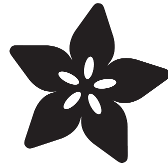
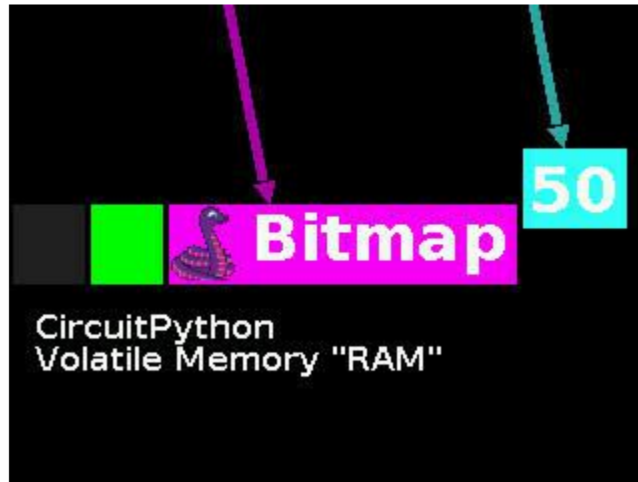


Memory-saving tips for CircuitPython

Created by Kevin Matocha



https://learn.adafruit.com/Memory-saving-tips-for-CircuitPython

Last updated on 2022-12-01 04:02:50 PM EST

Table of Contents

Overview 3

Flash storage: Space-saving tips 4

- Remove hidden files

Memory: CircuitPython basics 6

Memory: Measuring memory use 10

Memory: Optimizing memory use 11

- Biggest memory user #1: Bitmaps
- Biggest memory user #2: Fonts and Text Labels

Memory: Reducing fragmentation 13

- Memory allocation failed, but I have plenty of memory free! (memory fragmentation)

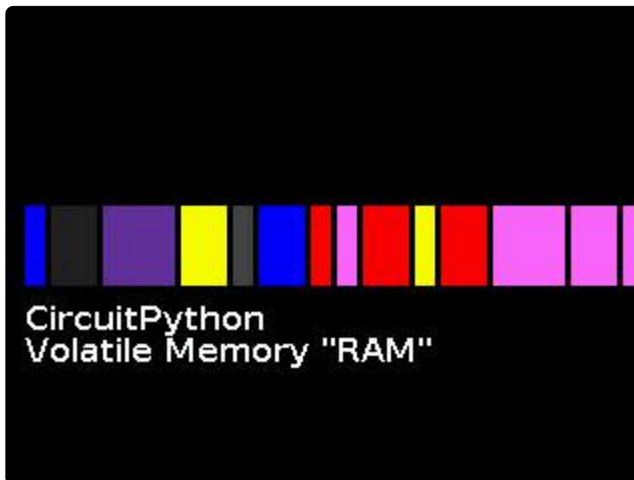
Memory: Additional memory-saving tips and resources 15

- A few more tips
- Other resources for optimizing memory in CircuitPython

Overview

This guide gives you the debug tools to identify the cause of memory-related errors and some tips and techniques to help get your CircuitPython project running again.

When creating a new project or adapting an existing one, sometimes you can run into storage or memory-related errors that stand in your way. First, it's important to understand the type of error you're getting so you will know which solutions to try.



Types of storage: Non-volatile and volatile

Items in non-volatile storage retain their values even after power is turned off, so this is used for storing files on the CIRCUITPY drive (.py files, libraries, bitmaps and font files). This non-volatile type of storage is sometimes called "Flash memory" or "EEPROM".

Volatile memory is often called "RAM" or "Random Access Memory". RAM is really fast to read and write, so it's used for storing all the active variables in your code.

Microcontrollers have differing amounts of non-volatile and volatile memory.

As your project gets more complex, you may run into situations where you reach the limit of the microcontroller's non-volatile Flash storage or its volatile memory. This guide gives you techniques to identify the root cause of memory-related errors and helps you squeeze as much as you can out of a given microcontroller board.

