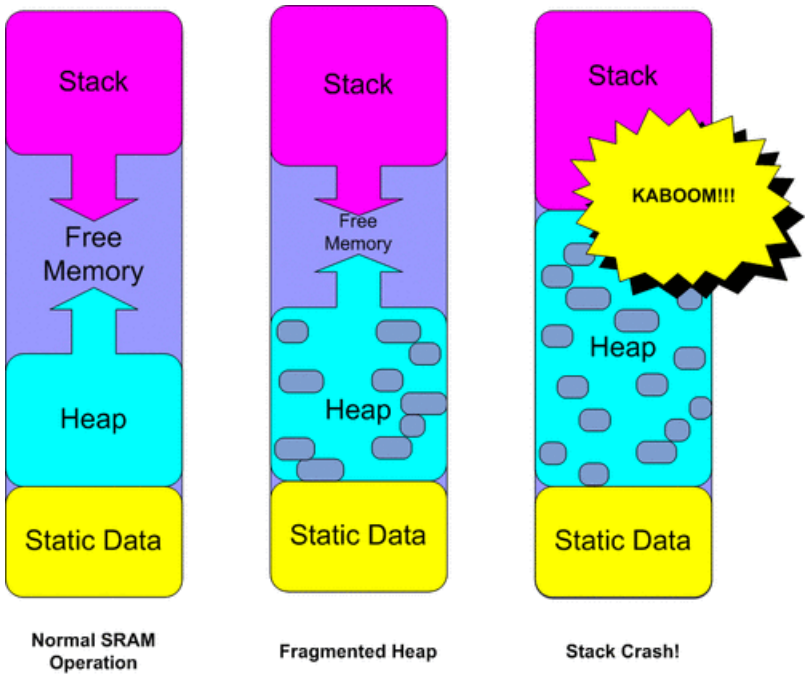
adafruit learning system

Memories of an Arduino

Created by Bill Earl



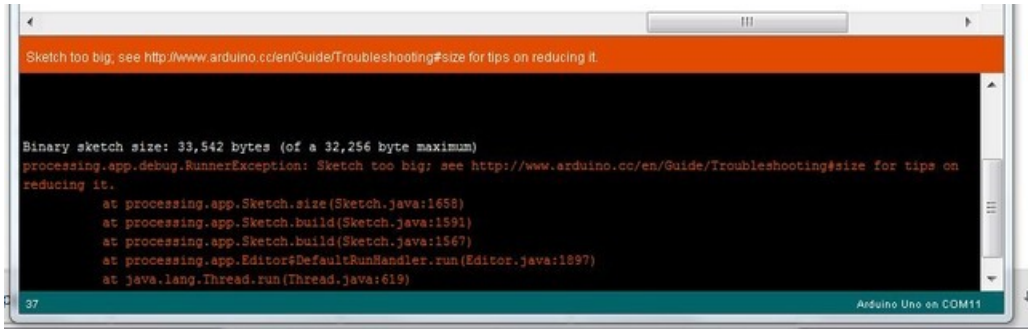
Last updated on 2018-08-22 03:36:50 PM UTC

Guide Contents

You know you have a memory problem when...

> "The memory is the first thing to go."
> (I don't remember who told me that)

The most obvious sign of a memory problem is when the compiler tells you that your sketch is too big.



But many memory problems show much more subtle symptoms. Your program may load, but not run. It may crash hard, or just start acting funky.

If your program compiles and loads successfully, but any of the following statements are true, there is a good chance that you have a memory problem.

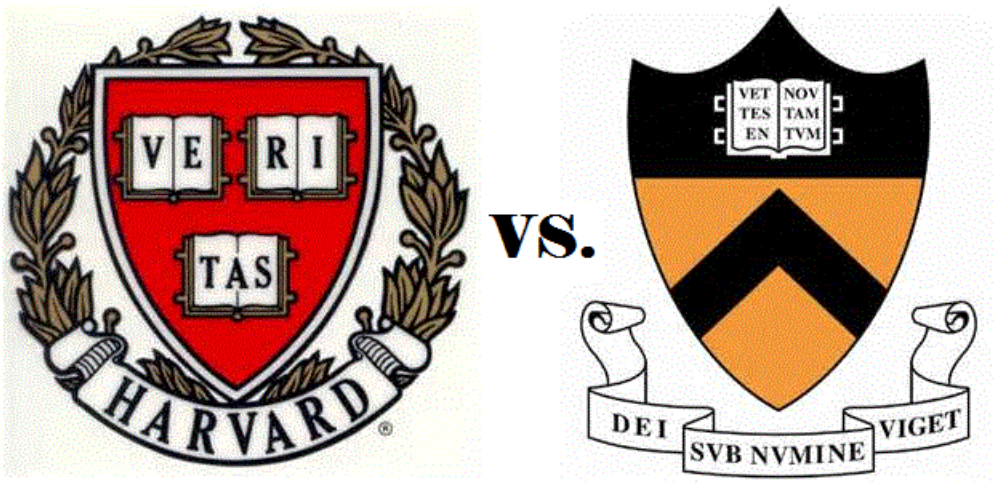
> ***"My program worked fine until I"*** *(choose one)*
>
> - ***"Included one more library"***
> - ***"Added some more LED pixels"***
> - ***"Opened a file on the SD card"***
> - ***"Initialized a graphical display"***
> - ***"Merged in another sketch"***
> - ***"Added a new function"***

If you think you might have a memory problem, you can skip right to the "Solving Memory Problems" page. But you should first take a look through the next few pages to better understand Arduino memory and how it works.

