Chapter 2

Embedded C/C++ Programming

Zhang Jianwen
elezhan@nus.edu.sg
Office: E4-05-21

High-level codes to machine codes

- MCU only can run machine codes.
  - Machine code is binary instructions understandable by MCU.
  - Assembly is low-level programming language, closely related to the machine codes.
  - Machine codes can be generated from assembly codes by assembler.

High-level codes to machine codes

- C/C++ is a high-level programming language
  - Friendly to programmers
    - Abstraction – such as the hardware abstraction layer (HAL).
    - Readability
    - Libraries, community and documentation
    - Error handling and memory management
  - Convert to machine code before it can run in MCU.

High-level codes to machine codes

- Compilation and linking process
  - Source code → readable codes (C/C++ syntax)
  - Preprocessing → expand source code based on preprocessing directives
  - Compilation and assembly → convert to binary codes (object files)
  - Linking → Put object files together and generate machine codes

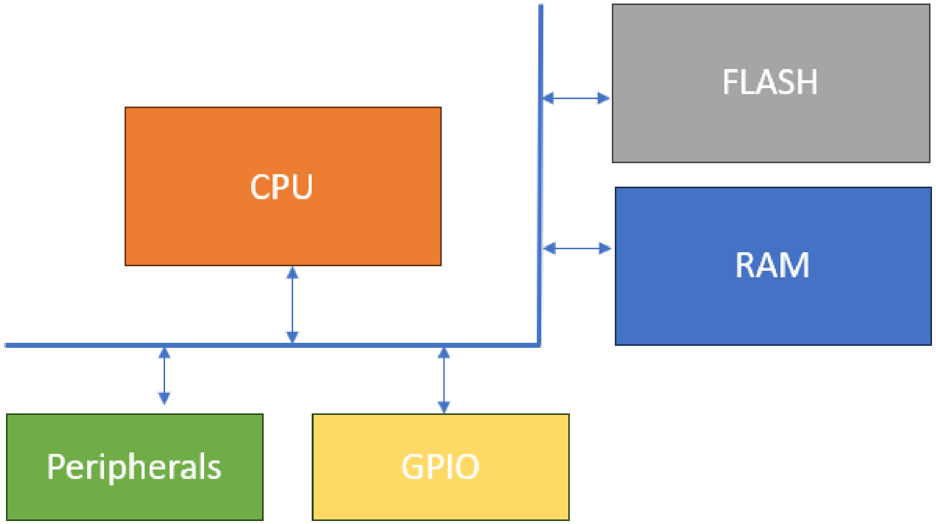
High-level codes to machine codes

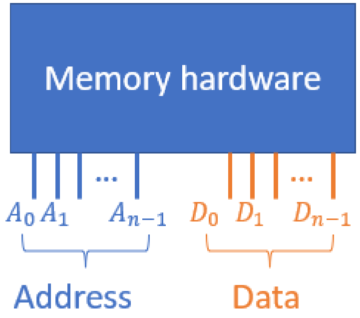
- Preprocessor directives
  - #include: include standard libs or custom head files.

    #include <WiFi.h>
  - #define and #undef: define/undef macros or constants
  - Conditional compilation directives.
  - #pragma, #error and #warning : compile-time instructions
  - Include guard

System memory

- Data can be accessed from: FLASH, CPU registers, RAM and peripherals.





System memory and address

- Memory of an embedded system
  - Code memory (IRAM)
  - Data memory (SRAM)
  - Register memory (peripherals + GPIOs)
  - RTC memory

