← 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 / Writing a Library for Arduino

Writing a Library for Arduino

Creating libraries to extend the functionality of Arduino. Goes step-by-step through the process of making a library from a sketch.

Last revision • 04/24/2023

This document explains how to create a library for Arduino. It starts with a sketch for flashing Morse code and explains how to convert its functions into a library. This allows other people to easily use the code that you've written and to easily update it as you improve the library.

For more information, see the **API Style Guide** for information on making a good Arduino-style API for your library.

We start with a sketch that does simple Morse code:

COPY

```
int pin = 13;

void setup()
{
  pinMode(pin, OUTPUT);
}

void loop()
{
  dot(); dot(); dot();
  dash(); dash(); dash();
  dot(); dot(); dot();
  delay(3000);
}

void dot()
{
  digitalWrite(pin, HIGH);
  delay(250);
  digitalWrite(pin, LOW);
  delay(250);
}

void dash()
{
  digitalWrite(pin, HIGH);
  delay(1000);
  digitalWrite(pin, LOW);
  delay(250);
}
```

If you run this sketch, it will flash out the code for SOS (a distress call) on pin 13.

The sketch has a few different parts that we'll need to bring into our library. First, of course, we have the `dot()` and `dash()` functions that do the actual blinking. Second, there's the `pin` variable which the functions use to determine which pin to use. Finally, there's the call to `pinMode()` that initializes the pin as an output.

Help

You need at least two files for a library: a header file (w/ the extension .h) and the source file (w/ extension .cpp). The header file has definitions for the library: basically a listing of everything that's inside; while the source file has the actual code. We'll call our library "Morse", so our header file will be `Morse.h`. Let's take a look at what goes in it. It might seem a bit strange at first, but it will make more sense once you see the source file that goes with it.

The core of the header file consists of a line for each function in the library, wrapped up in a class along with any variables you need:

COPY

```
class Morse
{
  public:
    Morse(int pin);
    void begin();
    void dot();
    void dash();
  private:
    int _pin;
};
```

A class is simply a collection of functions and variables that are all kept together in one place. These functions and variables can be public, meaning that they can be accessed by people using your library, or private, meaning they can only be accessed from within the class itself. Each class has a special function known as a constructor, which is used to create an instance of the class. The constructor has the same name as the class, and no return type.

You need a couple of other things in the header file. One is an #include statement that gives you access to the standard types and constants of the Arduino language (this is automatically added to normal sketches, but not to libraries). It looks like this (and goes above the class definition given previously):

COPY

```
#include "Arduino.h"
```

Finally, it's common to wrap the whole header file up in a weird looking construct:

COPY

```
#ifndef Morse_h
#define Morse_h

// the #include statement and code go here...

#endif
```

Basically, this prevents problems if someone accidentally `#include`'s your library twice.

Finally, you usually put a comment at the top of the library with its name, a

Let's take a look at the complete header file:

COPY

```
/*
  Morse.h - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Released into the public domain.
*/
#ifndef Morse_h
#define Morse_h

#include "Arduino.h"

class Morse
{
  public:
    Morse(int pin);
    void begin();
    void dot();
    void dash();
  private:
    int _pin;
};

#endif
```

Now let's go through the various parts of the source file, Morse.cpp.

First comes a couple of #include statements. These give the rest of the code access to the standard Arduino functions, and to the definitions in your header file:

COPY

```
#include "Arduino.h"
#include "Morse.h"
```

Then comes the constructor. Again, this explains what should happen when someone creates an instance of your class. In this case, the user specifies which pin they would like to use. The constructor records that in a private variable for use in the other functions:

COPY

```
Morse::Morse(int pin)
{
  _pin = pin;
}
```

There are a couple of strange things in this code. First is the **Morse::** before the name of the function. This says that the function is part of the **Morse** class. You'll see this again in the other functions in the class. The second unusual thing is the underscore in the name of our private variable, `_pin`. This variable can actually have any name you want, as long as it matches the definition in the header file. Adding an underscore to the start of the name is a common convention to make it clear which variables are private, and also to distinguish the name from that of the argument to the function (**pin** in this case).

