Platform specification

This is the Arduino platform specification, for use with Arduino development software starting from the Arduino IDE 1.5.x series.

Platforms add support for new boards to the Arduino development software. They are installable either via Boards Manager or manual installation to the *hardware* folder of Arduino's sketchbook folder (AKA "user directory").
A platform may consist of as little as a single configuration file.

Hardware Folders structure

The new hardware folders have a hierarchical structure organized in two levels:

- the first level is the vendor/maintainer
- the second level is the supported architecture

A vendor/maintainer can have multiple supported architectures. For example, below we have three hardware vendors called "arduino", "yyyyy" and "xxxxx":

```
hardware/arduino/avr/...     - Arduino - AVR Boards
hardware/arduino/sam/...     - Arduino - SAM (32bit ARM) Boards
hardware/yyyyy/avr/...       - Yyy - AVR
hardware/xxxxx/avr/...       - Xxx - AVR
```

The vendor "arduino" has two supported architectures (AVR and SAM), while "xxxxx" and "yyyyy" have only AVR.

Architecture values are case sensitive (e.g. `AVR` != `avr`).

If possible, follow existing architecture name conventions when creating hardware packages. Use the vendor folder name to differentiate your package. The architecture folder name is used to determine library compatibility and to permit referencing resources from another core of the same architecture, so use of a non-standard architecture name can have a harmful effect.

Architecture configurations

Each architecture must be configured through a set of configuration files:

- **platform.txt** contains definitions for the CPU architecture used (compiler, build process parameters, tools used for upload, etc.)
- **boards.txt** contains definitions for the boards (board name, parameters for building and uploading sketches, etc.)
- **programmers.txt** contains definitions for external programmers (typically used to burn bootloaders or sketches on a blank CPU/board)

Configuration files format

A configuration file is a list of "key=value" properties. The **value** of a property can be expressed using the value of another property by putting its name inside brackets "{" "}". For example:

```
compiler.path=/tools/g++_arm_none_eabi/bin/
compiler.c.cmd=arm-none-eabi-gcc
[....]
recipe.c.o.pattern={compiler.path}{compiler.c.cmd}
```

In this example the property **recipe.c.o.pattern** will be set to **/tools/g++_arm_none_eabi/bin/arm-none-eabi-gcc**, which is the composition of the properties **compiler.path** and **compiler.c.cmd**.

Comments

Lines starting with **#** are treated as comments and will be ignored.

```
# Like in this example
# --------------------
# I'm a comment!
```

Automatic property override for specific OS

We can specify an OS-specific value for a property. For example the following file:

```
tools.bossac.cmd=bossac
tools.bossac.cmd.windows=bossac.exe
```

will set the property **tools.bossac.cmd** to the value **bossac** on Linux and macOS and **bossac.exe** on Windows. Supported suffixes are `.linux`, `.windows` and `.macosx`.

Global Predefined properties

The following automatically generated properties can be used globally in all configuration files:

- `{runtime.platform.path}`: the absolute path of the board platform folder (i.e. the folder containing boards.txt)
- `{runtime.hardware.path}`: the absolute path of the hardware folder (i.e. the folder containing the board platform folder)
- `{runtime.ide.path}`: the absolute path of the Arduino IDE or Arduino CLI folder
- `{runtime.ide.version}`: the version number of the Arduino IDE as a number (this uses two digits per version number component, and removes the points and leading zeroes, so Arduino IDE 1.8.3 becomes `01.08.03` which becomes `runtime.ide.version=10803`). When using Arduino development software other than the Arduino IDE, this is set to a meaningless version number.
- `{ide_version}`: Compatibility alias for `{runtime.ide.version}`
- `{runtime.os}`: the running OS ("linux", "windows", "macosx")
- `{software}`: set to "ARDUINO"
- `{name}`: platform vendor name
- `{_id}`: board ID of the board being compiled for
- `{build.fqbn}`: the FQBN (fully qualified board name) of the board being compiled for. The FQBN follows the format: `VENDOR:ARCHITECTURE:BOARD_ID[:MENU_ID=OPTION_ID[,MENU2_ID=OPTION_ID ...]]`
- `{build.source.path}`: Path to the sketch being compiled. If the sketch is in an unsaved state, it will the path of its temporary folder.
- `{build.library_discovery_phase}`: set to 1 during library discovery and to 0 during normal build. A macro defined with this property can be used to disable the inclusion of heavyweight headers during discovery to reduce compilation time. This property was added in Arduino IDE 1.8.14/Arduino Builder 1.6.0/Arduino CLI 0.12.0. Note: with the same intent, `-DARDUINO_LIB_DISCOVERY_PHASE` was added to `recipe.preproc.macros` during library discovery in Arduino Builder 1.5.3/Arduino CLI 0.10.0. That flag was replaced by the more flexible `{build.library_discovery_phase}` property.
- `{compiler.optimization_flags}`: see "Sketch debugging configuration" for details
- `{extra.time.utc}`: Unix time (seconds since 1970-01-01T00:00:00Z) according to the machine the build is running on
- `{extra.time.local}`: Unix time with local timezone and DST offset
- `{extra.time.zone}`: local timezone offset without the DST component
- `{extra.time.dst}`: local daylight savings time offset

Compatibility note: Versions before Arduino IDE 1.6.0 only used one digit per version number component in `{runtime.ide.version}` (so 1.5.9 was `159`, not `10509`).

platform.txt

The platform.txt file contains information about a platform's specific aspects (compilers command line flags, paths, system libraries, etc.).

The following meta-data must be defined:

```
name=Arduino AVR Boards
version=1.5.3
```

The **name** will be shown as the Arduino IDE's Board menu section title or the Name field of `arduino-cli core list`'s output for the platform.
The **version** is currently unused, it is reserved for future use (probably together with the Boards Manager to handle dependencies on cores).

Build process

The platform.txt file is used to configure the build process. This is done through a list of **recipes**. Each recipe is a command line expression that explains how to call the compiler (or other tools) for every build step and which parameter should be passed.

The Arduino development software, before starting the build, determines the list of files to compile. The list is composed of:

- the user's Sketch
- source code in the selected board's Core
- source code in the Libraries used in the sketch

A temporary folder is created to store the build artifacts whose path is available through the global property **{build.path}**. A property **{build.project_name}** with the name of the project and a property **{build.arch}** with the name of the architecture is set as well.

- `{build.path}`: The path to the temporary folder to store build artifacts
- `{build.project_name}`: The project name
- `{build.arch}`: The MCU architecture (avr, sam, etc...)

There are some other **{build.xxx}** properties available, that are explained in the boards.txt section of this guide.