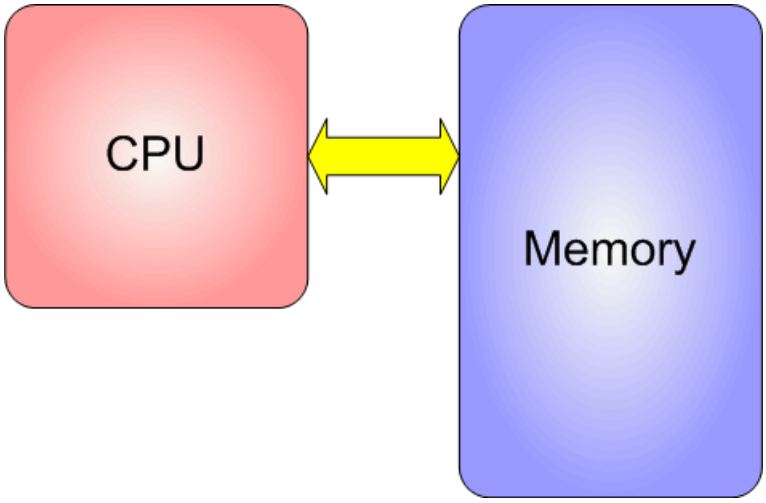
Memory Architectures

Harvard vs Princeton

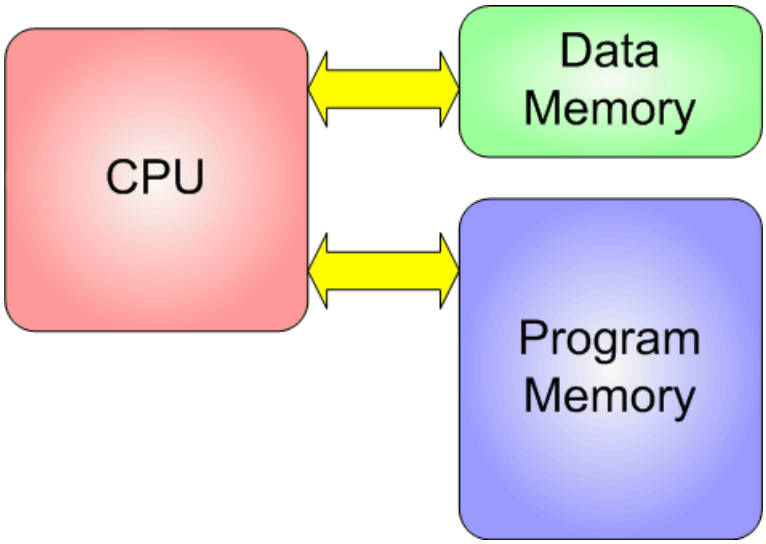
In the early days of electronic computing, two different processor/memory architectures emerged:



The **Von Neumann (https://adafru.it/coe)** (a.k.a. (https://adafru.it/coe)**Princeton (https://adafru.it/coe)**) (https://adafru.it/coe)**architecture (https://adafru.it/coe)** developed for the **ENIAC (https://adafru.it/cof)**uses the same memory and data paths for both program and data storage.



The **Harvard architecture (https://adafru.it/cog)** characterized by the **Harvard Mark 1 (https://adafru.it/coh)** used physically separate memory and data paths for program and memory.



Which is better?

Each architecture has its advantages: All else being equal, the Harvard model has the edge in performance. The Von Neumann model is more flexible.

Modern Hybrids

These days, most general purpose computers (PC's Mac's etc.) are hybrid designs that give you the best of both architectures. Deep within the CPU they operate on the Harvard model using separate caches for instructions and data to maximize performance. But the instruction and data caches are both loaded automatically from a common memory space. From a programming perspective, these computers appear to be pure Von Neumann machines with many gigabytes of virtual storage.

Microcontrollers

Microcontrollers such as the ones that power the Arduinos are designed for embedded applications. Unlike general purpose computers, an embedded processor typically has a well defined task that it must perform reliably and efficiently - and at minimal cost, Microcontroller designs tend to be rather spartan. They forego the luxuries of multi-layer caching and disk-based virtual memory systems and stick to what is essential to the task.

The Harvard model turns out to be a good match for embedded applications and the Atmega 328 used in the Arduino UNO use a relatively pure Harvard architecture. Programs are stored in Flash memory and data is stored in SRAM.

For the most part, the compiler and run-time systems take care of managing these for you, but when things start getting tight, it helps to be aware of how things work under the hood. And things start getting tight much quicker on these tiny machines!

A completely different Scale

The biggest difference between these microcontrollers and your general purpose computer is the sheer amount of memory available. The Arduino UNO has only 32K bytes of Flash memory and 2K bytes of SRAM. That is more than **100,000 times LESS physical memory than a low-end PC!** And that's not even counting the disk drive!

Working in this minimalist environment, you must use your resources wisely.

Arduino Memories

There are 3 types of memory in an Arduino:

- **Flash or Program Memory**
- **SRAM**
- **EEPROM**

Flash Memory

Flash memory is used to store your program image and any initialized data. You can execute program code from flash, but you can't modify data in flash memory from your executing code. To modify the data, it must first be copied into SRAM

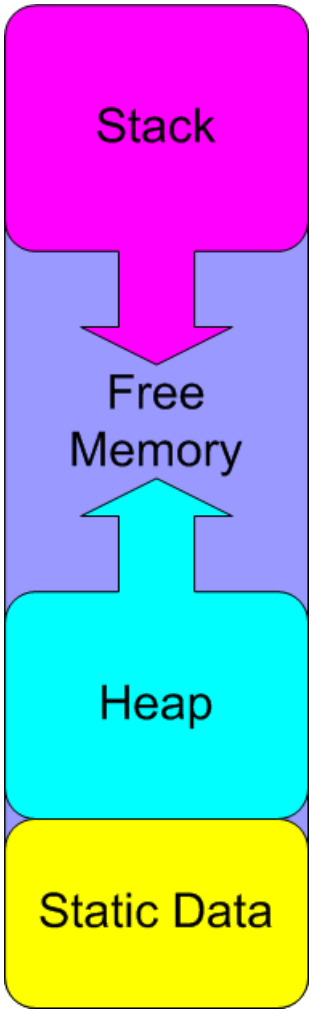
Flash memory is the same technology used for thumb-drives and SD cards. It is non-volatile, so your program will still be there when the system is powered off.

