MEMORY MANAGEMENT

Programming with the Arduino

JUNE 16, 2025

NSCC ONLINE CAMPUS
Peter Vaughan's Self-Learning

Contents

Overview

This paper builds upon my previous work and code available in my GitHub repository on memory management in embedded systems. In this installment, the focus is specifically on memory management in Arduino environments. While the topics discussed range from beginner to advanced, memory management is a fundamental concept that all embedded systems programmers should understand to write efficient and reliable code.

I am passionate about learning and sharing knowledge. Feel free to reach out if you are interested in programming or embedded systems. The repository listed below will continue to be updated regularly until it is eventually archived. If you notice any issues or missing information, do not hesitate to let me know.

Additional Info

My name is Peter and here is how to contact me or to look at my other content:

GitHub Repo: https://github.com/Vaughan-Peter/ArduinoLearning

LinkedIn Profile: https://www.linkedin.com/in/peter-vaughan-997478239/

Contact E-mail: otherhalifaxprojects@gmail.com

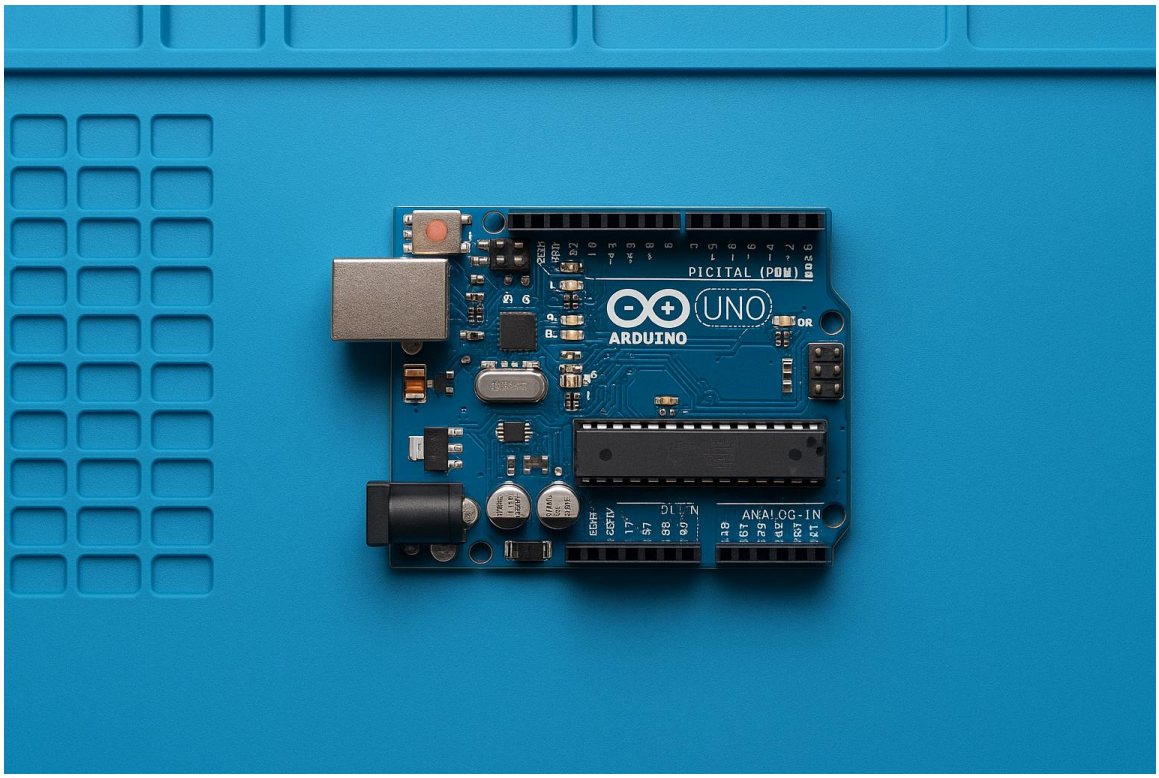
YouTube: https://www.youtube.com/@Learning-Arduino-Concepts/videos

Arduino Memory Management

Memory management is important in Arduino programming because these microcontrollers have very limited memory resources**.** Efficient memory use ensures your program runs reliably without crashing or behaving unpredictably. Poor memory management can lead to issues like stack overflows, heap fragmentation**,** and unexpected resets**,** especially when dealing with dynamic memory or large data sets. By managing memory wisely, you can build more stable, scalable, and power-efficient embedded systems.