If your code still runs into limitations with memory-usage, you can consider finding another microcontroller board with more capability. But before switching boards, it's best to identify the root cause of what is using up your storage and memory.

Flash storage: Space-saving tips

If you encounter an "out of disk space" error when trying to save a file to your `CIRCUITPY` drive, you are running into a limitation of your non-volatile Flash storage.

If you've already deleted all unnecessary files, there are a few other techniques to save some space on your Flash storage on your board. Your library files, bitmaps and font files will likely be the largest users of your file storage space, here's some ways to address those.

Shrink your code

When writing and editing CircuitPython code, you create text files with `.py` as the file extension. These `.py` files are normal ASCII text files that can be edited with any text editor. CircuitPython interprets your code from this text and executes your commands. Additionally CircuitPython can execute "partially compiled" `.mpy` files. These `.mpy` files take up about half the storage space of the original raw text `.py` file.

Use pre-compiled Libraries
To save storage space, be sure to use the `.mpy` file for all your libraries placed in the `/lib` folder. For all libraries supported by Adafruit or in the Community bundle, they are already prepared into the `.mpy` format. You can find these pre-compiled libraries in the bundles found on the CircuitPython.org website.

If you need to make a change to any library code, download the `.py` version of the library file from the library's repository, edit it and copy the file into the `/lib` folder of your `CIRCUITPY` drive. Always be sure to delete any of the pre-compiled `.mpy` versions of the library file to prevent a conflict with your custom `.py` library version.

When to pre-compile your own code
If your project uses a lot of lines of code, you can convert your `.py` files to `.mpy` files to save on storage space.

Creating `.mpy` files takes a little more work. The benefit of CircuitPython is so you can iterate your code quickly, so it's fastest to develop your code in text using `.py` files. Only convert your code to `.mpy` when you really need to save the last bit of Flash storage space. Or break your code into logical chunks and pre-compile only the code files that are stable, so you can iterate quickly on the new parts of your code. Here are the instructions for creating your own `.mpy` files.

Note: The `code.py` file cannot be pre-compiled, but feel free to organize your code so that the `code.py` calls other files and libraries that are precompiled. Precompile your main code to `mycode.mpy`, then your `code.py` can be simplified down to:

```
import mycode
```

Bitmap files

Graphical projects should use lots of cool graphics, so you will likely use bitmap files in your project. These files often take up a large amount of storage space, so try these techniques if you want to reduce their filesizes.

Reduce color depth
Bitmap files used in CircuitPython projects are sometimes formatted as indexed bitmaps (). To shrink the indexed bitmap filesize consider reducing the "color depth", the number of colors that are used in your bitmap. Small changes to the color depth won't make an impact, but reducing the color depth by multiple factors of two or four may reduce filesize.

Replace bitmaps with generated graphics
If you have background images with grids or simple shapes (lines, rectangles, circles and polygons) consider using the `vectorio` module to generate the graphics using code. The `vectorio` module is memory-optimized, so it also can reduce your RAM use too. Here are the the graphics objects available in the `vectorio` module.

Font files

Font files contain the graphical data necessary to display text labels on your LCD and matrix displays. These font files can take up significant storage space, especially for larger font sizes. If you use font files in your project, here are a couple techniques to consider when looking to free up Flash storage space.

Multiple font sizes
If you're using multiple font files, consider reducing the number of font files and use the `scale` option that is provided in the `label` and `bitmap_label` in the Adafruit CircuitPython Display Text library. () This technique will lose some resolution for your text labels, but it will reduce the total storage required for your font files.

BDF and PCF font files
If you're using font files, shrink the font file size by eliminating un-needed character glyphs from your files. Here's a guide to shrinking font file sizes. If you're using BDF-format font files and you've deleted unnecessary characters, you're ready to save even more storage space by converting to PCF format. Converting to PCF format, saves file storage space and the fonts will load faster too. Win, win!

