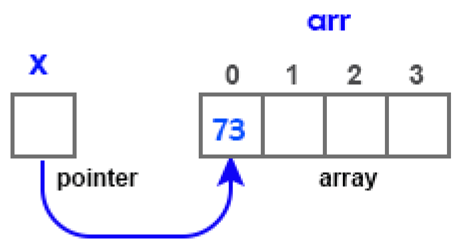
- A pointer can point at any array

```
int arr [ 4 ] , *x , *y ;
x = &arr [ 0 ];
```



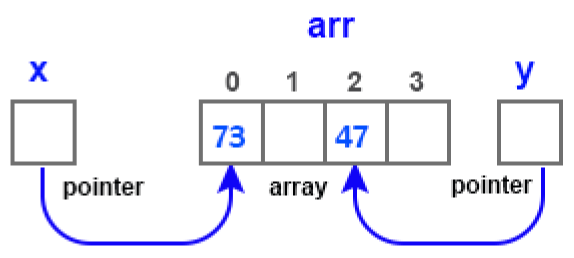
- Pointer arithmetic can be used to access array elements

```
* x = 73 ;
```



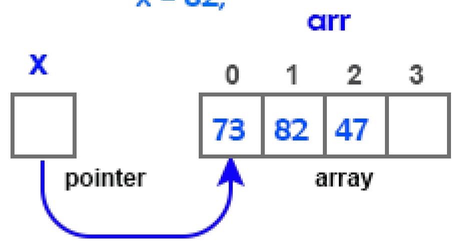
- Addition works

```
y = x + 2 ;
*y = 47 ;
```



- So does subtraction

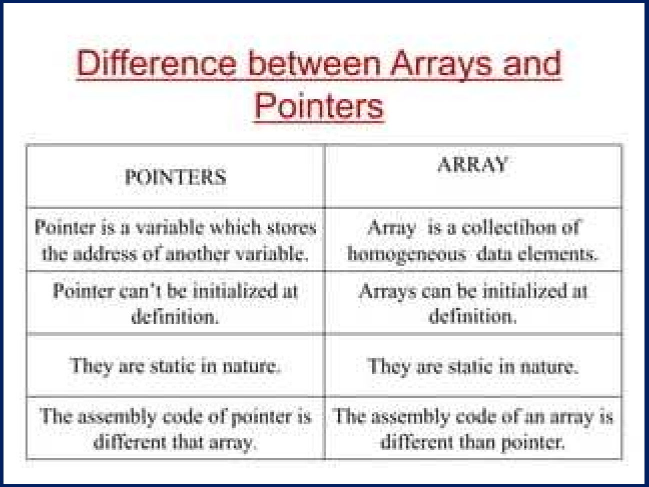
```
x = &arr [ 3 ];
x = x - 2;
*x = 82;
```



- Since pointers are variables, they can be manipulated with arithmetic operators
- In particular, addition and subtraction will cause a pointer to reference a different address in memory
- And this is how the computer does it for accessing the individual elements of an array

