Library specification

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

- rev.1 has been implemented starting with Arduino IDE version 1.5.3 (now superseded by rev.2)
- rev.2 will be implemented starting from Arduino IDE version 1.5.6
- rev.2.1 will be implemented starting from Arduino IDE version 1.6.10
- rev.2.2 will be implemented starting from Arduino IDE version 1.8.10

This new library format is intended to be used in tandem with **Library Manager**, available since Arduino IDE 1.6.2. The Library Manager allows users to automatically download and install libraries needed in their projects, with an easy to use graphical interface in the Arduino IDE/Arduino IDE 2.0 and Arduino Web Editor as well as `arduino-cli lib`.

More information about how Library Manager works is available here.

Arduino development software 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

- Arduino library style guide
- Library dependency resolution process documentation

1.5 library format (rev. 2.2)

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. It must be located in the root of the library folder.

library.properties file format

The library.properties file is a key=value properties list. Every field in this file is UTF-8 encoded. Unless noted otherwise below, **all fields are required**. 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 (-). They must start with a letter or number. They must contain at least one letter. Note that libraries with a `name` value starting with `Arduino` will no longer be allowed addition to the Library Manager index as these names are now reserved for official Arduino libraries.
- **version** - version of the library. Version should be semver 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** will be prepended to this, so you should start by writing the second sentence here
- **category** - (defaults to `Uncategorized`) allowed values:
  - 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. This is used for the "More info" links in Library Manager
- **architectures** - (defaults to `*`) a comma separated list of architectures supported by the library. If the library doesn't contain architecture specific code use `*` to match all architectures. This field is used as one factor in determining priority when multiple libraries match an `#include` directive and to provide a warning message when the library is compiled for a board of an architecture that doesn't match any on the list.
- **depends** - **(available from Arduino IDE 1.8.10/Arduino CLI 0.7.0)** (optional) a comma-separated list of dependencies (libraries that are needed to build the current library). The Arduino IDE's Library Manager will offer to install the dependencies during installation of the library. `arduino-cli lib install` will automatically install the dependencies. Since spaces are allowed in the `name` of a library, but not commas, you can refer to libraries containing spaces in the name without ambiguity for example:
  `depends=Very long library name, Another library with long-name`
  Version constraints for the dependency may be specified in parentheses after the name:
  `depends=ArduinoHttpClient (>=1.0.0)`
- **dot_a_linkage** - **(available from Arduino IDE 1.6.0 / arduino-builder 1.0.0-beta13)** (optional) 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 is required.
- **includes** - **(available from Arduino IDE 1.6.10)** (optional) a comma separated list of files of the library to be added to the sketch as `#include <...>` lines. This property is used with the "Include library" command in the Arduino IDE. If the `includes` property is missing, all the header files (.h) on the root source folder are included.
- **precompiled** - **(available from Arduino IDE 1.8.6/arduino-builder 1.4.0)** (optional) enables support for .a (archive) and .so (shared object) files. See the "Precompiled binaries" section for documentation of the required location in the library for these files. The static library should be linked as an ldflag. The **precompiled** field has two supported values, which control how any source files in the library are handled:
  - true - Source files are always compiled. This is useful for "mixed" libraries, such as those that contain both open source code and the precompiled binary of a closed source component. Support for "mixed" libraries was inadvertently lost in Arduino IDE 1.8.12/arduino-builder 1.5.2/Arduino CLI 0.8.0, and returned in Arduino IDE 1.8.13/arduino-builder 1.5.3/Arduino CLI 0.11.0.
  - full - **(available from Arduino IDE 1.8.13/arduino-builder 1.5.3/Arduino CLI 0.11.0)** If the library provides a precompiled library for the board being compiled for, the source files will not be compiled. If no precompiled library was provided for the selected board, source files are compiled as a fallback. This is useful for precompiling the library to reduce compilation time for specific target hardware, but also providing support for arbitrary boards by compiling the library on demand.
- **ldflags** - **(available from Arduino IDE 1.8.6/arduino-builder 1.4.0)** (optional) the linker flags to be added. Ex: `ldflags=-lm`

Example:

```
name=WebServer
version=1.0.0
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
depends=ArduinoHttpClient
```

Version constraints

(available from Arduino IDE 2.0.0-beta.3/Arduino CLI 0.7.0)

By default, the latest version of a dependency specified in the `depends` field of `library.properties` is installed along with the library. Specifying an exact version or range of versions is also supported.