Remove hidden files

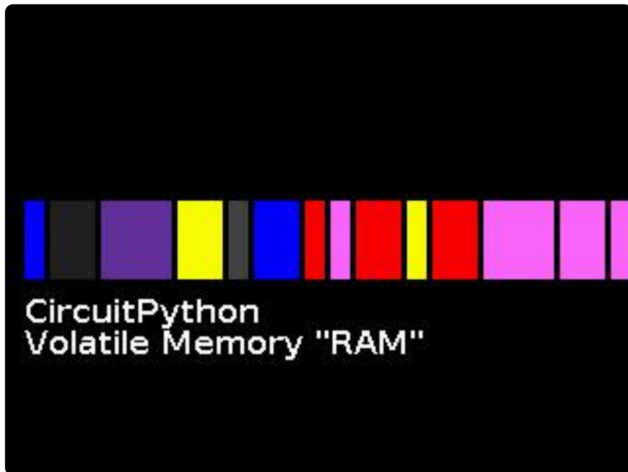
Sometimes your operating system will automatically create hidden files on your CircuitPython's file storage. Here is a guide that explains how to find and remove those hidden files that can fill up your `CIRCUITPY` drive. ()

Memory: CircuitPython basics

If you want to optimize your volatile memory usage, it is good to understand the basics of how the CircuitPython memory manager and garbage collector work. This section describes how the memory manager places newly created objects, and how and when the garbage collector cleans up old, unused objects. Once you understand these two memory workers, it is easier to understand the memory-saving tips in the next section.

To go straight to the tips, skip ahead to the next sections for a collection of RAM-saving techniques.

The volatile memory or "RAM" is used to store all the active variables that you use in your program. Whenever you use a new variable, the memory manager allocates memory space for that object.



Example: Creating a Bitmap in memory
When creating a Bitmap image of Blinka, the memory manager searches for the first available contiguous memory location that can fit the bitmap in its memory space. As shown in the animation, the memory manager scans from the beginning of the memory and looks for an available space in memory that can fit the Blinka bitmap. Once it finds a suitable-sized space, the memory manager stores the bitmap in that memory location.

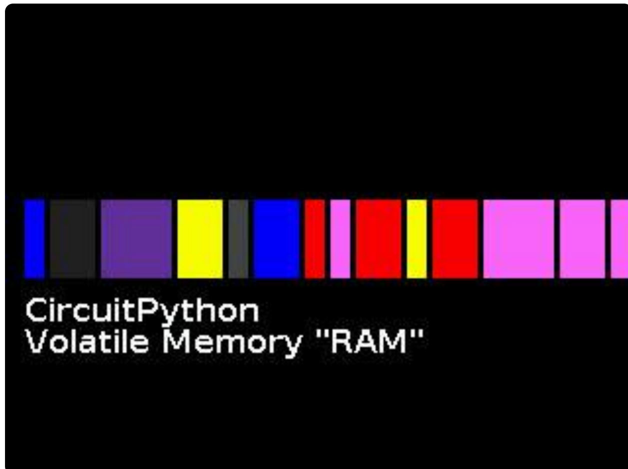
Memory "references"

When objects are created (sometimes called "instanced") they are placed in memory and are "referenced" by other objects in memory. By "referenced", this means that the object is somehow used by another object. In the case of displaying a Bitmap on a display, the bitmap is placed into a `displayio.TileGrid`, which in turn is placed into a `displayio.Group` which is further shown on the screen by `display.show()`. In this case, the Bitmap is referenced by the TileGrid, since the Bitmap is one of the inputs when the TileGrid is created. These are all examples of references.

This concept of "reference" is important since it highlights when memory objects are still needed. Whenever an object is still "referenced" by other objects, the object should be kept around. However, whenever nothing references a given memory object, then that object is no longer needed.

Once a memory object is no longer referenced, it remains in memory as a “phantom” memory object. That memory space remains unavailable until it is identified as free by the garbage collector.

Example: "phantom" memory objects
Blinka starts playing a game using a Blinka bitmap icon (of course!), so the code creates a memory object with the Blinka graphic bitmap. Blinka finishes with a high score of 50, and that score is stored in memory just after the bitmap.