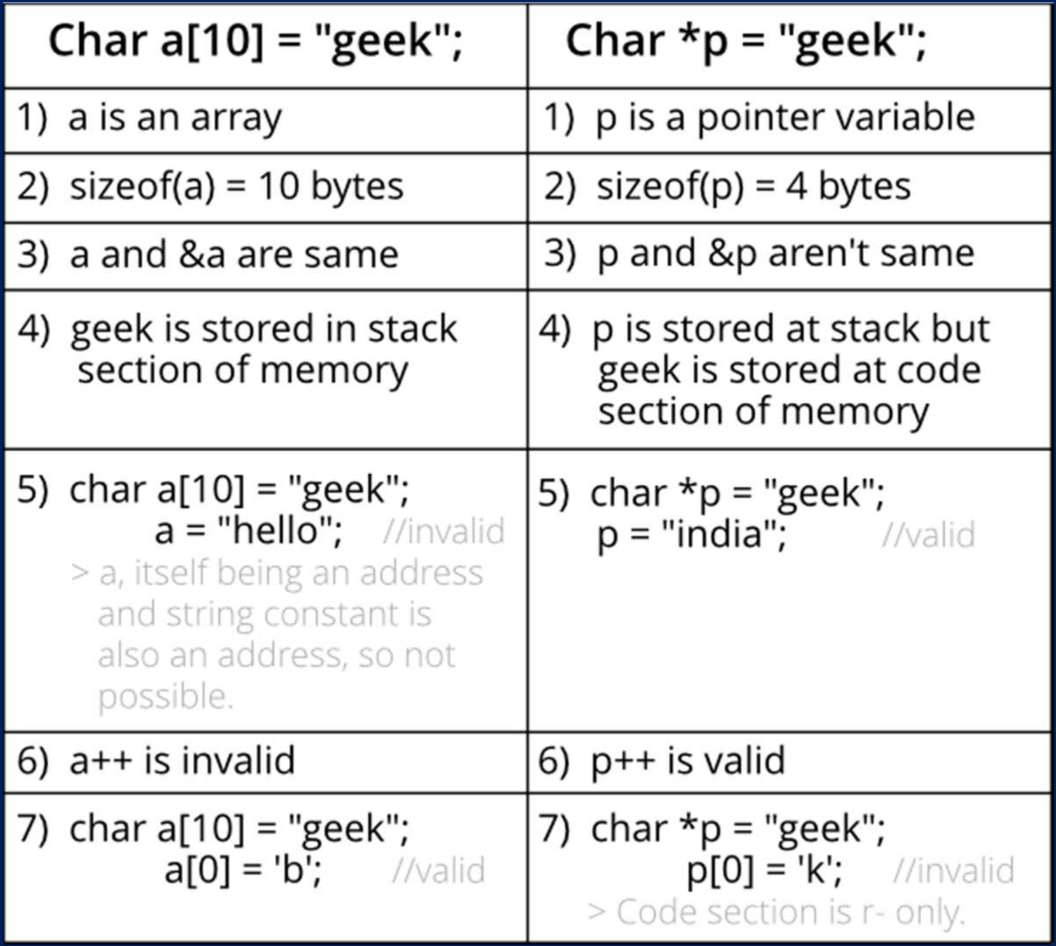
- It is possible to perform these operations “manually” using the address operator & and the pointer operator *.

Differences Between Arrays and Pointers



Difference between Arrays and Pointers

POINTERS	ARRAY
Pointer is a variable which stores the address of another variable.	Array is a collectihon of homogeneous data elements.
Pointer can't be initialized at definition.	Arrays can be initialized at definition.
They are static in nature.	They are static in nature.
The assembly code of pointer is different that array.	The assembly code of an array is different than pointer.



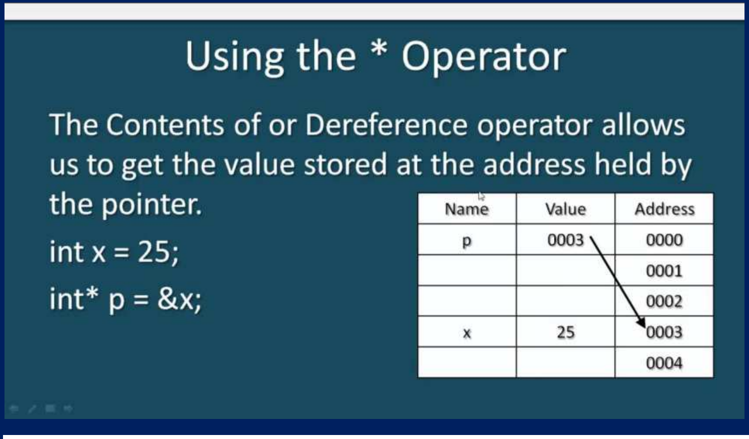
Char a[10] = "geek";	Char *p = "geek";
1) a is an array	1) p is a pointer variable
2) sizeof(a) = 10 bytes	2) sizeof(p) = 4 bytes
3) a and &a are same	3) p and &p aren't same
4) geek is stored in stack section of memory	4) p is stored at stack but geek is stored at code section of memory
5) char a[10] = "geek"; a = "hello"; //invalid > a, itself being an address and string constant is also an address, so not possible.	5) char *p = "geek"; p = "india"; //valid
6) a++ is invalid	6) p++ is valid
7) char a[10] = "geek"; a[0] = 'b'; //valid	7) char *p = "geek"; p[0] = 'k'; //invalid > Code section is r- only.

More detailed exposition of this content

https://www.youtube.com/watch?v=ToPkRyNOBZ8

Dereferencing

- So if we have a pointer, how do we use that pointer to access the contents of memory at that location?
- The process of accessing memory contents from a pointer is called dereferencing
- Dereferencing also uses the asterisk *
- How an asterisk is processed depends on where in a C statement it is used. Just be careful not to confuse pointer operations with arithmetic operations.
- A variable is dereferenced thusly:
  `foo = *bar;`