Uniform address space

ESP32-S3
32-bit addr
32-bit data

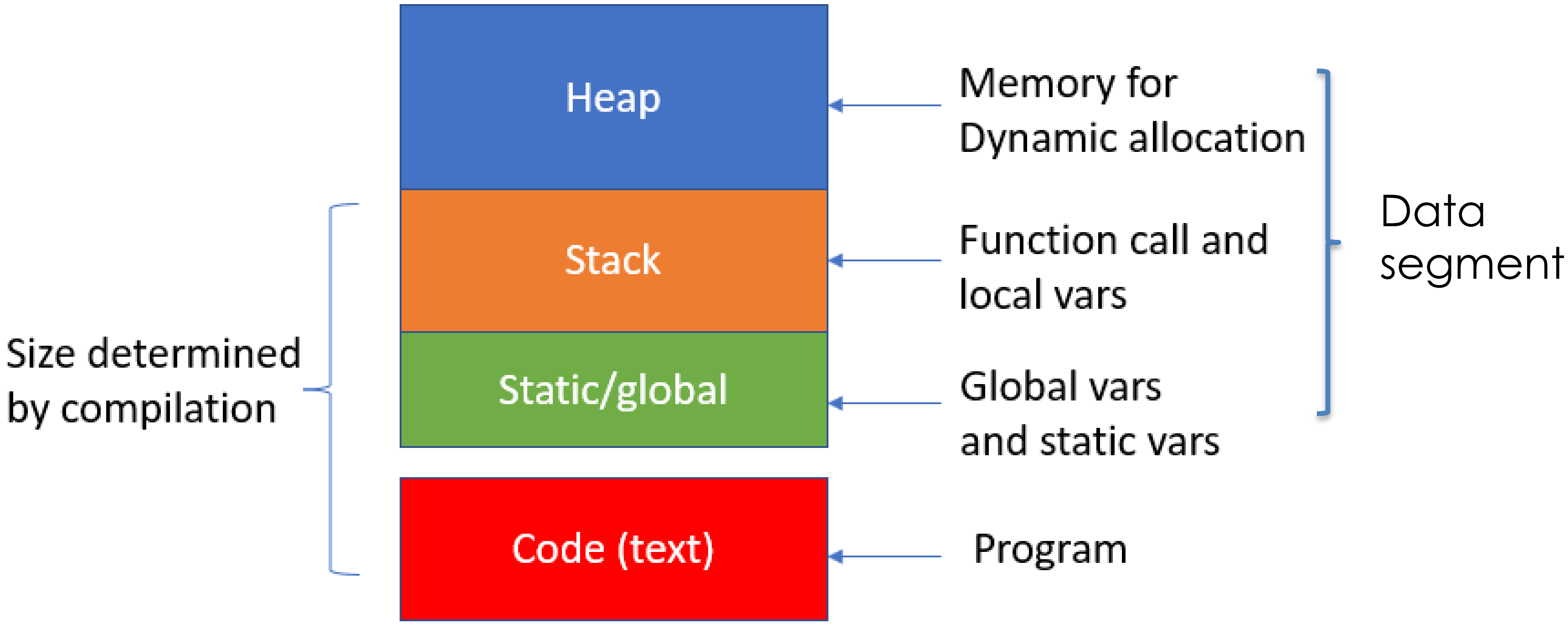
Practically, much smaller address space

- Use address (pointer) to access memory, peripherals and GPIO.

System memory and address

Memory Region	Start Address	Size	Purpose
RTC Slow Memory	0x3FF94000	8 KB	Retains data, low-power modes
RTC Fast Memory	0x3FF80000	8 KB	Retains data during deep sleep
DRAM	0x3FFB0000	320 KB	Data RAM (data, heap, stack)
IRAM	0x40000000	128 KB	Instruction RAM (code)
SPIFFS/LittleFS	Variable	Variable	File system
OTA Data	Variable	4 KB	OTA update metadata
Application Code (App)	0x10000	Variable	Main application firmware
Partition Table	0x8000	3 KB	Memory layout definitions
Bootloader	0x1000	64 KB	Boot code