After the game is over the graphics are no longer needed, so the bitmap is dereferenced (it turns to gray "phantom" in the animation). Now the game counts amount of time it was played (10 minutes, 55 seconds) and goes to store that value in memory.

The memory manager doesn't recognize the dereferenced Blinka bitmap as free space.

It goes straight past the "phantom" dereferenced bitmap and places the time object just after the score value.

There is a solution: We need to call the garbage collector.

Two workers: Memory manager and Garbage collector

The memory manager doesn’t “know” which objects are still referenced and which ones are phantom objects, so it will only place objects in free spaces in memory. We rely on the garbage collector to identify and clean out phantom memory objects. When called into action, the garbage collector looks through all the memory objects and determines which items are no longer used. Then it labels these memory spaces as free to make them usable by the memory manager.

The memory manager and garbage collector have separate jobs, but they work together manage the memory space for your variables.

Calling the garbage collector

As mentioned above, the memory manager only knows about open memory spaces and creates variables there. But remember, if the memory manager finds enough

available space anywhere, it will place it there and move on. However, if the memory manager cannot find space for a new object, it immediately calls in the garbage collector to clean out any unused memory items. Then the memory manager will search one more time for enough memory space to store the new object.

The garbage collector is always ready and waiting to go to work, but it only cleans out the memory space when it receives an request from the memory manager, or from you!

In your CircuitPython code, you can request that the garbage collector clean out all the unused variables in volatile memory space by using the `gc.collect()` command. There is a built-in module called `gc` that you import and then you can use the `gc.collect()` command as below:

```
import gc

# some lines of code here...

gc.collect()
```

Example: Clean out the "phantom" Blinka bitmap

In the previous example, the references to the Blinka bitmap were removed, yet the memory manager looked past it and placed the time variable after the "50" score variable.

Immediately after de-referencing the Blinka bitmap is a great time to call the garbage collector. After the Blinka's game is over, we immediately call `gc.collect()`. The garbage collector arrives to mark the unreferenced Blinka bitmap memory space as free and the memory manager can now place the time variable there.

So, after large memory items get dereferenced, it's a great time to call in the garbage collector to free out those unused memory spaces.

After large memory items get dereferenced or after calling a function and the function returns, it's a great time to call in the garbage collector to free out unused memory spaces. So call `gc.collect()` anytime you are finished using large memory objects. If you are done with a variable you can call `del large_variable`. If nothing else is referencing that variable, a call to `gc.collect()` will free up that memory space for use in your code.

Memory: Measuring memory use

When your project gets more complex, you can run into an error message of “Memory allocation failed”. This indicates that the memory manager (even after garbage collection) did not find a large enough continuous space in volatile memory to store a newly created variable. Eventually your project may become sufficiently complex that it will require additional RAM by upgrading to a different microcontroller board. However before upgrading, there are numerous techniques worth evaluating that can reduce your memory footprint and provide larger open memory spaces for your project.

Measure measurement tools

To find a good solution to reduce memory use, it’s best to identify what parts of your code use the most memory. Break your code into sections and use the combination of `gc.collect()` and printing `gc.mem_free()` to identify how much memory is used in each section. Pay particular attention to any graphics, font files or text labels since they tend to be heavy users of memory.

Use gc.collect() and gc.mem_free(): The `gc` library also contains a function called `gc.mem_free()` that returns the value of free memory bytes. The `gc.mem_free()` command returns the value of free memory that the memory manager sees. The command `gc.mem_free()` thinks that “phantom” variables are using space, so it only counts totally-free space.

Important: The command `gc.mem_free()` only counts free bytes of RAM. Always run `gc.collect()` first so you clean out any garbage “phantom” items before calling `gc.mem_free()`. That way you know exactly how much memory is really available for your projects.

When you want to check how much RAM your code is using, bracket your code with these `gc` library commands to measure your available memory and then calculate the

amount of bytes that were used by a code section. You can also measure how much memory is used by your imports the same way.