Security credential properties

Some of them allow specifying trusted security credentials (signing and encryption keys) that can be used by a "secure boot" system:

- `build.keys.keychain`: for the directory containing the keys
- `build.keys.sign_key`: for the signing key
- `build.keys.encrypt_key`: for the encryption key

If any of these properties are defined, the others are required.

These properties can be overwritten respectively with `--keys-keychain`, `--sign-key`, `--encrypt-key` compile flags in the Arduino CLI.

Recipes to compile source code

We said that the Arduino development software determines a list of files to compile. Each file can be source code written in C (.c files), C++ (.cpp/.cxx/.cc files) or Assembly (.S files). Every language is compiled using its respective **recipe**:

- `recipe.c.o.pattern`: for C files (.c)
- `recipe.cpp.o.pattern`: for CPP files (.cpp/.cxx/.cc)
- `recipe.S.o.pattern`: for Assembly files (.S)

(an optional `recipe.cxx.o.pattern` and `recipe.cc.o.pattern` may be provided, if `.cxx` or `.cc` needs special handling, but it's not required and we do not recommend it)

The recipes can be built concatenating the following automatically generated properties (for each file compiled):

- `{includes}`: the list of include paths in the format "-I/include/path -I/another/path...."
- `{source_file}`: the path to the source file
- `{object_file}`: the path to the output file

For example the following is used for AVR:

```
## Compiler global definitions
compiler.path={runtime.ide.path}/tools/avr/bin/
compiler.c.cmd=avr-gcc
compiler.c.flags=-c -g -Os -w -ffunction-sections -fdata-sections -MMD

[......]

## Compile c files
recipe.c.o.pattern="{compiler.path}{compiler.c.cmd}" {compiler.c.flags} -mmcu={build.mcu} -DF_CPU={build.f_cpu} -DARDUINO=
{runtime.ide.version} -DARDUINO_{build.board} -DARDUINO_ARCH_{build.arch} {build.extra_flags} {includes} "{source_file}" -o
"{object_file}"
```

Note that some properties, like **{build.mcu}** for example, are taken from the **boards.txt** file which is documented later in this specification.

Recipes to build the core.a archive file

The core of the selected board is compiled as described in the previous paragraph, but the object files obtained from the compile are also archived into a static library named *core.a* using the **recipe.ar.pattern**.

The recipe can be built concatenating the following automatically generated properties:

- `{object_file}`: the object file to include in the archive
- `{archive_file_path}`: fully qualified archive file (ex. "/path/to/core.a"). This property was added in Arduino IDE 1.6.6/arduino builder 1.0.0-beta12 as a replacement for `{build.path}/{archive_file}`.
- `{archive_file}`: the name of the resulting archive (ex. "core.a")

For example, Arduino provides the following for AVR:

```
compiler.ar.cmd=avr-ar
compiler.ar.flags=rcs
```

```
[......]

## Create archives
recipe.ar.pattern="{compiler.path}{compiler.ar.cmd}" {compiler.ar.flags} "{archive_file_path}" "{object_file}"
```

Recipes for linking

All the artifacts produced by the previous steps (sketch object files, libraries object files and core.a archive) are linked together using the **recipe.c.combine.pattern**.

The recipe can be built concatenating the following automatically generated properties:

- `{object_files}` : the list of object files to include in the archive ("file1.o file2.o ....")
- `{archive_file_path}` : fully qualified archive file (ex. "/path/to/core.a"). This property was added in Arduino IDE 1.6.6/arduino builder 1.0.0-beta12 as a replacement for `{build.path}/{archive_file}` .
- `{archive_file}` : the name of the core archive file (ex. "core.a")
- `{compiler.libraries.ldflags}` : the linking flags for precompiled libraries, which consist of automatically generated `-L` flags for the library path and `-l` flags for library files, as well as any custom flags provided via the `ldflags` field of library.properties. In order to support precompiled libraries, platform.txt must contain a definition of `compiler.libraries.ldflags` , to which any automatically generated flags will be appended. Support for precompiled libraries was added in Arduino IDE 1.8.6/arduino-builder 1.4.0.

For example the following is used for AVR:

```
compiler.c.elf.flags=-Os -Wl,--gc-sections
compiler.c.elf.cmd=avr-gcc

compiler.libraries.ldflags=

[......]

## Combine gc-sections, archives, and objects
recipe.c.combine.pattern="{compiler.path}{compiler.c.elf.cmd}" {compiler.c.elf.flags} -mmcu={build.mcu} -o
"{build.path}/{build.project_name}.elf" {object_files} {compiler.libraries.ldflags} "{archive_file_path}" "-L{build.path}" -lm
```

Recipes for extraction of executable files and other binary data

An arbitrary number of extra steps can be performed at the end of objects linking. These steps can be used to extract binary data used for upload and they are defined by a set of recipes with the following format:

```
recipe.objcopy.FILE_EXTENSION_1.pattern=[.....]
recipe.objcopy.FILE_EXTENSION_2.pattern=[.....]
[.....]
```

`FILE_EXTENSION_x` must be replaced with the extension of the extracted file, for example the AVR platform needs two files a `.hex` and a `.eep` , so we made two recipes like:

```
recipe.objcopy.eep.pattern=[.....]
recipe.objcopy.hex.pattern=[.....]
```

There are no specific properties set by the Arduino development software here.

A full example for the AVR platform can be:

```
## Create eeprom
recipe.objcopy.eep.pattern="{compiler.path}{compiler.objcopy.cmd}" {compiler.objcopy.eep.flags}
"{build.path}/{build.project_name}.elf" "{build.path}/{build.project_name}.eep"

## Create hex
recipe.objcopy.hex.pattern="{compiler.path}{compiler.elf2hex.cmd}" {compiler.elf2hex.flags}
"{build.path}/{build.project_name}.elf" "{build.path}/{build.project_name}.hex"
```

Recipes to compute binary sketch size

At the end of the build the Arduino development software shows the final binary sketch size to the user. The size is calculated using the recipe **recipe.size.pattern**. The output of the command executed using the recipe is parsed through the regular expressions set in the properties:

- **recipe.size.regex**: Program storage space used.
- **recipe.size.regex.data**: Dynamic memory used by global variables.

For AVR we have:

```
compiler.size.cmd=avr-size
```

```
[....]
## Compute size
recipe.size.pattern="{compiler.path}{compiler.size.cmd}" -A "{build.path}/{build.project_name}.hex"
recipe.size.regex=^(?:\.text|\.data|\.bootloader)\s+([0-9]+).*
recipe.size.regex.data=^(?:\.data|\.bss|\.noinit)\s+([0-9]+).*
```

Two properties can be used to define the total available memory:

- `{upload.maximum_size}` : available program storage space
- `{upload.maximum_data_size}` : available dynamic memory for global variables

If the binary sketch size exceeds the value of these properties, the compilation process fails.