This will be called from the `setup()` function of the sketch. Hardware configuration is done in a dedicated function instead of the constructor because the hardware has not yet been initialized at the time the constructor code is executed. In our library, we need to set the pin as an output:

COPY

```
void Morse::begin()
{
  pinMode(_pin, OUTPUT);
}
```

Then comes the actual code from the sketch that you're turning into a library (finally!). It looks pretty much the same, except with **Morse::** in front of the names of the functions, and `_pin` instead of **pin**:

COPY

```
void Morse::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash()
{
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
}
```

Finally, it's typical to include the comment header at the top of the source file as well. Let's see the whole thing:

COPY

```
/*
  Morse.cpp - Library for flashing Morse code.
  Created by David A. Mellis, November 2, 2007.
  Updated by Jason A. Cox, February 18, 2023.
  Released into the public domain.
*/

#include "Arduino.h"
#include "Morse.h"

Morse::Morse(int pin)
{
  _pin = pin;
}

void Morse::begin()
{
  pinMode(_pin, OUTPUT);
}

void Morse::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash()
```

And that's all you need (there's some other nice optional stuff, but we'll talk about that later). Let's see how you use the library.

First, make a **Morse** directory inside of the **libraries** sub-directory of your sketchbook directory. Copy or move the Morse.h and Morse.cpp files into that directory. Now launch the Arduino environment. If you open the **Sketch > Import Library** menu, you should see Morse inside. The library will be compiled with sketches that use it. If the library doesn't seem to build, make sure that the files really end in .cpp and .h (with no extra .ino, .pde or .txt extension, for example).

Let's see how we can replicate our old SOS sketch using the new library:

COPY

```
#include <Morse.h>

Morse morse(13);

void setup()
{
  morse.begin();
}

void loop()
{
  morse.dot(); morse.dot(); morse.dot();
  morse.dash(); morse.dash(); morse.dash();
  morse.dot(); morse.dot(); morse.dot();
  delay(3000);
}
```

There are a few differences from the old sketch (besides the fact that some of the code has moved to a library).

First, we've added an #include statement to the top of the sketch. This makes the Morse library available to the sketch and includes it in the code sent to the board. That means if you no longer need a library in a sketch, you should delete the #include statement to save space.

Second, we now create an instance of the Morse class called **morse**:

COPY

```
Morse morse(13);
```

When this line gets executed (which actually happens even before the `setup()` function), the constructor for the `Morse` class will be called, and passed the argument you've given here (in this case, just 13).

Notice that our `setup()` now has a call to `morse.begin()` which configures the pin that was set in the constructor.

Finally, to call the `dot()` and `dash()` functions, we need to prefix them with **morse**. - the name of the instance we want to use. We could have multiple instances of the `Morse` class, each on their own pin stored in the _pin private variable of that instance. By calling a function on a particular instance, we specify which instance's variables should be used during that call to a function. That is, if we had both:

COPY

```
Morse morse(13);
Morse morse2(12);
```

then inside a call to `morse2.dot()`, `_pin` would be 12.

If you tried the new sketch, you probably noticed that nothing from our library was recognized by the environment and highlighted in color. Unfortunately, the Arduino software can't automatically figure out what you've define in your library (though it would be a nice feature to have), so you have to give it a little help. To do this, create a file called **keywords.txt** in the Morse directory. It should look like this:

COPY

```
Morse   KEYWORD1
begin   KEYWORD2
dash    KEYWORD2
dot KEYWORD2
```

Each line has the name of the keyword, followed by a tab (not spaces), followed by the kind of keyword. Classes should be KEYWORD1 and are colored orange; functions should be KEYWORD2 and will be brown. You'll have to restart the Arduino environment to get it to recognize the new keywords.

It's also nice to provide people with an example sketch that uses your library. To do this, create an **examples** directory inside the **Morse** directory. Then, move or copy the directory containing the sketch (let's call it **SOS**) we wrote

> Show Sketch Folder command.) If you restart the Arduino environment (this is the last time, I promise) - you'll see a **Library-Morse** item inside the