Memory map for an application



Storage segment for an application

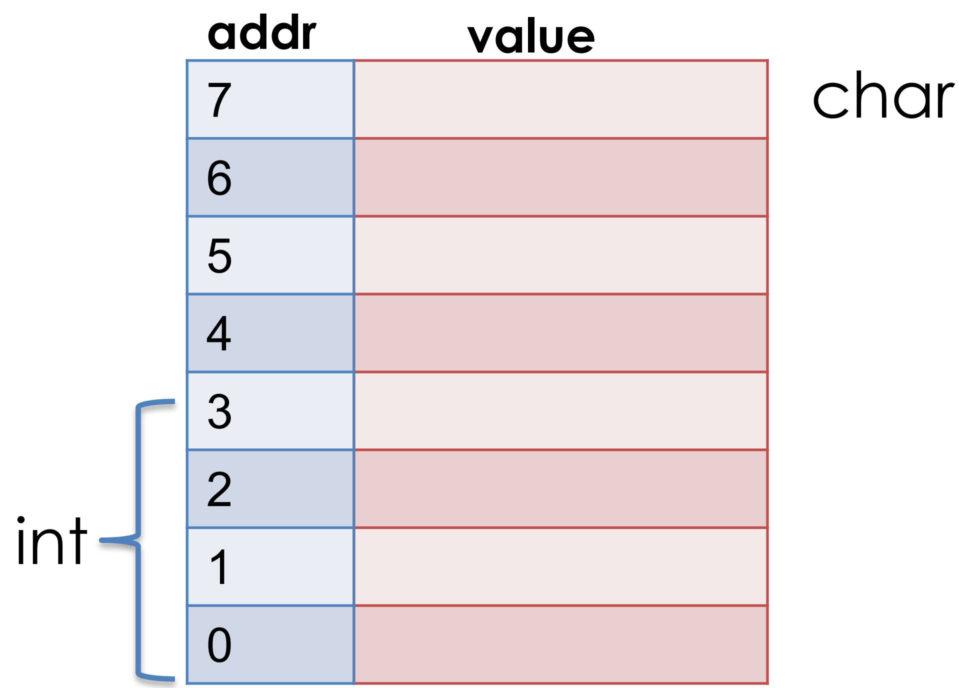
Data types and size

- Data types determine the variable storage, range and related operations.
- Integer types
  - Int: 32-bit, signed -- unsigned int
  - short: 16-bit, signed -- unsigned short
  - long: at least 32-bit. Same as int type in many platforms

    -- unsigned long
  - long long: at least 64-bit signed integer – unsigned long long

Data types and size

- Character types
  - char: 8-bit
  - unsigned char: 8-bit
  - signed char: 8-bit



Data types and size

- Floating-point types
  - float: 32-bit
  - double: 64-bit
- Boolean type
  - bool: only two possible values *true* and *false.*
- Pointer type

Storage classes

- auto: default storage for local variables. It can be omitted.
- static: persistent in storage duration, initial only once.

```
static int counter = 0;
```

- extern: declare a global variable or function in another file.
- const: declare a variable whose value cannot be changed once they are initialized.

Storage classes

```
int globalVar = 0; // Global variable

void count(void)
{   static int staticVar = 0; // Static variable
    staticVar ++;
    Serial.printf("%d\n", staticVar ++);
}

void setup() {
  Serial.begin(115200);
}

void loop() {
    count();
}
```

ESP32 Specific and ESP-IDF Data Types

- Fixed-width integer types
  - int8_t, uint8_t , int16_t, uint16_t , int32_t, uint_32_t, int64_t, uint64_t
- ESP-IDF specific types
  - esp_err_t , gpio_num_t, esp_timer_handle_t
  - wifi_mode_t, wifi_config_t, esp_ip4_addr_t, esp_ip6_addr_t
- ESP-IDF specific structures
  - esp_event_handler_instance_t, esp_log_level_t

ESP32 Specific and ESP-IDF Data Types

- Data structure
  - Queues, semaphores, mutexes
- Peripheral and Sensor Types
  - Specific types for interfacing with hardware peripherals

    i2c_cmd_handle_t for I2C

    adc_channel_t for ADC channels.