This information is displayed in the console output after compiling a sketch, along with the relative memory usage value:

```
Sketch uses 924 bytes (2%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.
```

Recipes to compute binary sketch size for more complex systems (since Arduino CLI >=0.21.0)

A platform may provide a tool for the specific purpose to analyze the binaries and compute the sketch size and memory usage statistics. This is especially useful for boards with non-trivial memory layouts where the classic reg-exp based approach is not sufficient.

The command line to run is specified with the recipe **recipe.advanced_size.pattern**.

The expected output from the tool is a JSON object with the following format:

```
{
  "output": "Your sketch uses 2200 bytes of program memory out of 8192 (27%)\nThe static RAM used is 200 bytes (of 2048 max)",
  "severity": "info",
  "sections": [
    { "name": "text", "size": 2200, "max_size": 8192 },
    { "name": "data", "size": 200, "max_size": 2048 }
  ]
}
```

The meaning of the fields is the following:

- `output` : is a preformatted text that is displayed as-is in console.
- `severity` : indicates the warning level of the output messages, it must be `info`, `warning` or `error`. Warnings and errors are displayed in red (or in a different color than normal output). Errors will make the build/upload fail.
- `sections` : is an array containing the memory sections and their usage level. This array is used to report memory usage in a machine-readable format if requested by the user. Each item represents a memory section and may contain the following fields
  - `name` : an identifier for the section
  - `size` : the sketch size for the section
  - `max_size` : the maximum size for the section

When the `severity` is set to `error` the build/upload is interrupted and an exception is returned to the calling process. In this case an extra exception message must be provided through the `error` field, for example:

```
{
  "output": "Your sketch uses 12200 bytes of program memory out of 8192 (149%))\nThe static RAM used is 200 bytes (of 2048
max)",
  "severity": "error",
  "error": "Sketch is too big!",
  "sections": [
    { "name": "text", "size": 12200, "max_size": 8192 },
    { "name": "data", "size": 200, "max_size": 2048 }
  ]
}
```

This means that the `sections` part is **NOT** used to automatically check if the sketch size exceeds the available memory: this check is now delegated to the tool that must report a `"severity":"error"` with a meaningful error message.

If both **recipe.size.pattern** and **recipe.advanced_size.pattern** are present then **recipe.advanced_size.pattern** will be used. Since the **recipe.advanced_size.pattern** feature is available starting from Arduino CLI>=0.21.0, to maximize backward compatibility, we recommend to provide both **recipe.size.pattern** and **recipe.advanced_size.pattern** if possible, so the old versions of the IDE/CLI will continue to work (even with a less detailed memory usage report).

Recipes to export compiled binary

When you do a **Sketch > Export compiled Binary** in the Arduino IDE, the compiled binary is copied from the build folder to the sketch folder. Two

binaries are copied; the standard binary, and a binary that has been merged with the bootloader file (identified by the `.with_bootloader` in the filename).

Two recipes affect how **Export compiled Binary** works:

- **recipe.output.tmp_file**: Defines the binary's filename in the build folder.
- **recipe.output.save_file**: Defines the filename to use when copying the binary file to the sketch folder.

As with other processes, there are pre and post build hooks for **Export compiled Binary**.

The **recipe.hooks.savehex.presavehex.NUMBER.pattern** and **recipe.hooks.savehex.postsavehex.NUMBER.pattern** hooks (but not **recipe.output.tmp_file** and **recipe.output.save_file**) can be built concatenating the following automatically generated properties:

- `{sketch_path}`: the absolute path of the sketch folder

Recipe to run the preprocessor

For detecting which libraries to include in the build, and for generating function prototypes, (just) the preprocessor is run. For this, the **recipe.preproc.macros** recipe exists. This recipe must run the preprocessor on a given source file, writing the preprocessed output to a given output file, and generate (only) preprocessor errors on standard output. This preprocessor run should happen with the same defines and other preprocessor-influencing-options as for normally compiling the source files.

The recipes can be built concatenating other automatically generated properties (for each file compiled):

- `{includes}`: the list of include paths in the format "-I/include/path -I/another/path...."
- `{source_file}`: the path to the source file
- `{preprocessed_file_path}`: the path to the output file

For example the following is used for AVR:

```
preproc.macros.flags=-w -x c++ -E -CC
recipe.preproc.macros="{compiler.path}{compiler.cpp.cmd}" {compiler.cpp.flags} {preproc.macros.flags} -mmcu={build.mcu}
-DF_CPU={build.f_cpu} -DARDUINO={runtime.ide.version} -DARDUINO_{build.board} -DARDUINO_ARCH_{build.arch}
{compiler.cpp.extra_flags} {build.extra_flags} {includes} "{source_file}" -o "{preprocessed_file_path}"
```

Note that the `{preprocessed_file_path}` might point to (your operating system's equivalent) of `/dev/null`. In this case, also passing `-MMD` to gcc is problematic, as it will try to generate a dependency file called `/dev/null.d`, which will usually result in a permission error. Since platforms typically include `{compiler.cpp.flags}` here, which includes `-MMD`, the `-MMD` option is automatically filtered out of the `recipe.preproc.macros` recipe to prevent this error.

If **recipe.preproc.macros** is not defined, it is automatically generated from **recipe.cpp.o.pattern**.

Note that older Arduino IDE versions used the **recipe.preproc.includes** recipe (which is not documented here) to determine includes. Since Arduino IDE 1.6.7 (arduino-builder 1.2.0) this was changed and **recipe.preproc.includes** is no longer used.