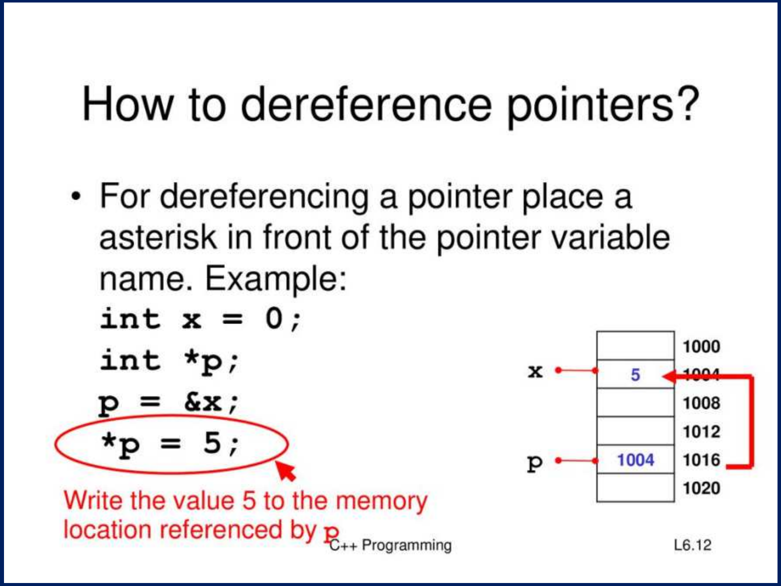


Using the * Operator

The Contents of or Dereference operator allows us to get the value stored at the address held by the pointer.

```
int x = 25;
int* p = &x;
```

Name	Value	Address
p	0003	0000
		0001
		0002
x	25	0003
		0004



How to dereference pointers?

- For dereferencing a pointer place a asterisk in front of the pointer variable name. Example:

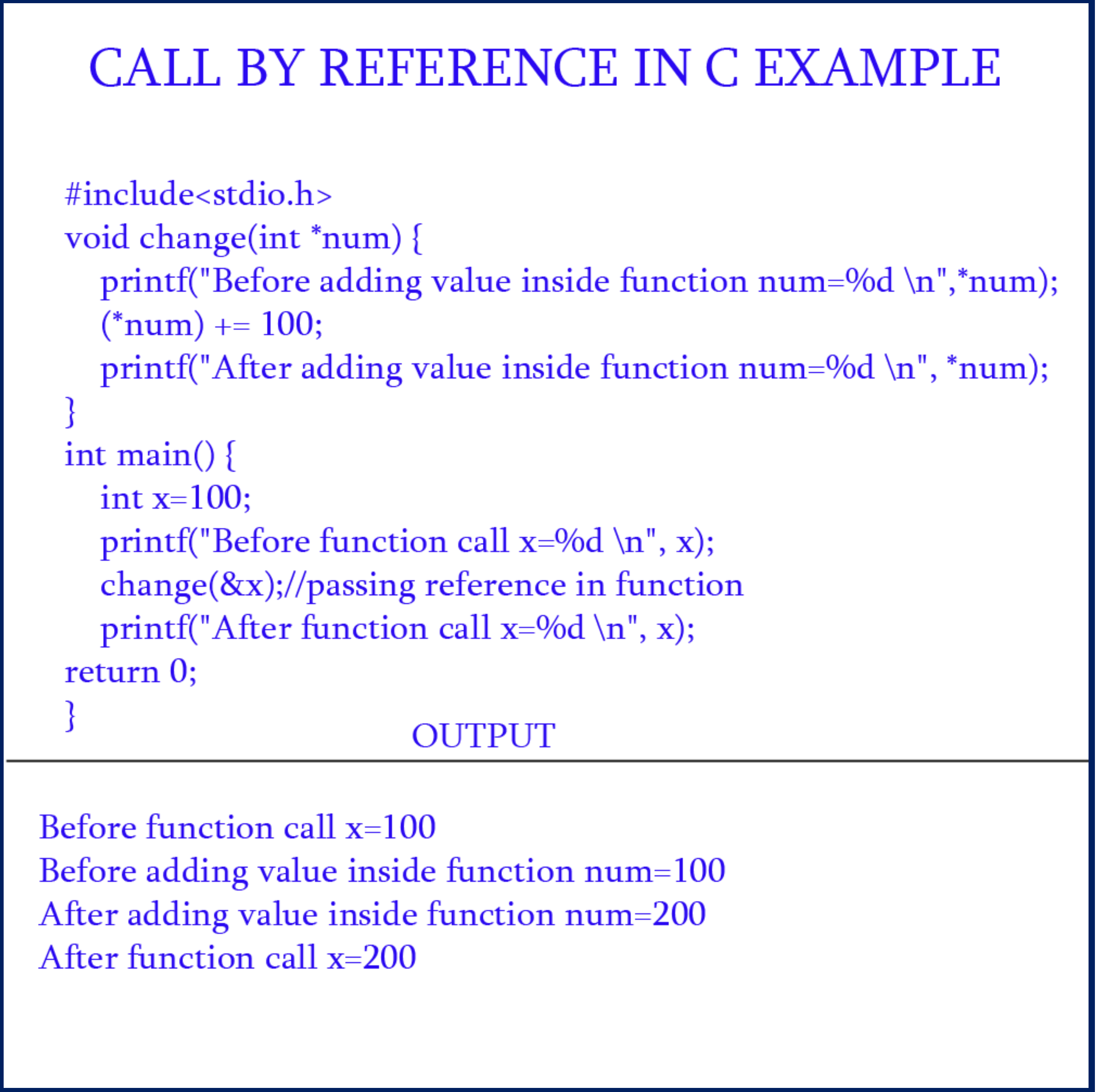
```
int x = 0;
int *p;
p = &x;
*p = 5;
```