Arduino framework types on ESP32

- String: class for dynamic strings. End with ‘ \0 ’.
  - Char s[]=“ESP32”; int c = sizeof(s);
  - String s = “ESP32”; int c = s.length();
- Hardware-specific types
  - pinMode

    {INPUT, OUTPUT, INPUT_PULLUP, INPUT_PULLDOWN}
  - digitalPin   {HIGH, LOW}

Structure

- A structure is a user-defined data type that groups variables of different data types under a single name.
- It helps in organizing related data into a single entity.

```
struct Point {
    int x;
    int y;
};
```

struct

- Representing complex data: useful in representing complex data like coordinates, employee records, or RGB color values.
- Data passing in functions: passing multiple related parameters as a single argument to functions.

```
struct Employee {
    int id;
    char name[50];
    float salary;
};
```

enum

- An enumeration is a user-defined data type that consists of a set of named integer constants.
- enums enhance code readability by assigning names to integral values.

```
enum Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

enum

```
enum TrafficLight {
    Red,
    Yellow,
    Green
};

TrafficLight light = Red;
```

Customized data type name

- Use typedef for a customized number

```
typedef unsigned char uint8_t;


typedef struct {
    float temperature;
    float humidity;
} DHT22Reading;

DHT22Reading tempReading = {25,0.7};
```

Operators

<table>
<tr><td>Arithmetic operators</td><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td></td></tr>
<tr><td>Relational operators</td><td>==</td><td>!=</td><td>></td><td><</td><td>>=</td><td><=</td></tr>
<tr><td>Logical operators</td><td>&&</td><td>||</td><td>!</td><td></td><td></td><td></td></tr>
<tr><td>Bitwise operators</td><td>&</td><td>|</td><td>^</td><td>~</td><td><<</td><td>>></td></tr>
</table>

Bitmasks

```
uint8_t a = 0b10101010;
uint8_t b = 0b11001100;

//bitwise AND operation
uint8_t result = a & b;
```

- A bitmask is a sequence of bits that can manipulate specific bits within another sequence.

```
// Bitmask with lower 4 bits set
uint8_t mask = 0b00001111;
```

Bitmasks

- Setting bits: using bitwise OR operator

```
uint8_t value = 0b00000000;
uint8_t mask =  0b00001111;
value |= mask; // Set lower 4 bits
```

- Clearing bits: using bitwise AND and NOT operators

```
uint8_t value = 0b11111111;
uint8_t mask =  0b11110000;
value &= ~mask; // Clear upper 4 bits
```

Bitmasks

- Toggling bits: using bitwise XOR operator

```
uint8_t value = 0b10101010;
uint8_t mask =  0b11110000;
value ^= mask; // Toggle upper 4 bits
```

- Checking bits: using bitwise AND operator

```
uint8_t value = 0b10101010;
uint8_t mask =  0b00001000;
// Check if 4th bit is set
bool isSet = value & mask;
```

Operators

- Assignment operators

=	+=	-=	*=	/=	%=

&=	|=	^=	<<=	>>=

- Increment and decrement operators

++	--

Operators

- Conditional (ternary) operator
  - Shorthand for if-else statement.

```
max = (a > b) ? a : b;
                Y   N
```

- Comma operator
  - Separate expressions.
  - `x = (y = 2, z = 3, y + z);`
  - `for (int i = 0, j = 10; i < j; i++, j--) { }`

Operators

- Member and pointer operators
  - . (Direct member access)
  - -> (Indirect member access, through pointer)
  - * (Pointer dereference)
  - & (Address of)
- Sizeof operator
  - Size of a data type or object in bytes

```
sizeof(uint8_t);
```

Operators

- Type cast operators
  - Convert a variable from one data type to another.
  - Ensure the correct type and precision in operations when working with mixed data types

```c
int x = 10;  float y = (float)x;

int a = 5;  int b = 2;
int result_i;
result_i = a/b;
float result_f;
result_f = (float)a/b;
```

Control structures

- Control structures guide the flow of a program.
  - Conditional statements
  - Loops
  - Functions
- Essential for decision-making, looping through data, and structuring code.

Control structures

- Statement if, else, and else-if.

```
if (temperature > 30) {
    digitalWrite(fanPin, HIGH);
} else {
    digitalWrite(fanPin, LOW);
}
```

Control structures

- Statement switch-case for multi-way decisions.