Pre and post build hooks (since Arduino IDE 1.6.5)

You can specify pre and post actions around each recipe. These are called "hooks". Here is the complete list of available hooks:

- `recipe.hooks.prebuild.NUMBER.pattern` (called before sketch preprocessing and libraries discovery)
- `recipe.hooks.sketch.prebuild.NUMBER.pattern` (called before sketch compilation)
- `recipe.hooks.sketch.postbuild.NUMBER.pattern` (called after sketch compilation)
- `recipe.hooks.libraries.prebuild.NUMBER.pattern` (called before libraries compilation)
- `recipe.hooks.libraries.postbuild.NUMBER.pattern` (called after libraries compilation)
- `recipe.hooks.core.prebuild.NUMBER.pattern` (called before core compilation)
- `recipe.hooks.core.postbuild.NUMBER.pattern` (called after core compilation)
- `recipe.hooks.linking.prelink.NUMBER.pattern` (called before linking)
- `recipe.hooks.linking.postlink.NUMBER.pattern` (called after linking)
- `recipe.hooks.objcopy.preobjcopy.NUMBER.pattern` (called before objcopy recipes execution)
- `recipe.hooks.objcopy.postobjcopy.NUMBER.pattern` (called after objcopy recipes execution)
- `recipe.hooks.savehex.presavehex.NUMBER.pattern` (called before savehex recipe execution)
- `recipe.hooks.savehex.postsavehex.NUMBER.pattern` (called after savehex recipe execution)

Example: you want to execute two commands before sketch compilation and one after linking. You'll add to your platform.txt:

```
recipe.hooks.sketch.prebuild.1.pattern=echo sketch compilation started at
```

```
recipe.hooks.sketch.prebuild.2.pattern=date

recipe.hooks.linking.postlink.1.pattern=echo linking is complete
```

Warning: hooks recipes are sorted before execution. If you need to write more than 10 recipes for a single hook, pad the number with a zero, for example:

```
recipe.hooks.sketch.prebuild.01.pattern=echo 1
recipe.hooks.sketch.prebuild.02.pattern=echo 2
...
recipe.hooks.sketch.prebuild.11.pattern=echo 11
```

Note: all the `pre*` hooks are executed while producing the "compilation database" (a JSON file with the list of commands to run to compile the sketch), but the `post*` hooks and all compile commands are skipped. See the `arduino-cli compile` command reference for more info.

Global platform.txt

Properties defined in a platform.txt created in the **hardware** subfolder of the Arduino IDE installation folder will be used for all platforms and will override local properties. This feature is currently only available when using the Arduino IDE.

platform.local.txt

Introduced in Arduino IDE 1.5.7. This file can be used to override properties defined in `platform.txt` or define new properties without modifying `platform.txt` (e.g. when `platform.txt` is tracked by a version control system). It must be placed in the same folder as the `platform.txt` it supplements.

boards.txt

This file contains definitions and metadata for the boards supported by the platform. Boards are referenced by their short name, the board ID. The settings for a board are defined through a set of properties with keys having the board ID as prefix.

For example, the board ID chosen for the Arduino Uno board is "uno". An extract of the Uno board configuration in boards.txt looks like:

```
[......]
uno.name=Arduino Uno
uno.build.mcu=atmega328p
uno.build.f_cpu=16000000L
uno.build.board=AVR_UNO
uno.build.core=arduino
uno.build.variant=standard
[......]
```

Note that all the relevant keys start with the board ID **uno.xxxxx**.

The **uno.name** property contains the human-friendly name of the board. This is shown in the Board menu of the IDEs, the "Board Name" field of Arduino CLI's text output, or the "name" key of Arduino CLI's JSON output.

The **uno.build.board** property is used to set a compile-time macro **ARDUINO_{build.board}** to allow use of conditional code between `#ifdef` s. If not defined, a **build.board** value is automatically generated and the Arduino development software outputs a warning. In this case the macro defined at compile time will be `ARDUINO_AVR_UNO`.

The other properties will override the corresponding global properties when the user selects the board. These properties will be globally available, in other configuration files too, without the board ID prefix:

```
uno.build.mcu           =>   build.mcu
uno.build.f_cpu         =>   build.f_cpu
uno.build.board         =>   build.board
uno.build.core          =>   build.core
uno.build.variant       =>   build.variant
```

This explains the presence of **{build.mcu}** or **{build.board}** in the platform.txt recipes: their value is overwritten respectively by **{uno.build.mcu}** and **{uno.build.board}** when the Uno board is selected! Moreover the following properties are automatically generated:

- `{build.core.path}`: The path to the selected board's core folder (inside the core platform, for example hardware/arduino/avr/core/arduino)
- `{build.system.path}`: The path to the core platform's system folder if available (for example hardware/arduino/sam/system)
- `{build.variant.path}`: The path to the selected board variant folder (inside the variant platform, for example hardware/arduino/avr/variants /micro)

If the platform supports pluggable discovery it may also declare a set of `upload_port.*` properties, these properties will be used to identify a board

by the discovery process when plugged in.

For example we could declare a series of `upload_port.vid` and `upload_port.pid` properties for the Uno like so:

```
uno.upload_port.vid.0=0x2341
uno.upload_port.pid.0=0x0043
uno.upload_port.vid.1=0x2341
uno.upload_port.pid.1=0x0001
uno.upload_port.vid.2=0x2A03
uno.upload_port.pid.2=0x0043
uno.upload_port.vid.3=0x2341
uno.upload_port.pid.3=0x0243
```

In this case we're using the board's USB VID/PID pair to identify it but `upload_port.*` properties can be anything that can help identify a certain board. For more detailed information see the board identification section of the pluggable discovery documentation.

Cores

Cores are placed inside the **cores** subfolder. Many different cores can be provided within a single platform. For example the following could be a valid platform layout:

- `hardware/arduino/avr/cores/` : Cores folder for "avr" architecture, package "arduino"
- `hardware/arduino/avr/cores/arduino` : the Arduino Core
- `hardware/arduino/avr/cores/rtos` : a hypothetical RTOS Core

The board's property **build.core** is used to find the core that must be compiled and linked when the board is selected. For example if a board needs the Arduino core the **build.core** variable should be set to:

```
uno.build.core=arduino
```

or if the RTOS core is needed, to:

```
uno.build.core=rtos
```

In any case the contents of the selected core folder are compiled and the core folder path is added to the include files search path.

ArduinoCore-API

Although much of the implementation of a core is architecture-specific, the standardized core API and the hardware independent components should be the same for every Arduino platform. In order to free platform authors from the burden of individually maintaining duplicates of this common code, Arduino has published it in a dedicated repository from which it may easily be shared by all platforms. In addition to significantly reducing the effort required to write and maintain a core, ArduinoCore-API assists core authors in providing the unprecedented level of portability between platforms that is a hallmark of the Arduino project.

See the arduino/ArduinoCore-API repository for more information.

Core Variants

Sometimes a board needs some tweaking on the default core configuration (different pin mapping is a typical example). A core variant folder is an additional folder that is compiled together with the core and allows platform developers to easily add specific configurations.