The following operators are available:

`=`	equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to

`!`	NOT [1]
`&&`	AND
`||`	OR
`(` `)`	constraint group

[1] Available from Arduino IDE 2.0.0-rc7/Arduino CLI 0.22.0

EXAMPLES

If the library "ArduinoHttpClient" has the following releases:

- `0.1.0`
- `1.0.0`
- `2.0.0`
- `2.1.0`

The version of it installed as a dependency would be as follows:

`depends` field value	Installs version
`ArduinoHttpClient`	`2.1.0`
`ArduinoHttpClient (=1.0.0)`	`1.0.0`
`ArduinoHttpClient (>1.0.0)`	`2.1.0`
`ArduinoHttpClient (>=1.0.0)`	`2.1.0`
`ArduinoHttpClient (<2.0.0)`	`1.0.0`
`ArduinoHttpClient (<=2.0.0)`	`2.0.0`
`ArduinoHttpClient (!=1.0.0)`	`2.1.0`
`ArduinoHttpClient (>1.0.0 && <2.1.0)`	`2.0.0`
`ArduinoHttpClient (<1.0.0 || >2.0.0)`	`2.1.0`
`ArduinoHttpClient ((>0.1.0 && <2.0.0) || >2.1.0)`	`1.0.0`

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.

Library Root folder

The library root folder name must start with a basic letter (A-Z or a-z) or number (0-9), followed by basic letters, numbers, underscores (_), dots (.) and dashes (-). The maximum length is 63 characters.

Source code

For libraries intended to be used with Arduino IDE 1.5.x+ only, 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

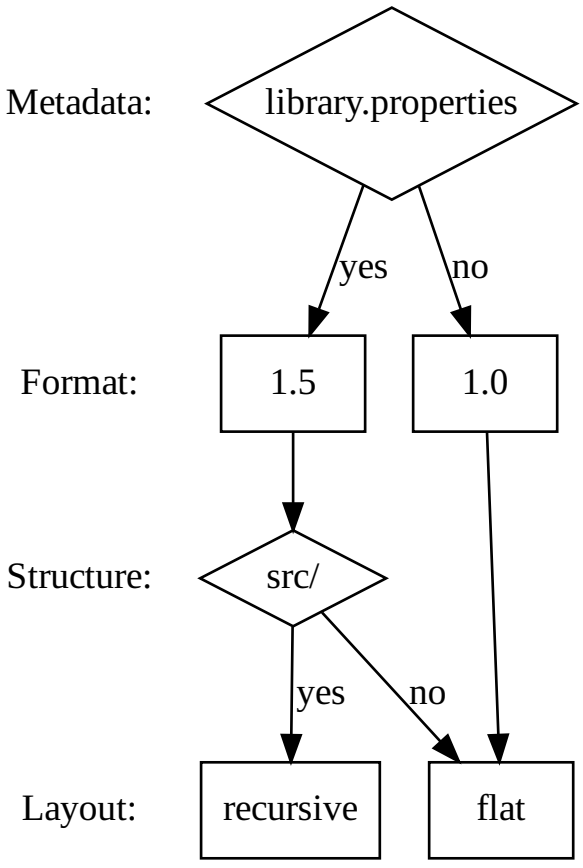
path (both when compiling the sketch and the library). When the user imports a library into their sketch (from the Arduino IDE's "Sketch > Include Library" menu or the Arduino Web Editor's "Include" button), the default behavior (configurable via the library.properties `includes` field) is for an `#include` statement to 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 IDE 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 format libraries to compile under Arduino IDE 1.5.x+ as well, and vice-versa. If a library only needs to run on Arduino IDE 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 Arduino IDE 1.0.x doesn't support recursive compilation, backwards compatibility wouldn't be possible anyway).