```
switch (mode) {
    case 1:
        digitalWrite(ledPin, HIGH);
        break;
    case 2:
        digitalWrite(ledPin, LOW);
        break;
    default:
        break;
}
```

Control structures

- For loop -- iterating a fixed number of times

```
 for (int i = 0; i < 10; i++) {
     Serial.println(i);
     //Repeating actions here.
 }

int sensorPins[] = {A0, A1, A2};
for (int i = 0; i < 3; i++) {
    sensorValues[i] = analogRead(sensorPins[i]);
}
```

Control structures

```
for (;;) {
    digitalWrite(ledPin, HIGH);
    delay(500);
    digitalWrite(ledPin, LOW);
    delay(500);
}
```

Control structures

- While loop -- continuous condition checking

```
while (digitalRead(buttonPin) == LOW) {
    delay(100);
}
```

Control structures

- Do-While loop -- Execute at least once

```
do {
    readSensor();
} while (sensorValue < threshold);
```

Control structures

- Nested control structures

```
for (int i = 0; i < 5; i++) {
    if (sensorArray[i] > threshold) {
        activateAlarm();
    }
}
```

Control structures

- The continue statement -- skips the current iteration and moves to the next iteration of the loop.

```
for (int i = 0; i < 10; i++) {
    if (sensorValues[i] < 0) {
        continue; // Skip negative readings
    }
    process(sensorValues[i]);
}
```

Control structures

- The break statement -- exits the loop immediately, regardless of the remaining iterations.

```
for (int i = 0; i < 10; i++) {
    if (sensorValues[i] > threshold) {
        break; // Stop loop
    }
}
```

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }
    Serial.println(i);
}
```

Control structures

- Loop summary
  - Iterate through sensor pins to collect data.
  - Process the data with conditional checks.
  - Output results, such as activating alarms or lights.

```
for (int i = 0; i < 10; i++) {
    if (i == 2) {
        continue;
    }
    if (i == 8) {
        break;
    }
    Serial.println(i);
}
```

Functions and parameters

- Functions are reusable blocks of code that perform a specific task – Reusability
- Function declaration
  - Tell the compiler the function information
  - Function name – entry point (addr in memory)
  - Return data type – int, char, float, uint8_t or void.
  - Parameters – number and type of expected inputs.

Functions and parameters

- Function definition and call
  - Definition is the actual implementation code.
  - Function call happens when the action of the function is needed.
    - The program jumps to the function definition and executes the code inside it.
    - Return to the next instruction.

Functions and parameters

```
int add(int, int); // function declaration

void setup() {
  int result = add(5, 3); // function call
        .
        .
        .
}

// function definition
int add(int a, int b) {
  return a + b;
}
```

Functions and parameters

- Return types
  - Void: functions do not return a value .

```
void printMessage() {
  Serial.println("Hello, World!");
}
```

  - Non-void.

Functions and parameters

- Passing parameters by value
  - Pass the values of argument into the parameters.
  - Modifications to the parameter inside the function do not affect the argument values.

Functions and parameters

```
void increment(int a) {
  a = a + 1;
}

void setup() {
  int x = 5;
  increment(x);
  Serial.println(x); // Outputs: 5
}
```

Functions and parameters

- Passing parameters by reference
  - Pass the address of the arguments to the function.
  - Modifications to the parameters inside the function affect the arguments.

Functions and parameters

```
void increment(int *a) {
  *a = *a + 1;
}

void setup() {
  int x = 5;
  increment(&x);
  Serial.println(x); //
Outputs: 6
}
```

Functions and parameters

```
void increment(int &a) {
  a = a + 1;
}

void setup() {
  int x = 5;
  increment(x);
  Serial.println(x); // Outputs: 6
}
```

Reference of x

Functions and parameters

- Default parameters
  - Parameters that assume a default value if a value is not provided during the function call.

```
void printMessage(const char* message = "Hi") {
  Serial.println(message);
}
void setup() {
  printMessage();          // Outputs: Hi
  printMessage("Hello"); // Outputs: Hello
}
```

Functions and parameters

- Function overloading
  - Multiple functions can have the same name with different parameters.
  - The correct function is chosen based on the arguments used during the call.