Variants must be placed inside the **variants** folder in the current architecture. For example, Arduino AVR Boards uses:

- `hardware/arduino/avr/cores` : Core folder for "avr" architecture, "arduino" package
- `hardware/arduino/avr/cores/arduino` : The Arduino core
- `hardware/arduino/avr/variants/` : Variant folder for "avr" architecture, "arduino" package
- `hardware/arduino/avr/variants/standard` : ATmega328 based variants
- `hardware/arduino/avr/variants/leonardo` : ATmega32U4 based variants

In this example, the Arduino Uno board needs the *standard* variant so the **build.variant** property is set to *standard*:

```
[.....]
uno.build.core=arduino
uno.build.variant=standard
[.....]
```

instead, the Arduino Leonardo board needs the *leonardo* variant:

```
[.....]
leonardo.build.core=arduino
leonardo.build.variant=leonardo
[.....]
```

In the example above, both Uno and Leonardo share the same core but use different variants.
In any case, the contents of the selected variant folder path is added to the include search path and its contents are compiled and linked with the sketch.

The parameter **build.variant.path** is automatically generated.

Board VID/PID

USB vendor IDs (VID) and product IDs (PID) identify USB devices to the computer. If the board uses a unique VID/PID pair, it may be defined in boards.txt:

```
uno.vid.0=0x2341
uno.pid.0=0x0043
uno.vid.1=0x2341
uno.pid.1=0x0001
```

The **vid** and **pid** properties end with an arbitrary number, which allows multiple VID/PID pairs to be defined for a board. The snippet above is defining the 2341:0043 and 2341:0001 pairs used by Uno boards.

The Arduino development software uses the **vid** and **pid** properties to automatically identify the boards connected to the computer. This convenience feature isn't available for boards that don't present a unique VID/PID pair.

Serial Monitor control signal configuration

Arduino boards that use a USB to TTL serial adapter chip for communication with the computer (e.g., Uno, Nano, Mega) often utilize the DTR (data terminal ready) or RTS (request to send) serial control signals as a mechanism for the Arduino development software to trigger a reset of the primary microcontroller. The adapter's DTR and RTS pins are set `LOW` when the control signals are asserted by the computer and this `LOW` level is converted into a pulse on the microcontroller's reset pin by an "auto-reset" circuit on the board. The auto-reset system is necessary to activate the bootloader at the start of an upload.

This system is also used to reset the microcontroller when Serial Monitor is started. The reset is convenient because it allows viewing all serial output from the time the program starts. In case the reset caused by opening Serial Monitor is not desirable, the control signal assertion behavior of Serial Monitor is configurable via the **serial.disableDTR** and **serial.disableRTS** properties. Setting these properties to `true` will prevent Serial Monitor from asserting the control signals when that board is selected:

```
[.....]
uno.serial.disableDTR=true
uno.serial.disableRTS=true
[.....]
```

Hiding boards

Adding a **hide** property to a board definition causes it to not be shown in the Arduino IDE's **Tools > Board** menu.

```
uno.hide=
```

The value of the property is ignored; it's the presence or absence of the property that controls the board's visibility.

programmers.txt

This file contains definitions for external programmers. These programmers are used by:

- The **Tools > Burn Bootloader** feature of the IDEs and `arduino-cli burn-bootloader`
- The **Sketch > Upload Using Programmer** feature of the IDEs and `arduino-cli upload --programmer <programmer ID>`

programmers.txt works similarly to boards.txt. Programmers are referenced by their short name: the programmer ID. The settings for a programmer are defined through a set of properties with keys that use the programmer ID as prefix.

For example, the programmer ID chosen for the "Arduino as ISP" programmer is "arduinoasisp". The definition of this programmer in programmers.txt looks like:

```
[......]
arduinoasisp.name=Arduino as ISP
```

```
arduinoasisp.protocol=stk500v1
arduinoasisp.program.speed=19200
arduinoasisp.program.tool=avrdude
arduinoasisp.program.extra_params=-P{serial.port} -b{program.speed}
[......]
```

These properties can only be used in the recipes of the actions that use the programmer (`erase` , `bootloader` , and `program`).

The **arduinoasisp.name** property defines the human-friendly name of the programmer. This is shown in the **Tools > Programmer** menu of the IDEs and the output of `arduino-cli upload --programmer list` and `arduino-cli burn-bootloader --programmer list` .

In Arduino IDE 1.8.12 and older, all programmers of all installed platforms were made available for use. Starting with Arduino IDE 1.8.13 (and in all relevant versions of other Arduino development tools), only the programmers defined by the board and core platform of the currently selected board are available. For this reason, platforms may now need to define copies of the programmers that were previously assumed to be provided by another platform.

Set a default programmer for a board (since Arduino CLI >=0.35.0, Arduino IDE >=2.3.0)

A default programmer for each board may be specified through the `programmer.default` directive in the board definition:

```
BOARD_ID.programmer.default=PROGRAMMER_ID
```

The default programmer will be selected automatically if the user do not specifiy or select another programmer. This may be useful for boards with an on-board programmer/debugger.

For example if we want to set Atmel ICE as the default programmer for the Arduino UNO we would add the following line to the `boards.txt` file:

```
uno.programmer.default=atmel-ice
```

Tools

The Arduino development software uses external command line tools to upload the compiled sketch to the board or to burn bootloaders using external programmers. For example, *avrdude* is used for AVR based boards and *bossac* for SAM based boards, but there is no limit, any command line executable can be used. The command line parameters are specified using **recipes** in the same way used for platform build process.

Tools are configured inside the platform.txt file. Every Tool is identified by a short name, the Tool ID. A tool can be used for different purposes:

- **upload** a sketch to the target board (using a bootloader preinstalled on the board)
- **program** a sketch to the target board using an external programmer
- **erase** the target board's flash memory using an external programmer
- burn a **bootloader** into the target board using an external programmer

Each action has its own recipe and its configuration is done through a set of properties having key starting with **tools** prefix followed by the tool ID and the action:

```
[....]
tools.avrdude.upload.pattern=[......]
[....]
tools.avrdude.program.pattern=[......]
[....]
tools.avrdude.erase.pattern=[......]
[....]
tools.avrdude.bootloader.pattern=[......]
[.....]
```

A tool may have some actions not defined (it's not mandatory to define all four actions).
Let's look at how the **upload** action is defined for avrdude:

```
tools.avrdude.path={runtime.tools.avrdude.path}
tools.avrdude.cmd.path={path}/bin/avrdude
tools.avrdude.config.path={path}/etc/avrdude.conf

tools.avrdude.upload.pattern="{cmd.path}" "-C{config.path}" -p{build.mcu} -c{upload.port.protocol} -P{upload.port.address}
-b{upload.speed} -D "-Uflash:w:{build.path}/{build.project_name}.hex:i"
```

The tool configuration properties are available globally without the prefix. For example, the **tools.avrdude.cmd.path** property can be used as **{cmd.path}** inside the recipe, and the same happens for all the other avrdude configuration variables.