Memory management on the Arduino is crucial due to its limited RAM and storage. Most Arduino boards (like the Uno) have:

- Flash memory (for storing the program/sketch): 32 KB
- SRAM (for variables and runtime data): 2 KB
- EEPROM (for long-term data storage): 1 KB



Types of Memory in Arduino

Arduino uses three main types of memory: Flash, SRAM, and EEPROM, each serving a distinct purpose. Flash memory stores the program code and constant data and, while it retains data after a reset, it cannot be written to during program execution. SRAM is used for variables and the program stack but is volatile, while EEPROM allows for saving data that must persist even after the board is powered off or reset.

Memory Type	Used For	Writable During Program?	Data Persists After Reset?
Flash	Storing code (sketch) and PROGMEM data	No	Yes
SRAM	Runtime variables, stack, heap	Yes	No
EEPROM	Long-term data storage	Yes	Yes

Common Memory Management Techniques

Efficient memory use is crucial while using the Arduino, so it is best to avoid dynamic allocation and instead use static memory to prevent fragmentation. To save SRAM, store constant strings in flash using the `F()` macro or `PROGMEM`, and use EEPROM when you need data to persist after power loss.

1. Avoid Dynamic Memory Allocation

Arduino's new, malloc(), and free() are risky in small SRAM systems due to fragmentation. Stick with static allocation when possible.

```
// Bad practice (risk of fragmentation)
char* buffer = (char*)malloc(50);
```

2. Use F() Macro to Save SRAM

```
Serial.print(F("This string stays in flash memory!"));
```

3. Use PROGMEM to Store Large Data in Flash

```
const char message[] PROGMEM = "Hello from flash!";
```

4. Use EEPROM for Persistent Storage

```
#include <EEPROM.h>

EEPROM.write(0, 42);       // Save a value
int val = EEPROM.read(0);  // Read it back
```

Example: Monitoring Free SRAM

This function estimates the available SRAM at runtime by measuring the gap between the heap and the stack. It is useful for monitoring memory usage and catching potential memory issues during program execution.

```
int freeMemory() {
  extern int __heap_start, *__brkval;
  int v;
  return (int)&v - (__brkval == 0 ? (int)&__heap_start : (int)__brkval);
}
```

Usage:

```
void loop() {
  Serial.print("Free memory: ");
  Serial.println(freeMemory());
  delay(1000);
}
```

Example Project: Sensor Data Logger with EEPROM Backup

This code reads an analog sensor value and saves it to EEPROM every 5 seconds, scaling the 10-bit reading (0–1023) down to 8 bits (0–255). An LED connected to pin 13 briefly lights up each time a value is saved to indicate a write operation. Using `EEPROM.write()` allows the sensor data to persist even after the Arduino is powered off or reset.

```
#include <EEPROM.h>

const int sensorPin = A0;
const int ledPin = 13;

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  int sensorValue = analogRead(sensorPin);
  Serial.println(sensorValue);

  // Save value every 5 seconds
  static unsigned long lastSave = 0;
  if (millis() - lastSave > 5000) {
    EEPROM.write(0, sensorValue / 4); // scale 0–1023 to 0–255
    lastSave = millis();
    digitalWrite(ledPin, HIGH);
    delay(100);
    digitalWrite(ledPin, LOW);
  }

  delay(1000);
}
```

Tips for Better Memory Management

One tip is to use byte, int8_t, or uint8_t over int for small-range values. Another tip is to reuse buffers and avoid large global arrays unless necessary. Also, regularly check memory usage if the sketch grows.

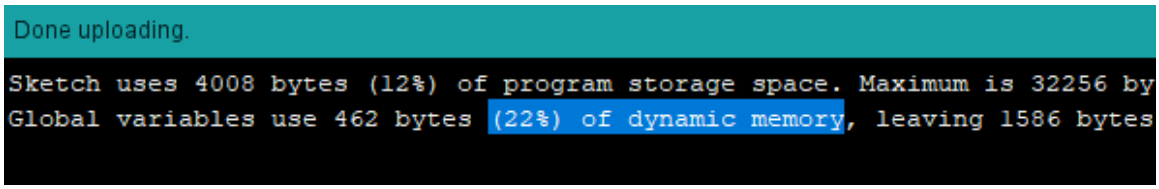
Here are some tips for better memory management with the Arduino:

- **Use smaller data types** like byte or uint8_t instead of int when possible.
- **Avoid dynamic memory allocation** (e.g., malloc, new) to prevent fragmentation.
- **Reuse buffers** and limit the use of large global arrays.
- **Move constant strings to flash memory** using the F() macro.
- **Store large constant data in PROGMEM** or EEPROM when needed.
- **Monitor free SRAM** during development to catch memory issues early.