Functions and parameters

```
int add(int a, int b) {
  return a + b;
}

float add(float a, float b) {
  return a + b;
}

void setup() {
 // Calls int version
  Serial.println(add(2, 3));

 // Calls float version
  Serial.println(add(2.5f, 3.5f));
}
```

Functions and parameters

- Use cases of functions
  - Communicate with peripherals.
  - Functions serve as interrupt service routines (ISRs).
  - Use functions to structure large programs into manageable modules
  - …

Functions and parameters

```
void handleButtonPress() {
  if (digitalRead(buttonPin) == LOW) {
    // Handle button press
  }
}
void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
}
void loop() {
  handleButtonPress();
}
```

```
void initializeSensors() { }

void readSensors() { }

void processSensors() { }


void setup() {
  initializeSensors();
}


void loop() {
  readSensors();
  processSensors();
}
```

Inline function

- Inline function - expanded in line where they are called.

```
inline void blinkLED(int pin) {
  digitalWrite(pin, !digitalRead(pin));
}

void toggleLED(int pin) {
  digitalWrite(pin, !digitalRead(pin));
}
```

Inline function

```
inline returnType functionName(parameters) {
  // function body
}
```

- Inline functions improve performance by reducing call overhead.
- Suitable for small, simple functions and not for large functions.
- Examples: LED blink, sensor reading, mathematical operations.

Lambda function

- Defined in C++ standard.
- Anonymous: Lambda functions are not bound to a name.
- Inline: They are typically defined inline where they are needed.
- Short: Usually used for short, simple operations.
- Single-use functions and simplifying code.
- Passing functions as arguments

Lambda function

```
Sytanx:
[ captures ] ( parameters ) -> return_type {
    // function body
};

[captures]: specify variables from the surrounding scope the
lambda function can access. It can be left empty ([]).
[&]: Capture all variables by reference.
[=]: Capture all variables by value.
[this]: Capture the this pointer, if the lambda is inside a class.

(parameters): The list of parameters the lambda function takes,
similar to a normal function.

-> return_type: (Optional) Specifies the return type. If omitted,
the compiler will try to deduce the return type.
```

Lambda function examples

```
//example of using a lambda function to add two numbers.
auto add = [](int a, int b) -> int {
    return a + b;
};

int result = add(3, 4); // result will be 7
```

Lambda function examples

```
//example of using a lambda function in AsyncWebServer.

server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/plain", "Hello, world!");
});
```

`[]`: No variables from the surrounding scope are captured.

`(AsyncWebServerRequest *request)`: The lambda function takes a single parameter, which is a pointer to an AsyncWebServerRequest object.

`{}`: The body of the lambda function. Inside the body, it sends a response to the client with an HTTP status code of 200 and the text "Hello, world!".

Lambda function examples

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/plain", "Hello, world!");
});
```

The equivalent normal function is:

```
void handleRootRequest(AsyncWebServerRequest *request) {
    request->send(200, "text/plain", "Hello, world!");
}

server.on("/", HTTP_GET, handleRootRequest);
```

Interrupts

- Interrupts are signals that temporarily halt the current code execution.
- Used to handle events like button presses, timers, and sensor data.
- Allow MCU to respond to external events immediately.

Interrupts

- Types of Interrupts
  - Hardware Interrupts
    - Triggered by external hardware events.
  - Software Interrupts
    - Triggered by software instructions.

Interrupts

- Pin change interrupts
  - Triggered when the state of a digital pin changes.
  - Example: Button press or release.
  - Functions: attachInterrupt(), detachInterrupt()

Interrupts

- Configuring interrupts

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

pin: Pin number.
ISR: Interrupt Service Routine function.

mode: Trigger mode (RISING, FALLING, CHANGE).

Interrupts

- Interrupt Service Routine (ISR)
  - A function that executes when an interrupt occurs.
  - Short and efficient.