How to retrieve tools path via `{runtime.tools.*}` properties

A **{runtime.tools.TOOLNAME.path}** and **{runtime.tools.TOOLNAME-TOOLVERSION.path}** property is generated for the tools provided by the current platform and for any other platform installed via Boards Manager.

See `{runtime.tools.*.path}` rules for details on how the runtime properties are determined.

Environment variables

All the tools launched to compile or upload a sketch will have the following environment variable set:

`ARDUINO_USER_AGENT`: contains the name and version of the client used by the user in HTTP user-agent format, for example `"arduino-cli/0.21.0"`. It may also contain multiple space-delimited entries like `"arduino-cli/0.21.0 ArduinoIDE/2.0.0-rc1"` if this information is available.

Pluggable discovery

Discovery tools are a special kind of tool used to find supported boards. A platform must declare one or more Pluggable Discoveries in its `platform.txt`. Discoveries can be referenced from other packages, including the `builtin` dummy package which contains the traditional discoveries.

There are two different syntaxes to declare discoveries. If the platform uses just one discovery:

```
pluggable_discovery.required=VENDOR_ID:DISCOVERY_NAME
```

instead if it needs multiple discoveries:

```
pluggable_discovery.required.0=VENDOR_ID:DISCOVERY_0_NAME
pluggable_discovery.required.1=VENDOR_ID:DISCOVERY_1_NAME
```

A platform that supports only boards connected via serial ports can easily use the `builtin` package's `serial-discovery` without creating a custom pluggable discovery:

```
pluggable_discovery.required=builtin:serial-discovery
```

if it also supports boards connected via the network, it can use the `builtin` package's `mdns-discovery`:

```
pluggable_discovery.required.0=builtin:serial-discovery
pluggable_discovery.required.1=builtin:mdns-discovery
```

Since the above syntax requires specifying a discovery via the `discoveryDependencies` field of the platform's package index, it might be cumbersome to use with manual installations. So we provide another syntax to ease development and beta testing:

```
pluggable_discovery.DISCOVERY_ID.pattern=DISCOVERY_RECIPE
```

`DISCOVERY_ID` must be replaced by a unique identifier for the particular discovery and `DISCOVERY_RECIPE` must be replaced by the command line to launch the discovery. An example could be:

```
## Teensy Ports Discovery
pluggable_discovery.teensy.pattern="{runtime.tools.teensy_ports.path}/hardware/tools/teensy_ports" -J2
```

We strongly recommend using this syntax only for development purposes and not on released platforms.

For backward compatibility, if a platform does not declare any discovery (using the `pluggable_discovery.*` properties in `platform.txt`) it will automatically inherit `builtin:serial-discovery` and `builtin:mdns-discovery` (but not other builtin discoveries that may be possibly added in the future).

For detailed information, see the Pluggable Discovery specification.

Pluggable monitor

Monitor tools are a special kind of tool used to let the user communicate with the supported boards.

A platform must declare one or more Pluggable Monitor in its `platform.txt` and bind them to a specific port protocol. Monitors can be referenced from other packages.

The following directive is used to bind a specific monitor tool to a specific port protocol:

```
pluggable_monitor.required.PROTOCOL=VENDOR_ID:MONITOR_NAME
```

where `PROTOCOL` must be replaced with the port protocol identifier and `VENDOR_ID:MONITOR_NAME` must be replaced with the monitor tool identifier.

The platform can support as many protocols as needed:

```
pluggable_monitor.required.PROTOCOL1=VENDOR_ID:MONITOR_NAME1
pluggable_monitor.required.PROTOCOL2=VENDOR_ID:MONITOR_NAME2
...
```

The above syntax requires specifying a monitor tool via the `monitorDependencies` field of the platform's package index. Since it might be cumbersome to use with manual installations, we provide another syntax to ease development and beta testing:

```
pluggable_monitor.pattern.PROTOCOL=MONITOR_RECIPE
```

where `MONITOR_RECIPE` must be replaced by the command line to launch the monitor tool for the specific `PROTOCOL`. An example could be:

```
pluggable_monitor.pattern.custom-ble="{runtime.tools.my-ble-monitor.path}/my-ble-monitor" -H
```

in this case the platform provides a new hypothetical `custom-ble` protocol monitor tool and the command line tool named `my-ble-monitor` is launched with the `-H` parameter to start the monitor tool. In this case the command line pattern may contain any extra parameter in the formula: this is different from the monitor tools installed through the `monitorDependencies` field that must run without any command line parameter.

We strongly recommend using this syntax only for development purposes and not on released platforms.

Built-in monitors

If a platform supports only boards connected via serial ports it can easily use the `builtin:serial-monitor` tool without creating a custom pluggable monitor:

```
pluggable_monitor.required.serial=builtin:serial-monitor
```

Backward compatibility

For backward compatibility, if a platform does not declare any discovery or monitor tool (using the `pluggable_discovery.*` or `pluggable_monitor.*` properties in `platform.txt` respectively) it will automatically inherit `builtin:serial-monitor` (but not other `builtin` monitor tools that may be possibly added in the future). This will allow all legacy non-pluggable platforms to migrate to pluggable monitor without disruption.

For detailed information, see the Pluggable Monitor specification.

Port configuration

Each pluggable monitor has its own default settings that can be overridden using the following board properties:

```
BOARD_ID.monitor_port.PROTOCOL.SETTING_NAME=SETTING_VALUE
```

where:

- `BOARD_ID` is the board identifier
- `PROTOCOL` is the port protocol
- `SETTING_NAME` and `SETTING_VALUE` are the port setting and the desired value

For example, let's suppose that a board needs the `baudrate` setting of the `serial` port to be `9600`, then the corresponding properties in the `boards.txt` file will be:

```
myboard.monitor_port.serial.baudrate=9600
```

The settings available in a specific pluggable monitor can be queried directly from it.

