# <https://github.com/arduino/Arduino/wiki/Library-Manager-FAQ>

# Library Manager FAQ

per1234 edited this page 10 days ago · [22 revisions](https://github.com/arduino/Arduino/wiki/Library-Manager-FAQ/_history)

**Library Manager overview**

The Arduino Library Manager is a feature of the Arduino IDE (Sketch > Include Library > Manage Libraries...) which makes it easy for users to find, install, and update both official and 3rd party libraries. When your library is added to the library list every release/tag version of the library in your repository will automatically be made available for installation via Library Manager. The users can set their preferences to display an update notification when a new version of any installed library on the list is available and easily update to the new version with just a couple clicks. More information:   
<https://www.arduino.cc/en/Guide/Libraries#toc3>

**How is the library list generated?**

From a list of public Git repos, a job (a small program that runs regularly) fetches every tag, verifies library files and pushes the updated [list](http://downloads.arduino.cc/libraries/library_index.json) onto the Arduino download server. Only valid libraries and their tags are published. A library is **not valid** when:

* it's not in [1.5 format](https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification) and in particular it misses a [library.properties](https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification#library-metadata) file (1.5 format folder layout is not required)
* its version number is not [semver compliant](http://semver.org/)
* it contains .exe files
* it contains a .development file
* it contains symlinks
* our antivirus finds infected files

**The job runs every hour. If a new library has been released, you can expect it to be listed within the hour**.

**How can I add my library to Library Manager?**

* Your library repository must be hosted on a major git-hosting website like GitHub, BitBucket or GitLab (other hosting site may be considered on request).
* Ensure your library is compliant with [1.5 format](https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification) (1.5 format folder layout is not required)
* [Tag](https://git-scm.com/book/en/v2/Git-Basics-Tagging) it and push the tag (or create a release if you web hosting offers a way to do it, for example with [GitHub "releases"](https://help.github.com/articles/creating-releases/))
* Open an issue on [Arduino's GitHub](https://github.com/arduino/Arduino/issues), specifying the URL of the repository from where to download your library
* After some days, a member of the Arduino team will add your library to the Library Manager index and close the issue. Shortly after that your library will be available for installation via Library Manager.

**How can I publish a new release once my library is in the list?**

Ensure you've changed version in your library.properties. Then tag your library once more and push the new tag (or create a release if your web hosting offers a way to do it, for example with GitHub "releases"). Our indexer checks for new releases every hour and will eventually fetch and publish your new release.

**Sorry, I did something wrong! How can I change or unpublish an already published library?**

In order to change contents of an already published library version, you can recreate its related tag.

In order to **un**publish a library version, delete its related tag/release.

Modified or Deleted library versions are processed every Sunday.

**How can I delete a library?**

Open your sketchbook "libraries" folder with your OS file explorer (Windows: Explorer, Mac: Finder, Linux: Nautilus, kfiles...) and delete the folder containing your library.

**Rationale**

No, we don't have a delete button. Libraries managed by the Library Manager are mixed with those you've manually installed, maybe libraries you've written: making a mistake and deleting the wrong library is too easy. That's why trash bins exist. Since the IDE has no knowledge of your trash bin, we didn't implement a "delete" button.

**Limitations**

**Is my Git repo OK?**

Your repo is OK if its root folder contains the file library.properties and the rest of the library source code. If your library is stored in a subfolder, your repo is not OK and we can't include your library in the list.

**Are Git submodules supported?**

No. The library archive distributed by Library Manager will only contain an empty folder where the submodule should be.

**Can I add my own URL with my own library list?**

No. At the moment, the IDE handles one URL only, and that's written into the code (dev jargon: it's hardcoded), this is a known limitation. However, if you know your way through the code, you can change that URL with another one.

**When I install a library that I know depends on another library, will this other library be installed as well?**

No, at the moment library dependencies are ignored. Current Library Manager aims at simplifying your life when dealing with installing a single library. If a library depends on another one, you'll have to install both.

**Can I install multiple versions of one library and use the proper one in my sketches?**

Library Manager installs libraries into your sketchbook "libraries" folder. Since you cannot create two folders with the same name, we can't install two versions of the same library. However, you can switch between library versions, by selecting the appropriate one from the version dropdown that pops up on Library Manager when more than one version is available.