Flash memory has a finite lifetime of about 100,000 write cycles. So if you upload 10 programs a day, every day for the next 27 years, you might wear it out.

SRAM

SRAM or **Static Random Access Memory**, can be read and written from your executing program. SRAM memory is used for several purposes by a running program:

- **Static Data** - This is a block of reserved space in SRAM for all the global and static variables from your program. For variables with initial values, the runtime system copies the initial value from Flash when the program starts.
- **Heap** - The heap is for dynamically allocated data items. The heap grows from the top of the static data area up as data items are allocated.
- **Stack** - The stack is for local variables and for maintaining a record of interrupts and function calls. The stack grows from the top of memory down towards the heap. Every interrupt, function call and/or local variable allocation causes the stack to grow. Returning from an interrupt or function call will reclaim all stack space used by that interrupt or function.



Most memory problems occur when the stack and the heap collide. When this happens, one or both of these memory areas will be corrupted with unpredictable results. In some cases it will cause an immediate crash. In others, the effects of the corruption may not be noticed until much later.

EEPROM

EEPROM is another form of non-volatile memory that can be read or written from your executing program. It can only be read byte-by-byte, so it can be a little awkward to use. It is also slower than SRAM and has a finite lifetime of about 100,000 write cycles (you can read it as many times as you want).

While it can't take the place of precious SRAM, there are times when it can be very useful!

Arduino Memory Comparision

The following chart shows the amounts of each type of memory for several Arduino and Arduino compatible boards.

Arduino	Processor	Flash	SRAM	EEPROM
UNO, Uno Ethernet, Menta, Boarduino	Atmega328	32K	2K	1K
Leonardo, Micro, Flora, 32U4 Breakout, Teensy, Esplora	Atmega 32U4	32K	2.5K	1K
Mega, MegaADK	Atmega2560	256K	8K	4K

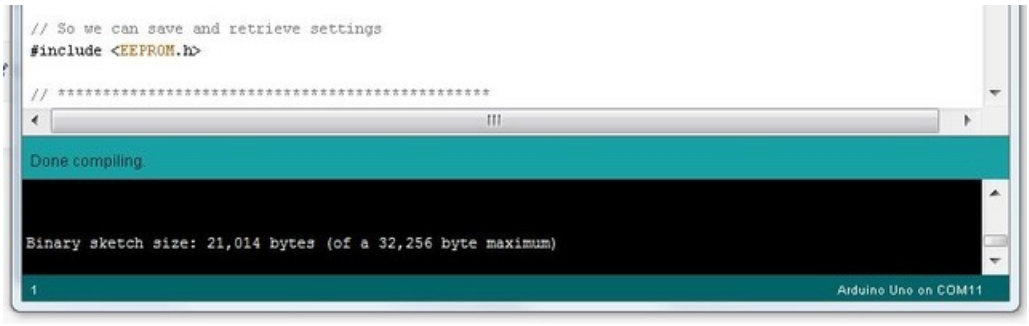
"...that's not got much SRAM in it."

Measuring Memory Usage

One way to diagnose memory problems is to measure how much memory is in use.

Flash

Measuring Flash memory usage is trivial. The compiler does that for you, every time you compile!



EEPROM

You are 100% in control of EEPROM usage. You have to read and write each byte to a specific address, so there is no excuse for not knowing exactly which bytes are in use!

```
// *************************************************
// Write floating point values to EEPROM
// *************************************************
void EEPROM_writeDouble(int address, double value)
{
   byte* p = (byte*)(void*)&value;
   for (int i = 0; i < sizeof(value); i++)
   {
      EEPROM.write(address++, *p++);
   }
}

// *************************************************
// Read floating point values from EEPROM
// *************************************************
double EEPROM_readDouble(int address)
{
   double value = 0.0;
   byte* p = (byte*)(void*)&value;
   for (int i = 0; i < sizeof(value); i++)
   {
      *p++ = EEPROM.read(address++);
   }
   return value;
}
```

SRAM

SRAM usage is more dynamic and therefore more difficult to measure. The `freeMemory()` function below is one way to do this. You can add this function definition to your code, then call it from various places in your code to report the amount of free SRAM.

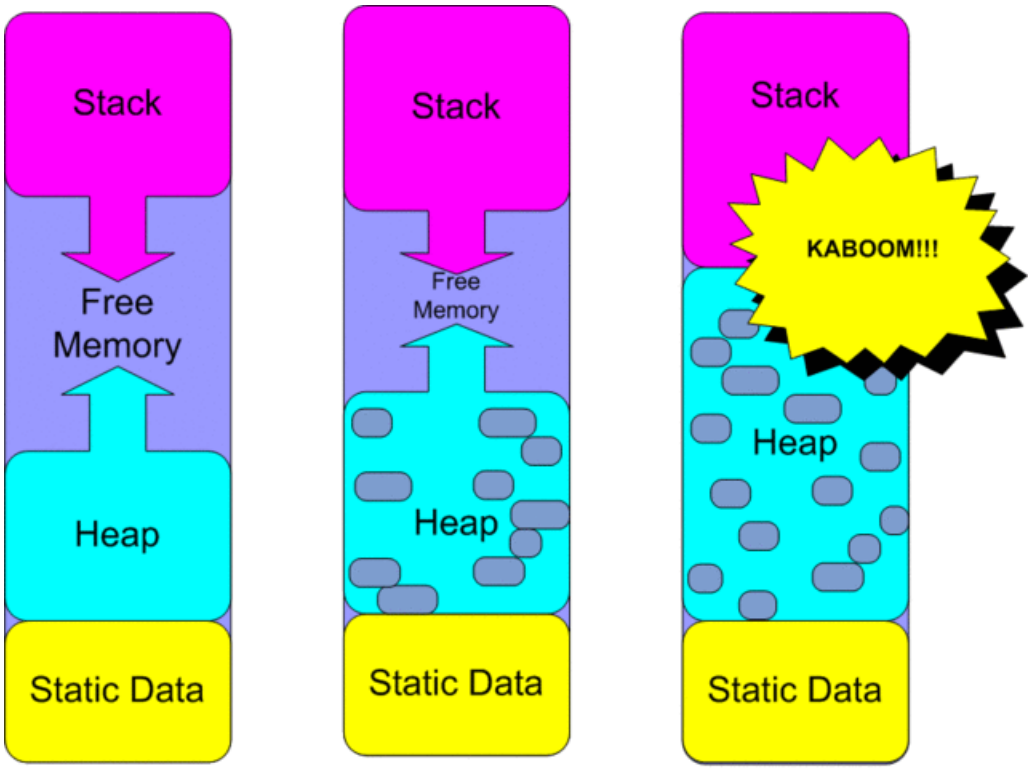
SRAM utilization is dynamic and will change over time. So It is important to call `freeMemory()` at various times and from various places in your sketch to see how it changes over time.