Ways to Check Memory Usage

1. Check Memory at Compile Time (IDE Output)

When you compile a sketch in the Arduino IDE, it shows memory usage:



- **Program storage space** → Flash memory (code and constants)
- **Dynamic memory** → SRAM (variables, stack, heap)

Tip: Keep SRAM usage (dynamic memory) below ~70–80% to avoid runtime instability. I learned about instability the difficult way with a big project while working with Arduino Uno.

2. Check Free SRAM at Runtime (Function Method)

Use this function to estimate **free SRAM** while the sketch runs:

```
int freeMemory() {
  extern int __heap_start, *__brkval;
  int v;
  return (int)&v - (__brkval == 0 ? (int)&__heap_start : (int)__brkval);
}
```

Example usage:

```
void loop() {
  Serial.print("Free memory: ");
  Serial.println(freeMemory());
  delay(1000);
}
```

This helps catch memory leaks or fragmentation issues.

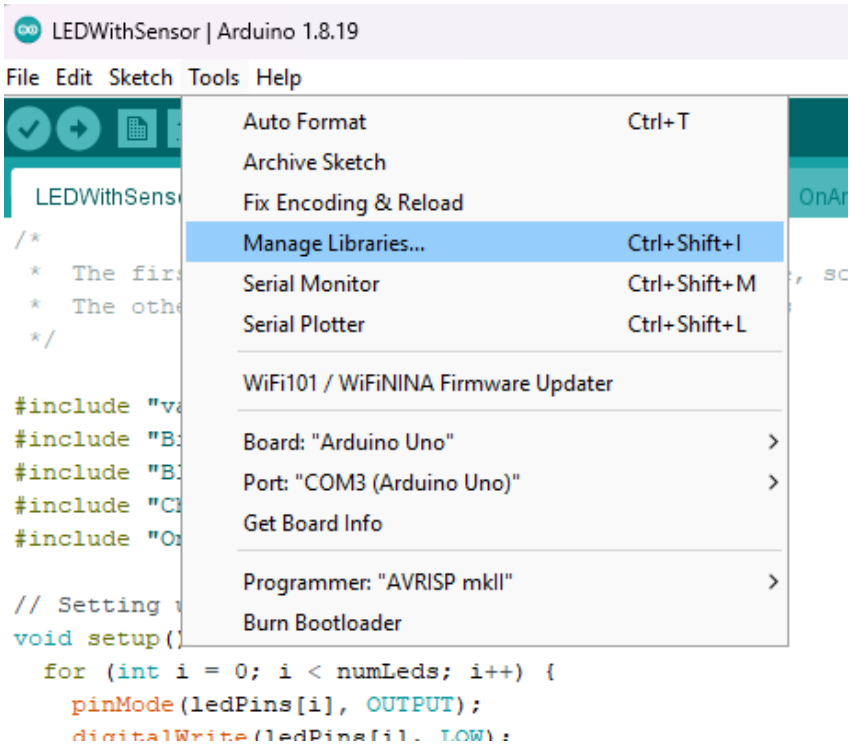
3. Use the Arduino MemoryFree Library (Optional)

For ease, install the MemoryFree library which wraps the same function:

```
#include <MemoryFree.h>

void loop() {
  Serial.print("Free memory: ");
  Serial.println(freeMemory());
}
```

You can install it via the Library Manager in the Arduino IDE.



Bonus: Analyze Memory Map with avr-size (Advanced)

If you are compiling using a Makefile or PlatformIO, you can run:

sh

```
avr-size -C --mcu=atmega328p your_sketch.elf
```

This shows a detailed breakdown of Flash, SRAM, and EEPROM usage.

Data Sorting

In one of my earlier papers, “Public General Use – Thinking of Teaching Arduino,” I explored topics that are important for teaching beginners about embedded systems, including memory management. The programming concepts discussed in that paper can be integrated into Arduino education. For instance, using a few string literals is acceptable in Arduino programming. However, opting for character pointers is often more efficient. Some memory management topics can be overwhelming for beginners, especially those with limited experience. Concepts like data structures can become particularly challenging. This is due to the memory and hardware constraints of the Arduino platform.

Sorting information involves organizing data in a specific order, such as ascending or descending numerical values or alphabetical names. The range of sorting can vary from small fixed-size arrays on microcontrollers to large datasets processed on cloud servers. On devices like Arduino, sorting is usually limited to a few dozen items due to memory constraints. More advanced systems can sort thousands or millions of records efficiently using optimized algorithms like Quick Sort or Merge Sort. Data can come from a variety of sources, including sensors, user input, serial communication, files, or wireless networks. The type and size of data impact the choice of sorting method—simple methods like Bubble Sort work well for small data, while more efficient algorithms are preferred for larger sets. Regardless of the system, sorting helps improve data organization, search speed, and decision-making accuracy.