Legacy `serial.disableRTS` and `serial.disableDTR` properties

In the old Arduino IDE (<=1.8.x) we used the properties:

```
BOARD_ID.serial.disableRTS=true
BOARD_ID.serial.disableDTR=true
```

to disable RTS and DTR when opening the serial monitor. To keep backward compatibilty the properties above are automatically converted to the corresponding pluggable monitor properties:

```
BOARD_ID.monitor_port.serial.rts=off
BOARD_ID.monitor_port.serial.dtr=off
```

Verbose parameter

It is possible for the user to enable verbosity from the Preferences panel of the IDEs or Arduino CLI's `--verbose` flag. This preference is transferred to the command line using the **ACTION.verbose** property (where ACTION is the action we are considering).
When the verbose mode is enabled, the **tools.TOOL_ID.ACTION.params.verbose** property is copied into **ACTION.verbose**. When the verbose mode is disabled, the **tools.TOOL_ID.ACTION.params.quiet** property is copied into **ACTION.verbose**. Confused? Maybe an example will make things clear:

```
tools.avrdude.upload.params.verbose=-v -v -v -v
tools.avrdude.upload.params.quiet=-q -q
tools.avrdude.upload.pattern="{cmd.path}" "-C{config.path}" {upload.verbose} -p{build.mcu} -c{upload.protocol} -P{serial.port}
-b{upload.speed} -D "-Uflash:w:{build.path}/{build.project_name}.hex:i"
```

In this example if the user enables verbose mode, then **{upload.params.verbose}** is used in **{upload.verbose}**:

```
tools.avrdude.upload.params.verbose    =>    upload.verbose
```

If the user didn't enable verbose mode, then **{upload.params.quiet}** is used in **{upload.verbose}**:

```
tools.avrdude.upload.params.quiet      =>    upload.verbose
```

Sketch upload configuration

The Upload action is triggered when the user clicks on the "Upload" button on the IDE toolbar or uses `arduino-cli upload`. Arduino uses the term "upload" for the process of transferring a program to the Arduino board.

The **upload.tool.<protocol_name>** property determines the tool to be used for upload. A specific **upload.tool.<protocol_name>** property should be defined for every board in boards.txt:

```
[......]
uno.upload.tool.serial=avrdude
[......]
leonardo.upload.tool.serial=avrdude
leonardo.upload.tool.network=arduino_ota
[......]
```

Multiple protocols can be defined for each board. When the user tries to upload using a protocol not supported by the board, it will fallback to `default` if one was defined:

```
[......]
uno.upload.tool.default=avrdude
[......]
leonardo.upload.tool.default=avrdude
leonardo.upload.tool.network=arduino_ota
[......]
```

`default` is also used when no upload address is provided by the user. This can be used with tools that have built-in port detection (e.g., `openocd`).

For backward compatibility with IDE 1.8.15 and older the previous syntax is still supported:

```
uno.upload.tool=avrdude
```

The previous syntax is equivalent to:

```
uno.upload.tool.default=avrdude
```

Other upload parameters can also be defined for the board. For example, in the Arduino AVR Boards boards.txt we have:

```
[.....]
uno.name=Arduino Uno
uno.upload.tool.serial=avrdude
uno.upload.protocol=arduino
uno.upload.maximum_size=32256
uno.upload.speed=115200
[.....]
leonardo.name=Arduino Leonardo
leonardo.upload.tool.serial=avrdude
leonardo.upload.protocol=avr109
leonardo.upload.maximum_size=28672
leonardo.upload.speed=57600
leonardo.upload.use_1200bps_touch=true
leonardo.upload.wait_for_upload_port=true
[.....]
```

Most **{upload.XXXX}** variables are used later in the avrdude upload recipe in platform.txt:

```
[.....]
tools.avrdude.upload.pattern="{cmd.path}" "-C{config.path}" {upload.verbose} -p{build.mcu} -c{upload.port.protocol}
-P{upload.port.address} -b{upload.speed} -D "-Uflash:w:{build.path}/{build.project_name}.hex:i"
[.....]
```

If necessary the same property can be defined multiple times for different protocols:

```
leonardo.upload.serial.speed=57600
leonardo.upload.network.speed=19200
```

The two above properties will be available as **{upload.speed}**, the value will depend on the protocol used to upload.

Properties from pluggable discovery

If a platform supports pluggable discovery it can also use the port's properties returned by a discovery. For example, the following port metadata coming from a pluggable discovery:

```
   {
      "eventType": "add",
      "port": {
        "address": "/dev/ttyACM0",
        "label": "ttyACM0",
        "protocol": "serial",
        "protocolLabel": "Serial Port (USB)",
        "properties": {
          "pid": "0x804e",
          "vid": "0x2341",
          "serialNumber": "EBEABFD6514D32364E202020FF10181E",
          "name": "ttyACM0"
        }
      }
    }
```

will be available on the recipe as the variables:

```
{upload.port.address} = /dev/ttyACM0
{upload.port.label} = ttyACM0
{upload.port.protocol} = serial
{upload.port.protocolLabel} = Serial Port (USB)
{upload.port.properties.pid} = 0x8043
{upload.port.properties.vid} = 0x2341
{upload.port.properties.serialNumber} = EBEABFD6514D32364E202020FF10181E
{upload.port.properties.name} = ttyACM0
{serial.port} = /dev/ttyACM0                # for backward compatibility
{serial.port.file} = ttyACM0                # only because protocol=serial
```

Here another example:

```
    {
      "eventType": "add",
      "port": {
        "address": "192.168.1.232",
        "label": "SSH on my-board (192.168.1.232)",
        "protocol": "ssh",
        "protocolLabel": "SSH Network port",
        "properties": {
          "macprefix": "AA:BB:CC",
          "macaddress": "AA:BB:CC:DD:EE:FF"
        }
      }
    }
```

that is translated to:

```
{upload.port.address} = 192.168.1.232
{upload.port.label} = SSH on my-board (192.168.1.232)
{upload.port.protocol} = ssh
{upload.port.protocolLabel} = SSH Network port
{upload.port.properties.macprefix} = AA:BB:CC
{upload.port.properties.macaddress} = AA:BB:CC:DD:EE:FF
{serial.port} = 192.168.1.232                 # for backward compatibility
```

This configuration, together with protocol selection, allows to remove the hardcoded `network_pattern`. Now we can replace the legacy recipe (split into multiple lines for clarity):

```
tools.bossac.upload.network_pattern="{runtime.tools.arduinoOTA.path}/bin/arduinoOTA"
                                -address {serial.port} -port 65280
                                -sketch "{build.path}/{build.project_name}.bin"
```

with:

```
tools.arduino_ota.upload.pattern="{runtime.tools.arduinoOTA.path}/bin/arduinoOTA"
                            -address {upload.port.address} -port 65280
                            -sketch "{build.path}/{build.project_name}.bin"
```

User provided fields

Some upload recipes might require custom fields that must be provided by the user, like username and password to upload over the network. In this case the recipe must use the special placeholder **{upload.field.FIELD_NAME}**, where **FIELD_NAME** must be declared separately in the recipe using the following format:

```
tools.UPLOAD_RECIPE_ID.upload.field.FIELD_NAME=FIELD_LABEL
tools.UPLOAD_RECIPE_ID.upload.field.FIELD_NAME.secret=true
```

FIELD_LABEL is the label shown in the graphical prompt where the user is asked to enter the value for the field.