# <https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification>

# Arduino IDE 1.5: Library specification

per1234 edited this page on Mar 21 · [60 revisions](https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification/_history)

This is the specification for the 3rd party library format to be used with Arduino IDE 1.5.x onwards.

* rev.1 has been implemented starting with IDE version 1.5.3 (now superseded by rev.2)
* rev.2 will be implemented starting from version 1.5.6
* rev.2.1 will be implemented starting from version 1.6.10

This new library format is intended to be used in tandem with the Arduino IDE"s **Library Manager**, available since version 1.6.2. The Library Manager allows users to automatically download and install libraries needed in their projects, with an easy to use graphic interface. It doesn't yet take care of dependencies between libraries. More information about how Library Manager works are available [here](https://github.com/arduino/Arduino/wiki/Library-Manager-FAQ).

Arduino IDE 1.5.x+ supports multiple microcontroller architectures (e.g. AVR, SAM, etc), meaning that libraries may need to work on multiple architectures. The new 1.5 library format doesn’t contain special support for cross-architecture libraries, but it does provide a preprocessor based mechanism for libraries to target sections of code to specific architectures.

### See also

The Arduino library style guide is here : <http://arduino.cc/en/Reference/APIStyleGuide>

The style guide for examples is here : <http://arduino.cc/en/Reference/StyleGuide>

# 1.5 library format (rev. 2.1)

## Library metadata

The most significant addition to the format is the ability to add information about the library itself through a properties file called **library.properties**.

This file allows the Library Manager to search and install a library and its dependencies in an easy and automated way.

### library.properties file format

The library.properties file is a key=value properties list. Every field in this file is UTF-8 encoded. The available fields are:

* **name** - the name of the library. Library names must contain only basic letters (A-Z or a-z) and numbers (0-9), spaces (), underscores (\_), dots (.) and dashes (-). It cannot start or end with a space, and also it cannot start with a number.
* **version** - version of the library. Version should be [semver](http://semver.org/) compliant. 1.2.0 is correct; 1.2 is accepted; r5, 003, 1.1c are invalid
* **author** - name/nickname of the authors and their email addresses (not mandatory) separated by comma ","
* **maintainer** - name and email of the maintainer
* **sentence** - a sentence explaining the purpose of the library
* **paragraph** - a longer description of the library. The value of **sentence** always will be prepended, so you should start by writing the second sentence here
* **category** - if present, one of these:
  + Display
  + Communication
  + Signal Input/Output
  + Sensors
  + Device Control
  + Timing
  + Data Storage
  + Data Processing
  + Other
* **url** - the URL of the library project, for a person to visit. For example, the library's GitHub page
* **architectures** - a comma separated list of architectures supported by the library. If the library doesn’t contain architecture specific code use \* to match all architectures
* **dot\_a\_linkage** - **(available from IDE 1.6.0 / arduino-builder 1.0.0-beta13)** when set to true, the library will be compiled using a .a (archive) file. First, all source files are compiled into .o files as normal. Then instead of including all .o files in the linker command directly, all .o files are saved into a .a file, which is then included in the linker command. [1.5 format library folder structure](https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification#layout-of-folders-and-files) is required.
* **includes** - **(available from IDE 1.6.10)** a comma separated list of files to be added to the sketch as #include <...> lines. This property is used with the "Include library" command in the IDE. If the property is undefined all the headers files (.h) on the root source folder are included.
* **precompiled** - **(feature not yet released, will be available in arduino-builder >1.3.25)** set to true to allow the use of .a (archive) and .so (shared object) files. The .a/.so file must be located at src/{build.mcu} where {build.mcu} is the architecture name of the target the file was compiled for. Ex: cortex-m3 for the Arduino DUE. The static library should be linked as an ldflag.
* **ldflags** - **(feature not yet released, will be available in arduino-builder >1.3.25)** - the linker flags to be added. Ex: ldflags=-lm

Example:

name=WebServer

version=1.0

author=Cristian Maglie <c.maglie@example.com>, Pippo Pluto <pippo@example.com>

maintainer=Cristian Maglie <c.maglie@example.com>

sentence=A library that makes coding a Webserver a breeze.

paragraph=Supports HTTP1.1 and you can do GET and POST.

category=Communication

url=http://example.com/

architectures=avr

includes=WebServer.h

## Layout of folders and files

Each folder has a specific purpose (sources, examples, documentation, etc). Folders not covered in this specification may be added as needed to future revisions.