This code is taken from this small library: https://github.com/mpflaga/Arduino-MemoryFree (https://adafru.it/CcO) and works on both AVR and ARM (M0) processors.

```
#ifdef __arm__
// should use uinstd.h to define sbrk but Due causes a conflict
extern "C" char* sbrk(int incr);
#else  // __ARM__
extern char *__brkval;
#endif  // __arm__

int freeMemory() {
  char top;
#ifdef __arm__
  return &top - reinterpret_cast<char*>(sbrk(0));
#elif defined(CORE_TEENSY) || (ARDUINO > 103 && ARDUINO != 151)
  return &top - __brkval;
#else  // __arm__
  return __brkval ? &top - __brkval : &top - __malloc_heap_start;
#endif  // __arm__
}
```

What `freeMemory()` is actually reporting is the space between the heap and the stack. it does not report any de-allocated memory that is buried in the heap. Buried heap space is not usable by the stack, and may be fragmented enough that it is not usable for many heap allocations either. The space between the heap and the stack is what you really need to monitor if you are trying to avoid stack crashes.



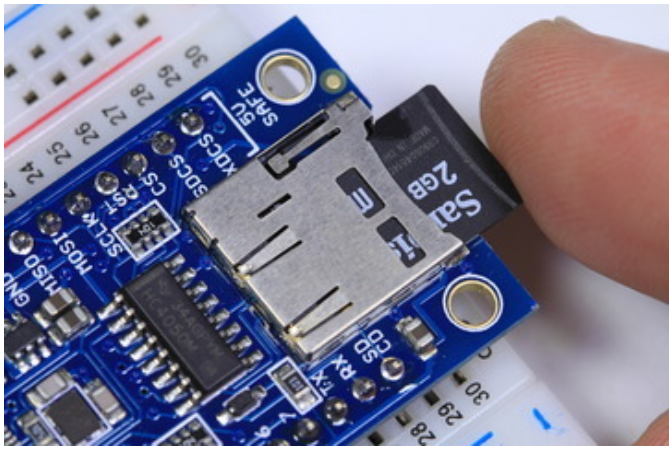
Normal SRAM Operation

Fragmented Heap

Stack Crash!

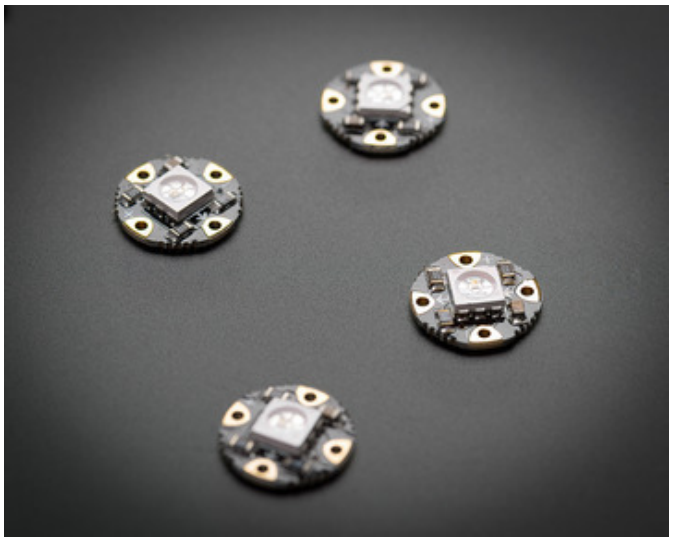
Large Memory Consumers

There are devices and drivers which require large amounts of SRAM to operate. Some of the largest memory consumers are:



SD Cards

Anything with an SD or Micro-SD interface requires a 512 byte SRAM buffer to communicate with the card.



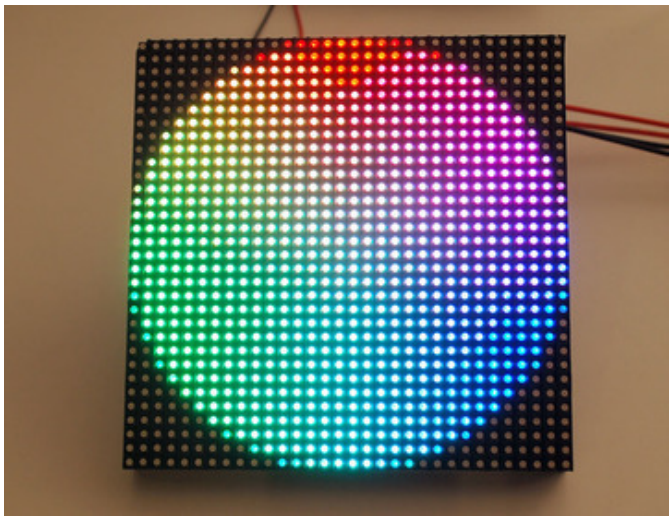
Pixels

Each pixel requires just 3 bytes of SRAM to store the color. But those bytes start to add up when you have many meters of strip or a large array.