Write the value 5 to the memory location referenced by p

C++ Programming L6.12

Pointers, Functions, and Call By Reference

- So far, when you passed a variable to a function, only the value of that variable was passed to the function. Modification of the passed value within the function would not affect the original variable.
- Passing the value of a variable in a function is referred to as **call by value**.
- However, it is possible to pass the address of a variable, rather than it's value. This allows the contents of the original variable to be modified.
- This is referred to as **call by reference**.



CALL BY REFERENCE IN C EXAMPLE

```
#include<stdio.h>
void change(int *num) {
   printf("Before adding value inside function num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside function num=%d \n", *num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(&x);//passing reference in function
   printf("After function call x=%d \n", x);
return 0;
}
```

OUTPUT

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Formal End of Lesson 1

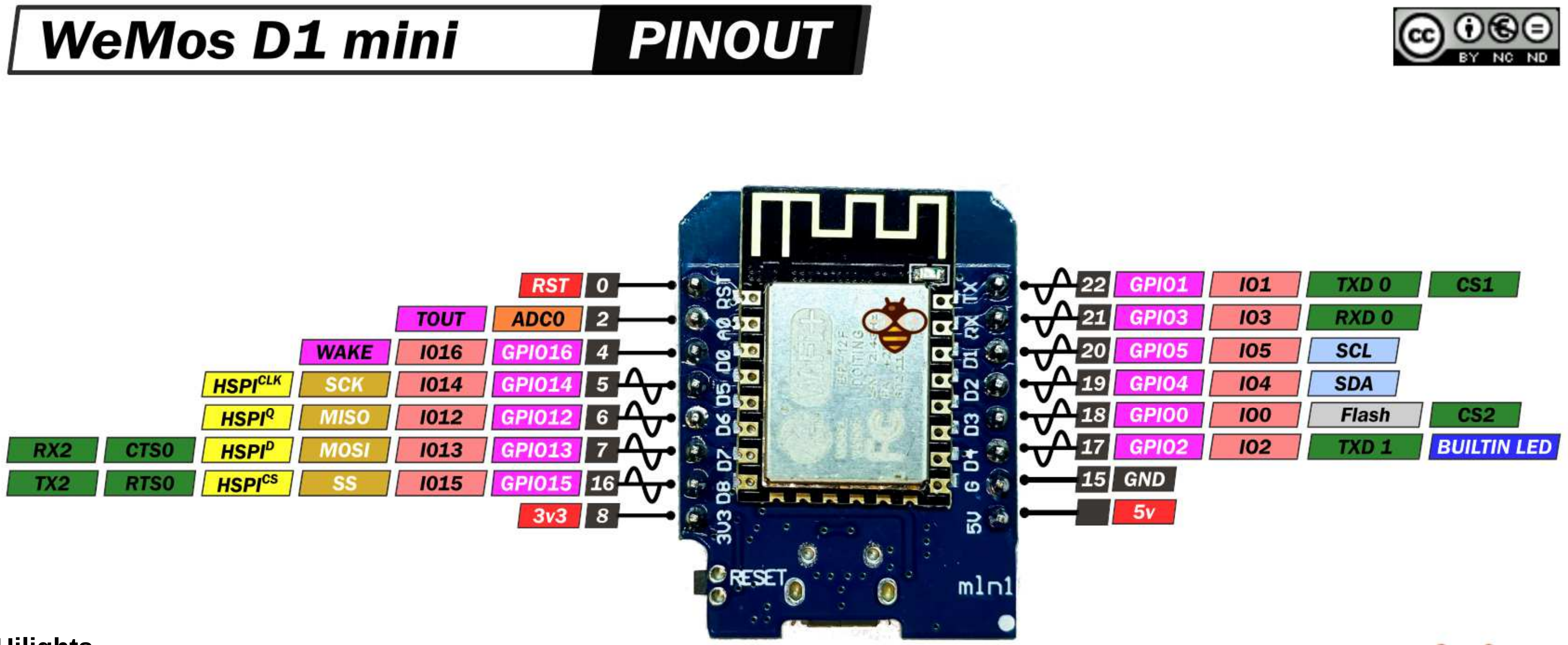
HOMEWORK!