### Source code

For 1.5.x+-only libraries, the source code resides in the **src** folder. For example:

Servo/src/Servo.h

Servo/src/Servo.cpp

The source code found in **src** folder and all its subfolders is compiled and linked in the user’s sketch. Only the src folder is added to the include search path (both when compiling the sketch and the library). When the user imports a library into their sketch (from the "Tools > Import Library" menu), an #include statement will be added for all header (.h) files in the src/ directory (but not its subfolders). As a result, these header files form something of a de facto interface to your library; in general, the only header files in the root src/ folder should be those that you want to expose to the user's sketch and plan to maintain compatibility with in future versions of the library. Place internal header files in a subfolder of the src/ folder.

For backward compatibility with Arduino 1.0.x, the library author may opt to place source code into the root folder, instead of the folder called **src**. In this case the 1.0 library format is applied and the source code is searched from the **library root folder** and the **utility** folder, for example:

Servo/Servo.h

Servo/Servo.cpp

Servo/utility/ServoTimers.h

Servo/utility/ServoTimers.cpp

This will allow existing 1.0.x libraries to compile under 1.5.x+ as well and vice-versa. If a library only needs to run on 1.5.x+, we recommend placing all source code in the src/ folder. If a library requires recursive compilation of nested source folders, its code must be in the src/ folder (since 1.0.x doesn’t support recursive compilation, backwards compatibility wouldn’t be possible anyway).

### Library Examples

Library examples must be placed in the **examples** folder. Note that the **examples** folder must be written exactly like that (with lower case letters).

Servo/examples/...

Sketches contained inside the examples folder will be shown in the Examples menu of the IDE.

### Extra documentation

An **extras** folder can be used by the developer to put documentation or other items to be bundled with the library. Remember that files placed inside this folder will increase the size of the library, so putting a 20MB PDF in a library that weights a few kilobytes may not be such a good idea.

The content of the extras folder is totally ignored by the IDE; you are free to put anything inside such as supporting documentation, etc.

### Keywords

A list of keywords for the library may be specified in a file named keywords.txt located in the root of the library folder. When a keyword of any installed library is used in a sketch the Arduino IDE colors it.

Servo/keywords.txt

An example keywords file:

#######################################

# Syntax Coloring Map For ExampleLibrary

#######################################

#######################################

# Datatypes (KEYWORD1)

#######################################

Test KEYWORD1

#######################################

# Methods and Functions (KEYWORD2)

#######################################

doSomething KEYWORD2

#######################################

# Instances (KEYWORD2)

#######################################

#######################################

# Constants (LITERAL1)

#######################################

This keywords file would cause the Arduino IDE to highlight Test as a DataType, and doSomething as a method / function.

#### keywords.txt format

keywords.txt is formatted in four fields which are separated by a single true tab (not spaces):

KEYWORD KEYWORD\_TOKENTYPE REFERENCE\_LINK RSYNTAXTEXTAREA\_TOKENTYPE

It is permitted to leave a field empty.

##### KEYWORD\_TOKENTYPE

| **KEYWORD\_TOKENTYPE** | **Use for** | **Theme property** |
| --- | --- | --- |
| KEYWORD1 | datatypes | editor.data\_type.style |
| KEYWORD2 | functions | editor.function.style |
| KEYWORD3 | structures | editor.function.style |
| LITERAL1 | constants | editor.reserved\_word\_2.style |
| LITERAL2 | ? | editor.function.style |

##### REFERENCE\_LINK

This field specifies the [Arduino Language Reference](https://www.arduino.cc/reference/en) page to open via **Right Click > Find in Reference** or **Help > Find in Reference** when the cursor is on that keyword. Generally it does not make sense to define the REFERENCE\_LINK field for 3rd party library keywords since they are not likely to be in the Arduino Language Reference.