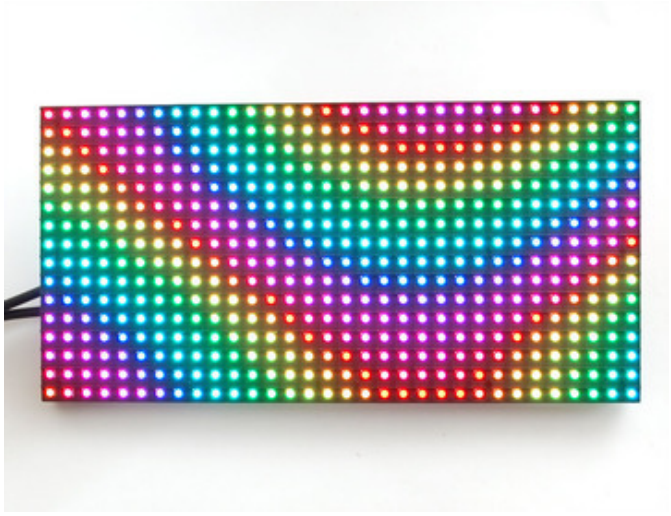
On an Uno, you might be able to drive as many as 500 pixels - assuming you don't use much SRAM for anything else.

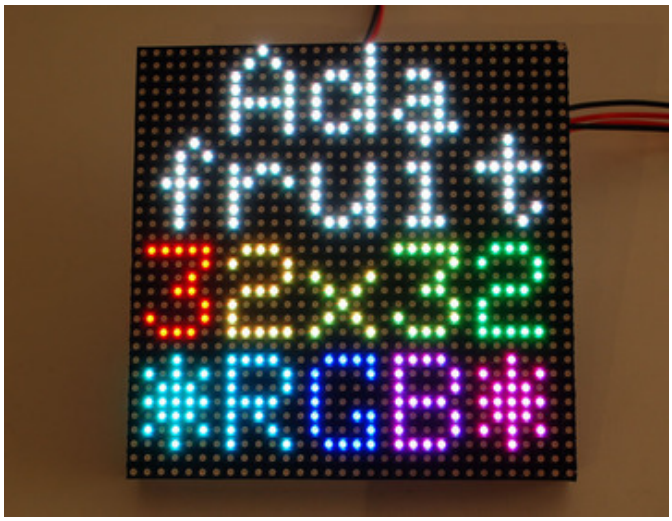
RGB Matrix Displays

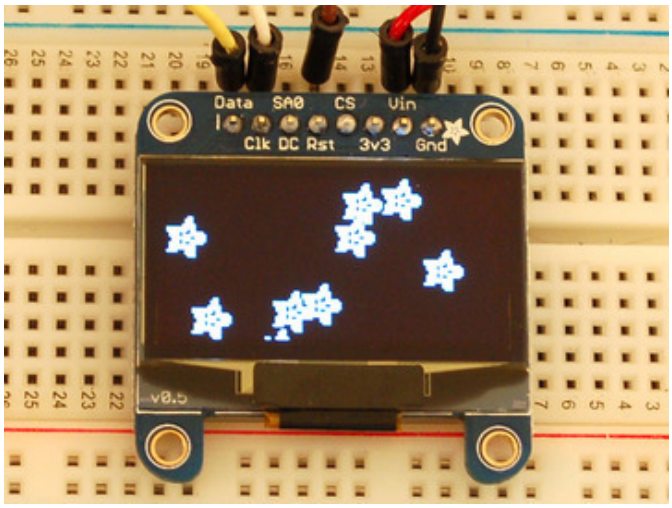
Like the pixels, these matrix displays require several bytes of SRAM per pixel.

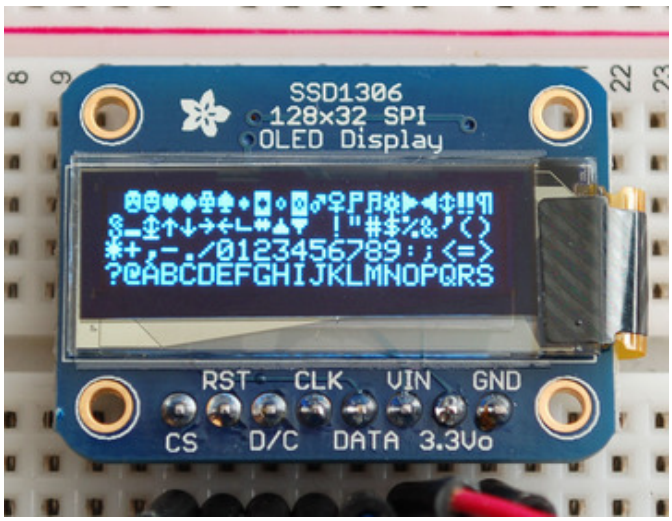


The 32x32 model requires about 1600 bytes of SRAM. The 16x32 needs around 800 bytes.