Here is a code snippet to quantify how many bytes of RAM are used by a code section.

```
import gc
# add other imports

gc.collect()
start_mem = gc.mem_free()
print( "Point 1 Available memory: {} bytes".format(start_mem) )

# add code here to be measured for memory use

gc.collect()
end_mem = gc.mem_free()

print( "Point 2 Available memory: {} bytes".format(end_mem) )
print( "Code section 1-2 used {} bytes".format(start_mem - end_mem) )
```

Measuring your code's memory usage is the first step to decide where to focus your improvements. The next section describes specific situations and techniques to help you get your projects running when you encounter memory limits on your board.

Memory: Optimizing memory use

Minimize imports

Libraries are necessary for almost all projects and will require some memory. As described in the previous section on optimizing file storage, be sure your library files are in `.mpy` format. That saves both file storage space and some RAM too.

When importing libraries, only import the functions that you need. You can selectively import functions using `from ... import` as shown below.

```
from adafruit_display_text.bitmap_label import Label
```

This code imports the `Label` function from the `adafruit_display_text.bitmap_label` library. The code can now call this function by using the `Label()` command.

By selectively importing only the functions you need, you may be able to reduce your memory usage versus importing a whole library. Each CircuitPython library is organized differently, so the impact of this technique will vary. Measure the memory use of the import to see how much memory is saved by using selective imports.

Biggest memory user #1: Bitmaps

Use OnDiskBitmap

If you are using `Adafruit_Imageload` for displaying bitmaps, that may be a large user of your precious RAM. Regarding bitmaps, one memory-saving alternative to `Adafruit_Imageload` is to display directly from the stored file using the OnDiskBitmap functions ()

Using OnDiskBitmap does not store the bitmap in RAM, it just draws it directly from the stored file location (could be the `CIRCUITPY` drive or an SD memory card). The downside is that the display will not update as fast when using `OnDiskBitmap` since it has to be loaded from the non-volatile memory which is often slower, and `OnDiskBitmap` does not take advantage of displayio's "dirty rectangle" tracking that reduces redraw times.

Reduce color depth

If you need the fast display redrawing that you get with `Adafruit_Imageload` of bitmaps, you can consider simplifying the color depth of your bitmaps. The bitmaps that CircuitPython can use are so-called "indexed" bitmaps. They contain a palette of colors used in the bitmap, and then each pixel on the bitmap has an entry in the file that tells it which palette color should be used. If you can reduce the number of colors then it may reduce memory usage significantly. But keep in mind due to the binary nature of how bitmap colors are stored, there are breakpoints in the number of colors that will have an impact on memory usage. For example, if you have 32 colors, going to 31 colors won't save anything, but reducing to 16 colors can reduce the amount of RAM that the bitmap uses. You need an external image editor to change the amount of colors, you can find software tools to do this. () The main jumps occur between 2, 4, 8, 16, 32, 64, 128, and 256 (they're binary factors of 2). Try reducing your color depth by a factor of two or four and evaluate whether that is a good solution for your project.

Use the vectorio module

If your bitmaps are relatively simple shapes and lines, replace them with the `vectorio` module functions. You can draw lines, circles and polygons and it doesn't use much RAM at all. Learn more about vectorio in the docs. ()

Biggest memory user #2: Fonts and Text Labels

Fonts can also take up a lot of RAM since they are basically a collection of little bitmaps for each character glyph. If you need to display different font sizes, consider loading one smaller font file and then use the `scale` parameter in the `label` or `bitmap_label` from the display_text library.

If you're going to use text labels with displayio, start with `bitmap_label`, it uses less RAM than `label`. (There are some counterintuitive situations where you may want to use `label` even though it uses more total RAM, we'll discuss that situation later.)