##### RSYNTAXTEXTAREA\_TOKENTYPE

In Arduino IDE 1.6.5 and newer this field overrides KEYWORD\_TOKENTYPE. In previous IDE versions the RSYNTAXTEXTAREA\_TOKENTYPE field is ignored and KEYWORD\_TOKENTYPE is used instead.

| **RSYNTAXTEXTAREA\_TOKENTYPE** | **Theme property** |
| --- | --- |
| RESERVED\_WORD | editor.reserved\_word.style |
| RESERVED\_WORD2 | editor.reserved\_word\_2.style |
| DATA\_TYPE | editor.data\_type.style |
| PREPROCESSOR | editor.preprocessor.style |

### Development flag file

Normally the Arduino IDE treats the contents of the library folder as read-only. This is to prevent users from accidentally modifying example sketches. During the library development process you may want to edit example sketches in place using the Arduino IDE. With Arduino IDE 1.6.6 and newer, the read-only behavior can be disabled by adding a file named .development to the root of the library folder. A [library.properties](https://github.com/arduino/Arduino/wiki/libraryproperties-file-format) file must also be present. The [Library Manager indexer](https://github.com/arduino/Arduino/wiki/Library-Manager-FAQ) will not pick up releases that contain a .development file so be sure not to push this file to your remote repository.

### A complete example

A hypothetical library named "Servo" that adheres to the specification follows:

Servo/

Servo/library.properties

Servo/keywords.txt

Servo/src/

Servo/src/Servo.h

Servo/src/Servo.cpp

Servo/src/ServoTimers.h

Servo/examples/

Servo/examples/Sweep/Sweep.ino

Servo/examples/Pot/Pot.ino

Servo/extras/

Servo/extras/Servo\_Connectors.pdf

# Working with multiple architectures

In 1.5.x+, libraries placed in the user’s sketchbook folder (in the libraries/ subfolder) will be made available for all boards, which may include multiple different processor architectures. To provide architecture-specific code or optimizations, library authors can use the ARDUINO\_ARCH\_XXX preprocessor macro (#define), where XXX is the name of the architecture (as determined by the name of the folder containing it), e.g. ARDUINO\_ARCH\_AVR will be defined when compiling for AVR-based boards. For example,

#if defined(ARDUINO\_ARCH\_AVR)

// AVR-specific code

#elif defined(ARDUINO\_ARCH\_SAM)

// SAM-specific code

#else

// generic, non-platform specific code

#endif

Alternatively, if a library only works on certain architectures, you can provide an explicit error message (instead of allowing the compilation to fail in a difficult to understand way):

#if defined(ARDUINO\_ARCH\_AVR)

// AVR-specific code

#elif defined(ARDUINO\_ARCH\_SAM)

// SAM-specific code

#else

#error “This library only supports boards with an AVR or SAM processor.”

#endif

# Old library format (pre-1.5)

In order to support old libraries (from Arduino 1.0.x), Arduino 1.5.x+ will also compile libraries missing a library.properties metadata file. As a result, these libraries should behave as they did in Arduino 1.0.x, although they will be available for all boards, including non-AVR ones (which wouldn’t have been present in 1.0.x).

## Arduino style guide

This is a guide for writing clear Arduino examples that can be read by beginners and advanced users alike. You don't have to code this way, but it helps if you want your code to be clear to all levels of users. This is not a set of hard and fast rules, it's a set of guidelines. Some of these guidelines might even conflict with each other. Use your judgment on when they're best followed, and if you're not sure, ask someone who'll be learning from what you write what makes the most sense. You might also be interested in the [API Style Guide for Arduino](https://www.arduino.cc/en/Reference/APIStyleGuide).

### Writing a tutorial (most of this is borrowed from various editors over the years)

Write in the active voice.

Write clearly and conversationally, as if the person following your instructions were there in the room with you.

When giving instruction, write in the second person, so the reader understands that she's the one who'll be doing it.

Use short, simple, declarative sentences and commands rather than compound sentences. It's easier for the reader to digest one instruction at a time.

Give directions in no uncertain terms like so:

"Next, you'll read the sensor..."

"Make a variable called thisPin..."

Avoid phrases that add no information. Don't tell the reader that "You want to set the pins", just tell her "Set the pins."

Use pictures and schematics rather than just schematics alone. Many electronics hobbyists don't read schematics.

Check your assumptions. Does the reader understand all the concepts you've used in your tutorial? If not, explain them or link to another tutorial that does.