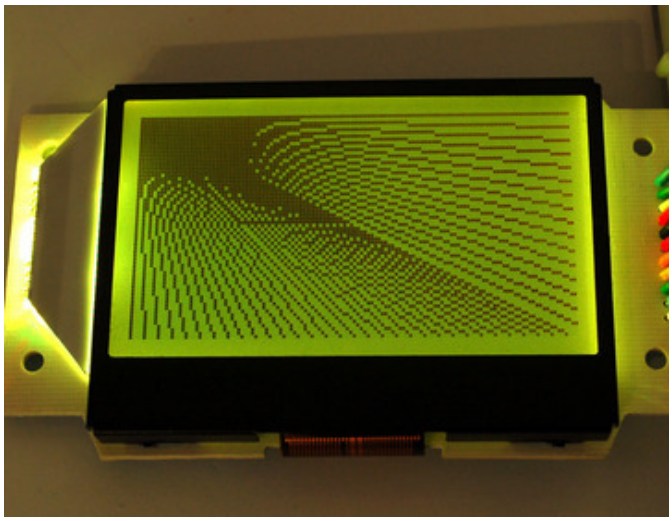


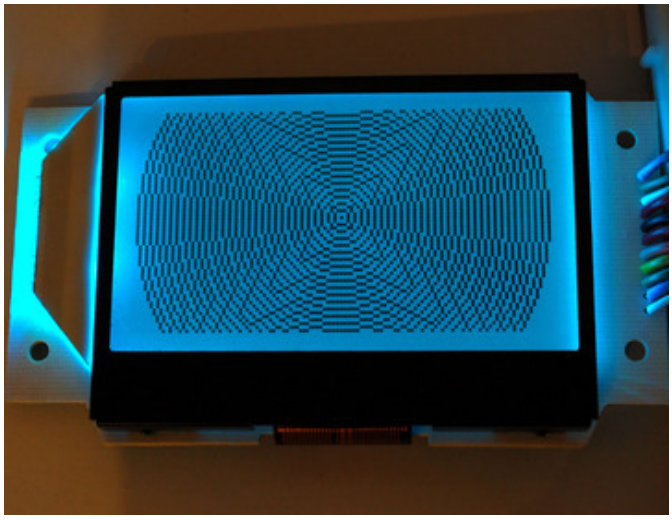


Monochrome OLED Displays

These only require 1 byte for every 8 pixels, but due to their high resolution, there are still a lot of pixels!

The 128x64 version requires 1K of SRAM
The 128x32 uses 512 bytes.





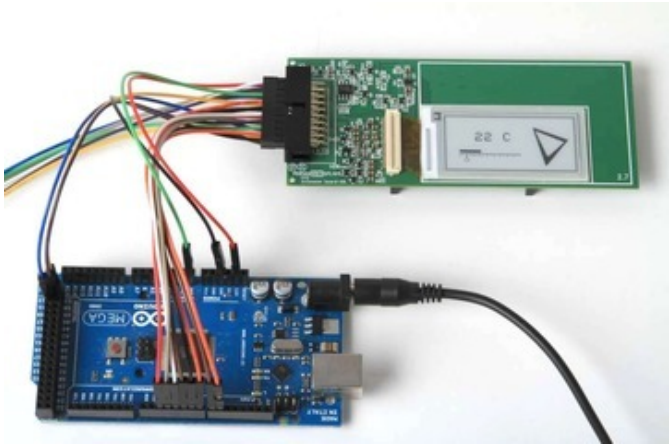
ST7565 LCD Displays

Like the monochrome OLEDs, they only need 1 byte for every 8 pixels, but they have a lot of pixels, so they require a 1K buffer.

e-Ink Displays

These high-resolution monochrome displays support some basic drawing modes that do not require in-processor buffering. But to enjoy the full capabilities of the Adafruit GFX library, a SRAM buffer is necessary.

The 2.0" version of this display requires 3K of SRAM, so GFX is only usable with a Mega.

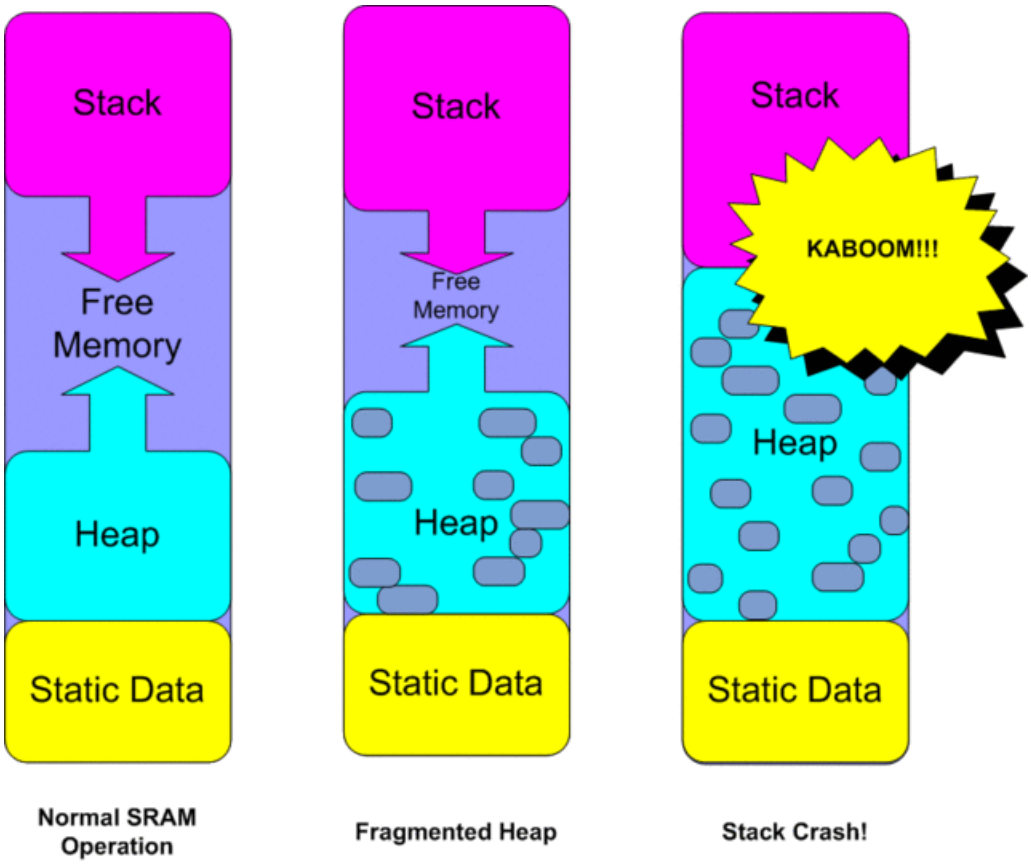


Solving Memory Problems

"Running Light Without Overbyte"