```
void IRAM_ATTR ISR() {
  // Code to execute
}
```

Interrupts example

```
const int buttonPin = 4;
volatile bool buttonPressed = false;

void IRAM_ATTR handleButtonPress() {
  buttonPressed = !buttonPressed;  // response
}

void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(buttonPin),
                                 handleButtonPress, FALLING);
}

void loop() {
  if (buttonPressed) {
    // Handle button press
    buttonPressed = false;
  }
}
```

Interrupts

- Debouncing with interrupts
  - Mechanical buttons can generate multiple signals (bounce).
  - Use software debouncing to handle bounce.

```
void IRAM_ATTR handleButtonPress() {
  static unsigned long lastInterruptTime = 0;
  unsigned long interruptTime = millis();
  if (interruptTime - lastInterruptTime > 200) {
    buttonPressed = !buttonPressed;
  }
  lastInterruptTime = interruptTime;
}
```

Interrupts

- Timer interrupts
  - Timer interrupts are triggered by hardware timers.
  - Useful for periodic tasks.

```
hw_timer_t *timer = NULL;

void IRAM_ATTR onTimer() {
  // Timer ISR code
}

void setup() {
  timer = timerBegin(0, 80, true);
  timerAttachInterrupt(timer, &onTimer, true);
  timerAlarmWrite(timer, 1000000, true);
  timerAlarmEnable(timer);
}
```

Pointer

- A pointer is a **variable** that stores the memory address of another object.
- Allows direct access to memory.
- Declare pointers `type *pointerName;`
  - Type: Depends on the variable type a pointer pointing to.
  - Storage size: same for all pointers. 32-bit in 32-bit system.

```
int *ptr;
```

```
int value = 10;
int *ptr = &value;
```

Pointer

- Referencing and dereferencing for pointers
  - Referencing (&) : Obtain the address of a variable (and store in a pointer).
  - Dereference (*): Get the value stored at the address that the pointer is pointing to.

```
int value = 10;
int *ptr = &value;
int anotherValue = *ptr; //anotherValue is 10
```

Pointer

addr	content
0107	0000 0000
0106	0000 0000
0105	0010 1010
0104	0100 0001
0103	0000 0000
0102	0000 0000
0101	0000 0100
0100	0000 0001

- Pointer Arithmetic
  - Increase/decrease the pointer (`int *p1`)

```
p1++; or p1+=1; or p1=p1+1;
p1--; or p1-=1; or p1=p1-1;
```

```
If p1 = 0x0104, what is *p1 after decrement?
```

  - Cast pointer (assume `p1 = 0x0104`)

```
char *p2; void *p;
p2 = (char *)p1; //what is *p2?
p2++;  //what is *p2?
p1--; p = p1; //what is *p?
```

Pointer

- Pointer and arrays
  - Arrays and pointers are closely related.

```
int arr[] = {1, 2, 3};
int *ptr = arr; // ptr points to arr[0]
// *(arr+1) = 2 and *(arr+2) = 3
// arr ++; compiler error
```

  - Perform arithmetic operations to navigate through arrays.

```
int *ptr = arr;
ptr++; // Points to the second element
int secondElement = *ptr; //secondElement is 20
```

Dynamic Memory Allocation

- Dynamic Memory Allocation
  - Allocate memory dynamically using new and delete.

```
int *ptr = new int; //allocate from heap
*ptr = 5;
.
.
.
delete ptr; // necessary to avoid leakage
```

Pointer examples

- Pointers create dynamic data structures like linked lists.

```
struct Node {
  int data;
  Node *next;
};

Node *head = new Node();
head->data = 1;
head->next = new Node();
head->next->data = 2;
```

Pointer examples

- Use pointers to pass large structures to functions to avoid copying.

```
struct SensorData {
  int temperature;
  int humidity;
};

void processSensorData(SensorData *data) {
  Serial.println(data->temperature);
}

void setup() {
  SensorData data = {25, 60};//initiate the structure
  processSensorData(&data);
}
```

Pointer examples

- Use pointers to manipulate C-style strings.

```
char str[] = "Hello";
char *ptr = str;
while(*ptr != '\0') {
  Serial.println(*ptr);
  ptr++;
}
```

Pointers to function

- Pointers to Functions
  - Pointer points to the entry address of a function and call the function through the pointer.