Explain things conceptually, so the reader has a big picture of what he's going to do. Then lay out instructions on how to use it step-by-step.

Whenever you use a technical term for the first time, define it. Have someone else check that you defined all new terms. There are probably one or two that you missed.

Be consistent with the terms you use. If you refer to a component or concept by a new name, make the relationship to the other name explicit. Don't use two terms interchangeably unless you tell the reader that they are interchangeable.

Don't use acronyms or abbreviations without spelling them out first.

Make your example do one thing well. Don't combine concepts or functions unless it's a tutorial about combining concepts.

### Writing Example Code

Efficiency is not paramount; readability is.

The most important users of Arduino are beginners and people who don't care about code, but about getting projects done.

Think generously about people who know less than you about code. Don't think they should understand some technical concept. They don't, and they're not stupid for not understanding. Your code should explain itself, or use comments to do the same. If it needs a complex concept like registers or interrupts or pointers, either explain it or skip it.

When forced to choose between technically simple and technically efficient, choose the former.

Introduce concepts only when they are useful and try to minimize the number of new concepts you introduce in each example. For example, at the very beginning, you can explain simple functions with no variable types other than int, nor for consts to define pin numbers. On the other hand, in an intermediate example, you might want to introduce peripheral concepts as they become useful. Concepts like using const ints to define pin numbers, choosing bytes over ints when you don't need more than 0 - 255, etc. are useful, but not central to getting started. So use them sparingly, and explain them when they're new to your lesson plan.

Put your setup() and your loop() at the beginning of the program. They help beginners to get an overview of the program, since all other functions are called from those two.

#### Commenting Your Code

Comment every variable or constant declaration with a description of what the variable does.

Comment every code block. Do it before the block if possible, so the reader knows what's coming

Comment every for loop

Use verbose if statements. For simplicity to the beginning reader, use the block format for everything, i.e. avoid this:

if (somethingIsTrue) doSomething;

Instead, use this:

if (somethingIsTrue == TRUE) {

doSomething;

}

Avoid pointers

Avoid #defines

#### Variables

Avoid single letter variable names. Make them descriptive

Avoid variable names like val or pin. Be more descriptive, like buttonState or switchPin

If you want to define pin names and other quantities which won't change, use const ints. They're less messy than #defines, yet still give you a way to teach the difference between a variable and a constant.

Use the wiring/Processing-style variable types, e.g. boolean,char,byte,int,unsigned int,long,unsigned long,float,double,string,array,void when possible, rather than uint8\_t, etc. The former are explained in the documentation, and less terse names.

Avoid numbering schemes that confuse the user, e.g.:

pin1 = 2

pin2 = 3

etc.

If you need to renumber pins, consider using an array, like this:

int myPins[] = { 2, 7, 6, 5, 4, 3 };

This allows you to refer to the new pin numbers using the array elements, like this:

digitalWrite(myPins[1], HIGH); // turns on pin 7

It also allows you to turn all the pins on or off in the sequence you want, like this:

for (int thisPin = 0; thisPin < 6; thisPin++) {

digitalWrite(myPins[thisPin], HIGH);

delay(500);

digitalWrite(myPins[thisPin], LOW);

delay(500);

}

### Explain the code at the start

Here's a good title block:

/\*

Sketch title

Describe what it does in layman's terms. Refer to the components

attached to the various pins.

The circuit:

\* list the components attached to each input

\* list the components attached to each output

Created day month year

By author's name

Modified day month year

By author's name

http://url/of/online/tutorial.cc

\*/

### Circuits

For digital input switches, the default is to use a pulldown resistor on the switch rather than a pullup. That way, the logic of a switch's interaction makes sense to the non-engineer.

Keep your circuits simple. For example, bypass capacitors are handy, but most simple inputs will work without them. If a component is incidental, explain it later. [Reference Home](https://www.arduino.cc/en/Reference/HomePage)

Corrections, suggestions, and new documentation should be posted to the [*Forum*](http://arduino.cc/forum/index.php/board,23.0.html).

The text of the Arduino reference is licensed under a [Creative Commons Attribution-ShareAlike 3.0 License](http://creativecommons.org/licenses/by-sa/3.0/). Code samples in the reference are released into the public domain.