Motto of the original "Dr. Dobb's Journal of Computer Calisthenics and Orthodontia"

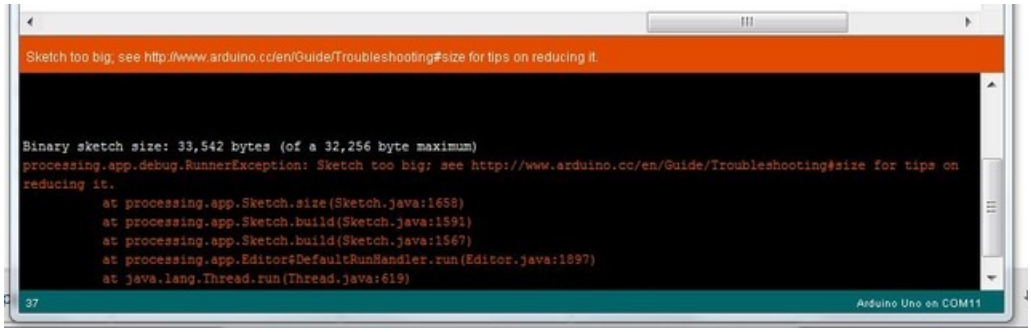
Memory is a finite resource on these tiny processors and some applications are just plain too big for an Arduino. But most code has some room for optimization. So if your program is just a little overweight, with a little diet and exercise, you can probably shed enough bytes to make it fit into that Uno again.



Optimizing Program Memory

When you compile your sketch, the IDE will tell you how big the program image is. If you have reached or exceeded the space available, some of these optimizations may help get you back under the limit.

This is not meant to be a definitive treatise on how to optimize your code - there are libraries full of books on the subject. What is presented here are just some simple tips to help harvest the low-hanging fruit.



Remove Dead Code

If your project is a mash-up of code from several sources, chances are there are parts that are not getting used and can be eliminated to save space.

- **Unused Libraries** - Are all the #include libraries actually used?
- **Unused Functions** - Are all the functions actually being called?
- **Unused Variables** - Are all the variables actually being used?
- **Unreachable Code** - Are there conditional expressions which will never be true?

Hint - *If you are not sure about an #include, a function or a variable. Comment it out. If the program still compiles, that code is not being used.*

Consolidate Repeated Code

If you have the same sequence of code statements in two or more places, consider making a function out of them.

Eliminate the Bootloader

If space is really-really tight, you might consider eliminating the bootloader. This can save as much as 2K or 4K of Flash - depending on which bootloader you are currently using.

The downside of this is that you will need to load your code using an ISP programmer instead of via a standard USB cable.

Optimizing SRAM

SRAM is the most precious memory commodity on the Arduino. Although SRAM shortages are probably the most common memory problems on the Arduino. They are also the hardest to diagnose. If your program is failing in an otherwise inexplicable fashion, the chances are good you have crashed the stack due to a SRAM shortage.

There are a number of things that you can do to reduce SRAM usage. These are just a few guidelines to get you started:

Remove Unused Variables

If you are not sure whether a variable is being used or not, comment it out. If the sketch still compiles, get rid of it!

F() Those Strings!

(Park the char* in Harvard PROGMEM)

Literal strings are repeat memory offenders. First they take up space in the program image in Flash, then they are copied to SRAM at startup as static variables. This is a horrible waste of SRAM since we will never be writing to them.

Paul Stoffregen of PJRC and Teensyduino fame developed the F() macro as a super-simple solution to this problem. The F() macro tells the compiler to keep your strings in PROGMEM. All you have to do is to enclose the literal string in the F() macro.

For example, replacing this:

```
Serial.println("Sram sram sram sram. Lovely sram! Wonderful sram! Sram sra-a-a-a-a-am sram sra-a-a-a-a-
```

with this:

```
Serial.println(F("Sram sram sram sram. Lovely sram! Wonderful sram! Sram sra-a-a-a-a-am sram sra-a-a-a-
```

Will save you 180 bytes of wonderful SRAM!

Reserve() your strings

The Arduino string library allows you to reserve buffer space for a string with the reserve() function. The idea is you can prevent String from fragmenting the heap by using reserve(num) to pre-allocate memory for a String that grows.

With the memory already allocated, String doesn't need to call realloc() if the string grows in length. In most usages, lots of other little String objects are used temporarily as you perform these operations, forcing the new string allocation to a new area of the heap and leaving a big hole where the previous one was (memory fragmentation). Usually all you need to do is use reserve() on any long-lived String objects that you know will be increasing in length as you process text.

You can do better with C strings, but if you just follow these guidelines for String objects, they work nearly as efficiently and using them is so much easier.

Move constant data to PROGMEM.

Data items declared as PROGMEM do not get copied to SRAM at startup. They are a little less convenient to work with, but they can save significant amounts of SRAM. The basic Arduino reference for PROGMEM is

here (https://adafru.it/aMw). And there is a more detailed tutorial on the subject here (https://adafru.it/coi).

Reduce Buffer Sizes

Buffer and Array Allocations:
If you allocate a buffer, make sure it is no bigger than it needs to be.

Buffers in Libraries:
Also be aware that some libraries allocate buffers behind the scenes that may be candidates for trimming as well.