Here's a quick summary of these bitmap and font tips:

- Load bitmaps directly from non-volatile memory using `OnDiskBitmap`
- Reduce the color depth of bitmaps
- use `bitmap_label` for creating text labels (from library `adafruit_display_text`)
- Use `vectorio` for graphics whenever possible instead of bitmaps.
- Load one small font and scale it as needed (see `scale` parameter for labels)

Memory: Reducing fragmentation

Memory allocation failed, but I have plenty of memory free! (memory fragmentation)

Sometimes, you can see with `gc.mem_free()` that you have plenty of memory available, but you still get a message "Memory allocation failed". As described in a previous section, the memory manager and garbage collector work to clean up and identify available memory space. After running your program for a bit, your memory may become "fragmented", meaning that there are only small contiguous areas of open memory space. While the total memory available may be sufficient for your new object, if there is no continuous memory space available the error is raised "Memory allocation failed". (Note: Doing a "defragmentation" of memory is not feasible with the memory structure of CircuitPython, but there are things you can do to help reduce memory fragmentation.) Here are several strategies to reduce memory fragmentation to help retain space for allocating larger memory objects.

Call the garbage collector early and often

The garbage collector is triggered when the memory manager cannot find an free memory space to fit a new variable. But you can call the garbage collector into action in your own code to clear out unused variables and reduce memory fragmentation.

Call `gc.collect()` periodically in your code to free up the memory space from unused memory objects, especially in code locations immediately after memory items are dereferenced. If you are done with a variable, and use `del large_variable`, call the garbage collector immediately.

After function returns

Using functions is a good way to "encapsulate" and free up your memory usage. After a function call returns from execution, all the local variables are immediately dereferenced and become "phantom" objects. After a function call that creates a lot of large variables, call `gc.collect()` immediately to free up all the "phantom" memory objects that were dereferenced when the function returned. Each call to `gc.collect()` will take some processor time but it can significantly reduce your memory fragmentation.

If you are reaching the limits of your memory, sprinkle in `gc.collect()` in your code to help reduce memory fragmentation. After a function returns is always a great time to call `gc.collect()` to free up space.

Before creating large memory objects

Another good time to call `gc.collect()` is immediately prior to creating large objects. There could be "unused" phantom variables clogging up a space that could accommodate your new big variable. By running the garbage collector, you clear out those unused variables so the new memory allocation can be placed in that space.

Other techniques to reduce memory fragmentation:

- Use functions where it makes sense for memory items that are temporary. Take advantage of the fact that after a function closes, all its local variables are automatically "unused". Call `gc.collect()` after function returns to reclaim all that memory space as free again.
- Use `gc.collect()` prior to any large memory allocations. This will help reduce the fragmentation since you clear out all phantom unused variables before you request a new chunk.

- For bitmaps and text labels that are static through your program, create them early in the life cycle ofyour code. If you allocate them toward the end, it is a higher chance that your memory will get fragmented and cause a memory allocation error when you later create the bitmap or text label. Allocate these large items early while memory space is relatively wide open.
- Changing text in a `label` causes reallocation of the `label`'s memory. Consider ways to avoid changing the text in your `label`.
- Special note with text labels: `bitmap_label.label` uses less overall RAM but it needs all its memory in one chunk. Somewhat counterintuitively, `label.label` uses more total RAM but splits it up into smaller pieces, so consider using label if you need to allocate text labels with frequently changing text. This is one way to "walk between the raindrops" when your memory is fragmented. If you really need to redo the text in a label frequently, evaluate whether `label.label` can help.
- Advanced programmers: Allocate a large memory buffer early in the life of your code and reuse the same memory buffer through your program.

Memory: Additional memory-saving tips and resources

A few more tips

- Use list generators when possible rather than creating lists (for example use `range()` to create a sequential list of values on the fly without assigning it to a variable).
- "pystack exhausted" errors - you may be doing something with a lot of of levels of function calls or recursion. Maybe try another way of solving your problem. Other times the cause is unclear, ask for guidance on the `help-with-CircuitPython` channel on discord: https://adafru.it/discord ()

Other resources for optimizing memory in CircuitPython

- FAQ: See "Memory" in the CircuitPython Frequently Asked Questions ()
- FAQ: Deeper dive via microPython and very relevant to CircuitPython ()
- Video: Project specific - @foamyguy's livestream on saving memory on the EZbake oven for the big-screened PyPortal Titano ()

- Video: Deep dive with Scott Shawcroft - deep into the weeds of the CircuitPython memory structure ()