LIBRARY LAYOUT



Layout	root compiled	`src` compiled	`utility` compiled
recursive	no	recursively	no
flat	yes	no	yes

Precompiled binaries

The `precompiled` field of library.properties enables support for the use of precompiled libraries. This requires providing .a (archive) or .so (shared object) files which are compiled for a particular processor architecture. The target architecture of the files is indicated by folder names.

The binaries 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 filenames of the compiled binaries should start with `lib` (e.g., `libFoo.a`).

(available from Arduino IDE 1.8.12/arduino-builder 1.5.2/Arduino CLI 0.8.0) The floating point ABI configuration of ARM core microcontrollers may be adjusted via compiler flags. An extra subfolder level can be used to provide files compiled for a specific floating point configuration: `src/{build.mcu}/{build.fpu}-{build.float-abi}`, where `{build.fpu}` is the value of the `-mfpu` compiler flag and `{build.float-abi}` is the value of the `-mfloat-abi` compiler flag. **(available from Arduino IDE 1.8.13/arduino-builder 1.5.3/Arduino CLI 0.11.0)** If floating point configuration flags are used but no folder matching that configuration is found, `src/{build.mcu}` is used as a fallback.

Below is an example library `src` folder structure that provides:

- Header file containing the declarations for the library API.
- Source file to use as a fallback for other architectures (`precompiled=full` mode).
- Archive file for the ARM Cortex M0+ architecture of the Arduino SAMD boards.
- Archive file for the ARM Cortex M4 architecture of the Arduino Nano 33 BLE, as a fallback for backwards compatibility with Arduino development software before the floating point configuration support was added.
- Archive file for the ARM Cortex M4 architecture of the Arduino Nano 33 BLE, compiled for the `-mfloat-abi=softfp -mfpu=fpv4-sp-d16` floating point ABI configuration.

```
Servo/src/Servo.h
Servo/src/Servo.cpp
Servo/src/cortex-m0plus/libServo.a
Servo/src/cortex-m4/libServo.a
Servo/src/cortex-m4/fpv4-sp-d16-softfp/libServo.a
```

Library Examples

Library examples must be placed in the **examples** folder. Note that the **examples** folder name 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 Arduino IDE and Arduino Web Editor.

More information:

- Arduino sketch specification
- Style guide for Arduino examples

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 Arduino development software; you are free to put anything inside.

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.txt file:

```
# Syntax Coloring Map For ExampleLibrary

# Datatypes (KEYWORD1)
Test    KEYWORD1

# Methods and Functions (KEYWORD2)
doSomething KEYWORD2

# Instances (KEYWORD2)

# Constants (LITERAL1)
```

This file would cause the Arduino IDE to highlight `Test` as a data type, 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

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 page to open via the Arduino IDE's **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	KEYWORD_TOKENTYPE equivalent
`RESERVED_WORD`	`editor.reserved_word.style`	`KEYWORD3`
`RESERVED_WORD_2`	`editor.reserved_word_2.style`	`LITERAL1`
`DATA_TYPE`	`editor.data_type.style`	`KEYWORD1`
`PREPROCESSOR`	`editor.preprocessor.style`	`KEYWORD3`
`LITERAL_BOOLEAN`	`editor.literal_boolean.style`	`LITERAL1`

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 file must also be present. The Library Manager indexer 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

Libraries placed in the `libraries` subfolder of the sketchbook folder (AKA "user directory") 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 architecture folder of the board's

platform). For example, `ARDUINO_ARCH_AVR` will be defined when compiling for AVR-based boards.

An 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
```