```
return_type (*pointer_name)(parameter_types);
```

```
void myFunction() {    // define myFunction
  Serial.println("Hello");
}
//define and initialize a function pointer
void (*funcPtr)() = myFunction;
funcPtr(); // Calls myFunction
```

Pointers to function

- Pointers to Functions
  - Pointer points to the entry address of a function and call the function through the pointer.

```
return_type (*pointer_name)(parameter_types);
```

```
void myFunction() {    // define myFunction
  Serial.println("Hello");
}
//define and initialize a function pointer
void (*funcPtr)() = myFunction;
funcPtr(); // Calls myFunction
```

Pointers to function

- Simplify the declaration of function pointer

```
typedef return_type (*typedef_name)(parameter_types);
```

```
int add(int a, int b) {
    return a + b;
}

typedef int (*Operation)(int, int);

Operation op = &add; // or just = add; in C
int result = op(3, 4); // result will be 7
```

Callback functions

- A callback function is a function passed as an argument to another function.
- Used to handle asynchronous events.

```
void callback() {
  Serial.println("Callback function called");
}

void registerCallback(void (*func)()) {
  func();
}

void setup() {
  registerCallback(callback);
}
```

Callback functions examples

- Use function pointers to handle timer interrupts.

```
void onTimer() {
  Serial.println("Timer interrupt");
}

hw_timer_t *timer = NULL;

void setup() {
  Serial.begin(115200);
  timer = timerBegin(0, 80, true);
  timerAttachInterrupt(timer, &onTimer, true);
  timerAlarmWrite(timer, 1000000, true);
  timerAlarmEnable(timer);
}
```

Callback functions examples

- Use function pointers to implement a state machine.

```
void state1() {
  Serial.println("State 1");
}

void state2() {
  Serial.println("State 2");
}

void (*state)() = state1;

void setup() {
  Serial.begin(115200);
}
void loop() {
  state();
  delay(1000);
  state = (state == state1) ? state2 : state1;
}
```

```
int compare(int a, int b) {
  return (a > b) - (a < b);
}

void sort(int *arr, int size, int (*cmp)(int, int)) {
  // Simple bubble sort for demonstration
  for (int i = 0; i < size - 1; i++) {
    for (int j = 0; j < size - 1 - i; j++) {
      if (cmp(arr[j], arr[j + 1]) > 0) { //call function by pointer
        int temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
}

void setup() {
  int arr[] = {5, 3, 8, 6, 2};
  sort(arr, 5, compare);
  for (int i = 0; i < 5; i++) {
    Serial.println(arr[i]);
  }
}
```

Callback functions examples

- Use callback functions for handling network communication.

```
void onReceive(char* data) {
  Serial.print("Received: ");
  Serial.println(data);
}

void setup() {
  Serial.begin(115200);
  // Assume WiFi or Ethernet setup
  attachNetworkCallback(onReceive);
}
```

Callback functions examples

- Use callback functions for custom serial protocol handling.

```
void onSerialDataReceived(String data) {
    Serial.print("Received data: ");
    Serial.println(data);
}

void attachSerialCallback(void (*callback)(String)) {
    if (Serial.available() > 0) {
        String data = Serial.readStringUntil('\n');
        callback(data);
    }
}

void setup() {
    Serial.begin(115200);
    delay(1000);  // Give some time for the Serial Monitor to start

    Serial.println("Serial Callback Example");
}

void loop() {
    // Attach the callback function for serial data reception
    attachSerialCallback(onSerialDataReceived);
}
```

Use of function pointer

- Ensure function signatures match.
- Use clear and descriptive names.
- Avoid excessive use of function pointers for better readability.
- Ensure proper documentation.

Potential issues for pointers

- Dangling pointers point to deallocated memory.
- Pointer out of bound or pointing to some unknow area which may not be identified by compiler and cause some safety issue.
- Pointer to misaligned data memory.
- Memory leakage if dynamically allocated memory is not released.