← Go Back

Learn

Arduino Ecosystem
Microcontrollers
Programming
Electronics
Communication
Hardware Design
Built-In Libraries
Contributions

Arduino Style Guide for Writing Content
Arduino Style Guide for Creating Libraries
Writing a Library for Arduino
Writing Content for Arduino

Home / Learn / Arduino Style Guide for Creating Libraries

Arduino Style Guide for Creating Libraries

Learn how to write library APIs in an Arduino style.

Last revision • 04/07/2023

This is a style guide to writing library APIs in an Arduino style. Some of these run counter to professional programming practice. We're aware of that, but it's what's made it possible for so many beginners to get started with Arduino easily. So please code with these principles in mind. If you have suggestions on how to make Arduino libraries clearer for that core audience, please jump in the discussion.

Be kind to the end user. Assume you are writing an API for an intelligent person who has not programmed before. Come up with a clear mental model of the concept you're working with, and the terms and functions you will use.

Match your API to the underlying capabilities. You don't want to expose implementation details to the user but you also don't want an API that suggests an inaccurate mental model of the possibilities. For example, if there are only a few possible options for a particular setting, don't use a function that takes an int, as it implies you can use any value you want.

Organize your public functions around the data and functionality that the user wants. Quite often, the command set for a particular electronic module is overly complicated for the most common uses, or can be re-organized around higher level functionality. Think about what the average person thinks the thing does, and try to organise your API functions around that. Adafruit's BMP085 library is a good example. The `readPressure()` function performs all the necessary steps to get the final pressure. The library wraps this commonly executed series of functions into a high-level single command which returns the value the user's looking for in a format she expects. It abstracts away not only the low-level I2C commands, but also the mid-level temperature and pressure calculations, while still offering those mid-level functions as public functions for those who want them.

Use full, everyday words. Don't be terse with your function names or variables. Use everyday terms instead of technical ones. Pick terms that correspond to popular perception of the concept at hand. Don't assume specialized knowledge. For example, this is why we used `analogWrite()` rather than `pwm()`. Abbreviations are acceptable, though, if they're in common use or are the primary name for something. For example, "HTML" is relatively common and "SPI" is effectively the name of that protocol ("serial-peripheral interface" is probably too long). ("Wire" was probably a mistake, as the protocol it uses is typically called "TWI" or "I2C".)

Avoid words that have different meanings to the general public. For example, to programmers, an error is a notification that something happened. To the general public, errors are bad things.

describing it to the general public FIRST. You'll likely come across a better term, and if not, you'll have started the documentation on your library.
Document and comment as you go. When writing examples and documentation, follow the Writing Style Guide

Use the established core libraries and styles.

- Use `read()` to read inputs, and `write()` to write to outputs, e.g. `digitalRead()`, `analogWrite()`, etc.
- Use the `Stream` and `Print` classes when dealing with byte streams. If it's not appropriate, at least try to use its API as a model. For more on this, see below
- For network applications, use the `Client` and `Server` classes as the basis.
- Use `begin()` to initialize a library instance, usually with some setting s. Use `end()` to stop it.
- Use camel case function names, not underscore. For example, **analogR ead**, not **analog_read**. Or **myNewFunction**, not **my_new_function**. We 've adopted this from Processing.org for readability's sake.

LONG_CONSTANT_NAMES_FULL_OF_CAPS are hard to read. Try to simplify when possible, without being terse.

Try to avoid boolean arguments. Instead, consider providing two different functions with names the describe the differences between them.

Don't assume knowledge of pointers. Beginning users of C find this the biggest roadblock, and get very confused by `&` and `*`, so whenever you can avoid having them hanging out in the API, do so. One way is to pass by reference using array notation rather than `*` notation, for example.

COPY

```
void printArray(char* array);
```

can be replaced by

COPY

```
void printArray(char[] array);
```

Though there are some libraries where we pass pointers by using structures like const chars, avoid anything that requires the user to pass them. For example,rather than:

COPY

```
foo.readAccel(&x, &y, &z);
```

use something like this:

COPY

```
xAxis = adxl.readX();
  yAxis = adxl.readY();
  zAxis = adxl.readZ();
```

When using serial communication, allow the user to specify any `Stream`

compatible with all serial ports on boards with multiple (e.g., Mega), and can also use alternate interfaces like SoftwareSerial. The Stream object can be passed to your library's constructor or to a `begin()` function (as a reference, not a pointer). See **Firmata 2.3** or **XBee 0.4** for examples of each approach.

When writing a library that provides byte-stream communication, inherit Arduino's `Stream` class, so your library can be used with all other libraries that accept `Stream` objects. If possible, buffer incoming data, so that `read()` immediately accesses the buffer but does not wait for more data to arrive. If possible, your `write()` method should store data to a transmit buffer, but `write()` must wait if the buffer does not have enough space to immediately store all outgoing data. The `yield()` function should be called while waiting.

Here are a few libraries that are exemplary from Adafruit. She breaks the functions of the devices down into their high-level activities really well.

- **https://github.com/adafruit/Adafruit-BMP085-Library**
- **https://github.com/adafruit/DHT-sensor-library**

This does a nice job of abstracting from the Wire (I2C) library: **https://github.com/adafruit/RTClib**