System Buffers:
Another buffer hidden deeply in the system is the 64 byte serial receive buffer. If your sketch is not receiving a lot of high-speed serial data, you can probably cut this buffer size in half - or maybe even less.

The Serial buffer size is defined in HardwareSerial.cpp. This file can be found in your Arduino install directory:

> **....\Arduino-1.x.x\hardware\arduino\cores\arduino\HardwareSerial.cpp**

Look for the line:

> **#define SERIAL_BUFFER_SIZE 64**

And change it to 32 or less.

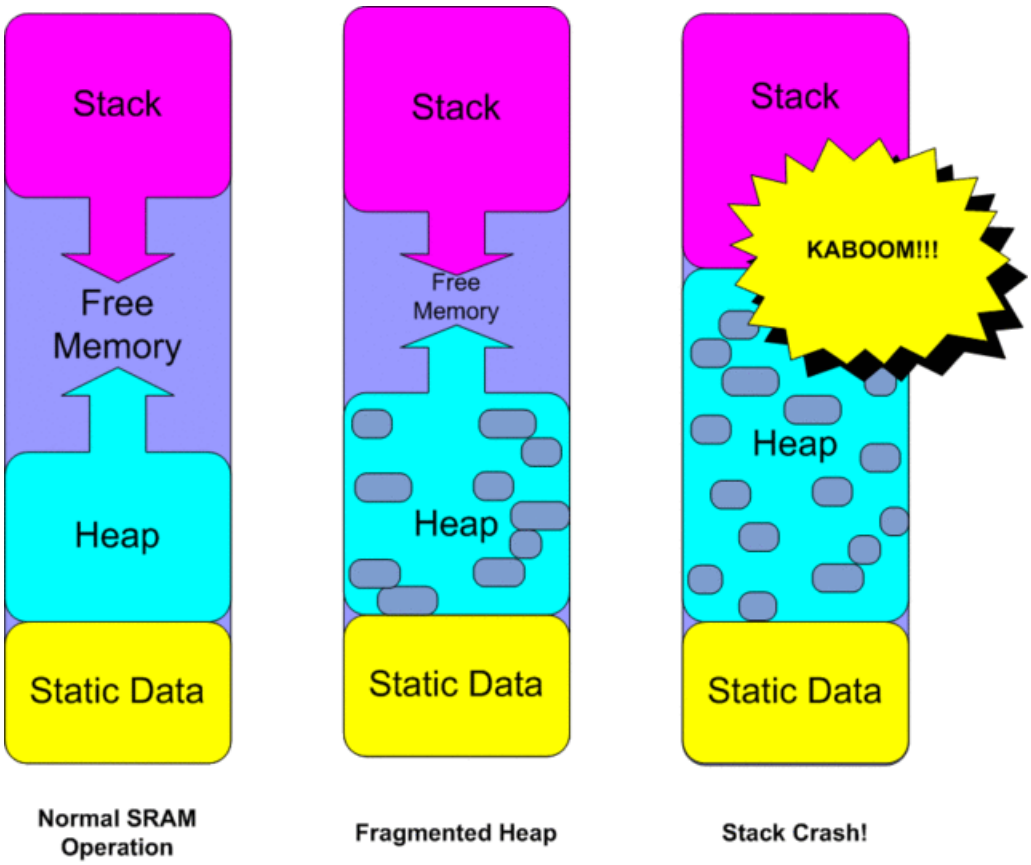
Reduce Oversized Variables

Don't use a float when an int will do. Don't use an int when a byte will do. Try to use the smallest data type capable of holding the information.

Data Types	Size in Bytes	Can contain:
boolean	1	true (1) or false (0)
char	1	ASCII character or signed value between -128 and 127
unsigned char, byte, uint8_t	1	ASCII character or unsigned value between 0 and 255
int, short	2	signed value between -32,768 and 32,767
unsigned int, word, uint16_t	2	unsigned value between 0 and 65,535
long	4	signed value between -2,147,483,648 and 2,147,483,647
unsigned long, uint32_t	4	unsigned value between 0 and 4,294,967,295
float, double	4	floating point value between -3.4028235E+38 and 3.4028235E+38 (Note that double is the same as a float on this platform.)

Think Globally. Allocate Locally.

Let's have another look at how SRAM is used (and abused):



Global & Static Variables

Global and Static variables are the first things loaded into SRAM. They push the start of the heap upward toward the stack **and they will occupy this space for all eternity.**

Dynamic Allocations

Dynamicaly allocated objects and data cause the heap to grow toward the stack. Unlike Global and Static variables, these variables can be de-allocated to free up space. **But this does not necessarily cause the heap to shrink!** If there is other dynamic data above it in the heap, the top of the heap will not move. When the heap is full of holes like swiss cheese we call it a "**fragmented heap**".

Local Variables

Every function call creates a stack frame that makes the stack grow toward the heap. Each stack frame will contain:

- All parameters passed to the function
- All local variables declared in the function.

This data is usable within the function, but **the space is 100% reclaimed when the function exits!**

The Takeaway?

- **Avoid dynamic heap allocations** - These can quickly fragment the limited heap-space.
- **Prefer local to global allocation** - Stack variables only exist while they are being used. If you have variables that only are used in a small section of your code, consider making that code into a function and declaring the

variables local to the function.

Using EEPROM

EEPROM is a handy, non-volatile storage space that works well for storing data such as calibration or tuning constants that are not practical to hard-code into Flash.

It is unusual to run out of EEPROM. And it is not often practical to use EEPROM to offload SRAM data. But we'll mention it here for completeness. Using EEPROM requires that you include the EEPROM library.

```
#include <EEPROM.h>
```

The EEPROM library gives us 2 functions:

uint8_t read(int)

Read a byte from the specified EEPROM address

void write(int, uint8_t)

Write a byte to the specified EEPROM address

Note that while reads are unlimited, there are a finite number of write cycles (typically about 100,000).