- **No explicit homework**
- **Have a look at the new IoT books added to your flash drives**

In next week's exciting episode

- Introduction to complex data structures
- Using complex data structures
- Class Exercise: Declare and use complex data structures
- Introduction to linked lists
- Linked list usage example
- Different storage types
- Storage declarations and Arduino

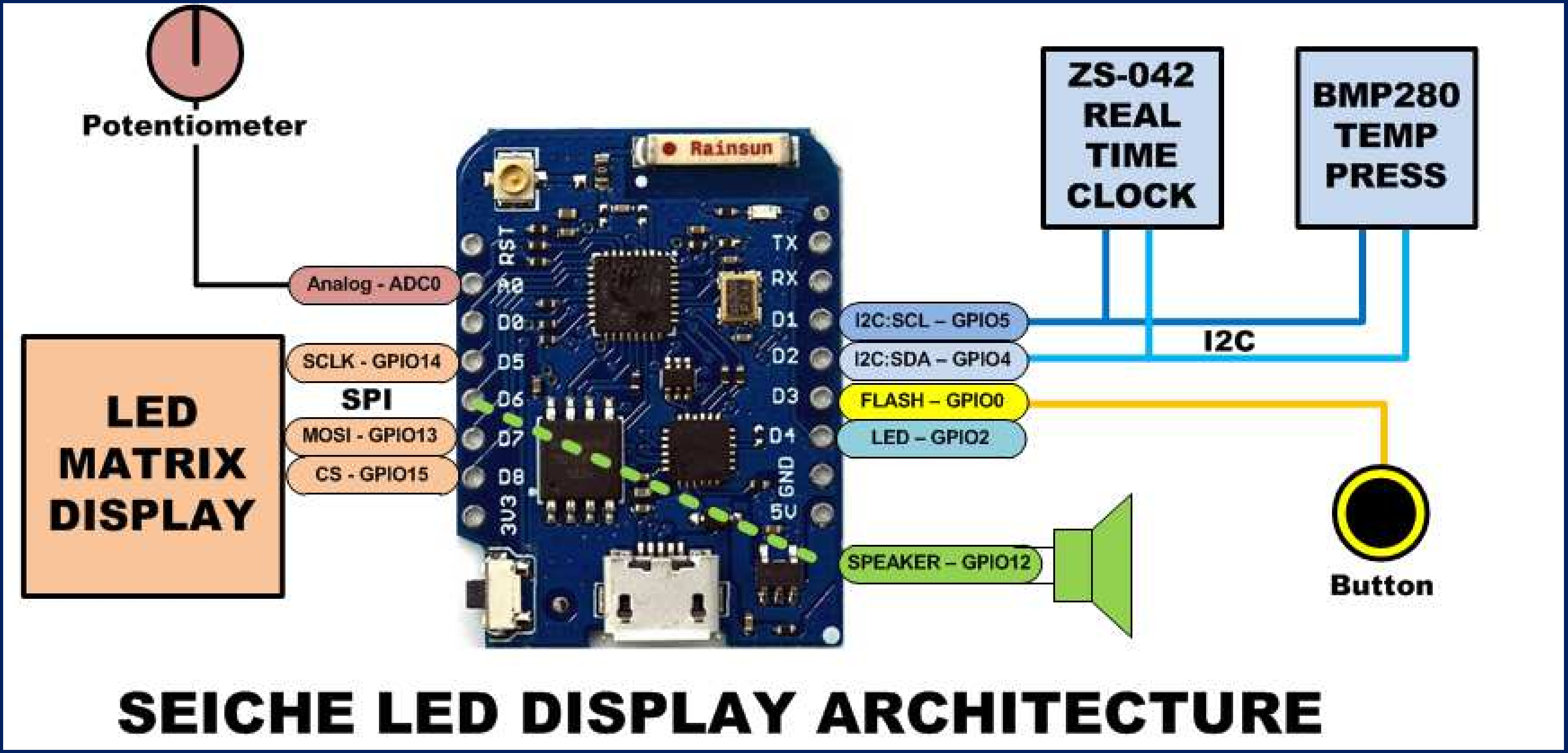
Our Microcontroller: The WeMos D1 Mini



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