The optional **secret** property should be set to `true` if the field is a secret (like a password or token).

Let's see a complete example:

```
tools.arduino_ota.upload.field.username=Username
tools.arduino_ota.upload.field.password=Password
tools.arduino_ota.upload.field.password.secret=true
tools.arduino_ota.upload.pattern="{runtime.tools.arduinoOTA.path}/bin/arduinoOTA" -address {upload.port.address} -port 65280
-username "{upload.field.username} -password "{upload.field.password}" -sketch "{build.path}/{build.project_name}.bin"
```

If a **FIELD_LABEL** is longer than 50 characters it will be truncated to 49 characters and an ellipsis (`…`) appended to it. For example this field:

```
tools.arduino_ota.upload.field.some_field=This is a really long label that ideally must never be set by any platform
```

will be shown to the user as:

```
This is a really long label that ideally must nev…
```

Upload verification

Upload verification can be enabled via the Arduino IDE's **File > Preferences > Verify code after upload** or `arduino-cli upload --verify` . This uses a system similar to the verbose parameter.

tools.TOOL_ID.ACTION.params.verify defines the value of the **ACTION.verify** property when verification is enabled and **tools.TOOL_ID.ACTION.params.noverify** the value when verification is disabled.

The **{ACTION.verify}** property is only defined for the `upload` and `program` actions of `upload.tool` .

Prior to Arduino IDE 1.6.9, **tools.TOOL_ID.ACTION.params.verify/noverify** were not supported and `{upload.verify}` was set to `true` / `false` according to the verification preference setting, while `{program.verify}` was left undefined. For this reason, backwards compatibility with older IDE versions requires the addition of definitions for the **upload.verify** and **program.verify** properties to platform.txt:

```
[.....]
tools.avrdude.upload.verify=
[.....]
tools.avrdude.program.verify=
[.....]
```

These definitions are overridden with the value defined by **tools.TOOL_ID.ACTION.params.verify/noverify** when a modern version of Arduino development software is in use.

1200 bps bootloader reset

Some Arduino boards use a dedicated USB-to-serial chip, that takes care of restarting the main MCU (starting the bootloader) when the serial port is opened. However, boards that have a native USB connection (such as the Leonardo or Zero) will have to disconnect from USB when rebooting into the bootloader (after which the bootloader reconnects to USB and offers a new serial port for uploading). After the upload is complete, the bootloader disconnects from USB again, starts the sketch, which then reconnects to USB. Because of these reconnections, the standard restart-on-serial open will not work, since that would cause the serial port to disappear and be closed again. Instead, the sketch running on these boards interprets a bitrate of 1200 bps as a signal the bootloader should be started.

To let the Arduino development software perform these steps, two board properties can be set to `true` :

- `use_1200bps_touch` causes the selected serial port to be briefly opened at 1200 bps (8N1) before starting the upload.
- `wait_for_upload_port` causes the upload procedure to wait for the serial port to (re)appear before and after the upload. This is only used when `use_1200bps_touch` is also set. When set, after doing the 1200 bps touch, the development software will wait for a new serial port to appear and use that as the port for uploads. Alternatively, if the original port does not disappear within a few seconds, the upload continues with the original port (which can be the case if the board was already put into bootloader manually, or the the disconnect and reconnect was missed). Additionally, after the upload is complete, the IDE again waits for a new port to appear (or the originally selected port to be present).

Note that the IDE implementation of this 1200 bps touch has some peculiarities, and the newer `arduino-cli` implementation also seems different (does not wait for the port after the reset, which is probably only needed in the IDE to prevent opening the wrong port on the serial monitor, and does not have a shorter timeout when the port never disappears).

Upload Using Programmer by default

If the **upload.protocol** property is not defined for a board, the Arduino IDE's "Upload" process will use the same behavior as "Upload Using Programmer". This is convenient for boards which only support uploading via programmer.

Serial port

The full path (e.g., `/dev/ttyACM0`) of the port selected via the IDE or `arduino-cli upload`'s `--port` option is available as a configuration property **{upload.port.address}**.

The file component of the port's path (e.g., `ttyACM0`) is available as the configuration property **{upload.port.label}**.

For backward compatibility with IDE 1.8.15 and older the old property **serial.port** is still available and is identical to **{upload.port.address}**. Instead **serial.port.file** is identical to **{upload.port.label}** and available only if protocol in use is **serial**.

Upload using an external programmer

The `program` action is triggered via the **Sketch > Upload Using Programmer** feature of the IDEs or `arduino-cli upload --programmer <programmer ID>`. This action is used to transfer a compiled sketch to a board using an external programmer.

The **program.tool** property determines the tool to be used for this action. This property is typically defined for each programmer in programmers.txt and uses the same syntax as the `upload` action:

```
[......]
usbasp.program.tool.serial=avrdude
[......]
arduinoasisp.program.tool.serial=avrdude
[......]
arduinoisp.program.tool.default=avrdude
[......]
```

For backward compatibility with IDE 1.8.15 and older the previous syntax is still supported:

```
[......]
usbasp.program.tool=avrdude
[......]
arduinoasisp.program.tool=avrdude
[......]
```

This action can use the same upload verification preference system as the `upload` action, via the **program.verify** property.

When using the Arduino IDE, if the selected programmer is from a different platform than the board, the `program` recipe defined in the programmer's platform is used without overrides from the properties defined in the platform.txt of the board platform. When using Arduino development software other than the Arduino IDE, the handling of properties is the same as when doing a standard Upload.

Burn Bootloader

The `erase` and `bootloader` actions are triggered via the **Tools > Burn Bootloader** feature of the Arduino IDE or `arduino-cli burn-bootloader`. This action is used to flash a bootloader to the board.

"Burn Bootloader" is unique in that it uses two actions, which are executed in sequence:

1. `erase` is typically used to erase the microcontroller's flash memory and set the configuration fuses according to the properties defined in the board definition
2. `bootloader` is used to flash the bootloader to the board

The **bootloader.tool** property determines the tool to be used for the `erase` and `bootloader` actions both. This property is typically defined for each board in boards.txt and uses the same syntax as the `upload` action:

```
[......]
uno.bootloader.tool.serial=avrdude
[......]
leonardo.bootloader.tool.serial=avrdude
leonardo.bootloader.tool.network=arduino_ota
[......]
duemilanove.bootloader.tool.default=avrdude
[......]
```

For backward compatibility with IDE 1.8.15 and older the previous syntax is still supported:

```
[......]
uno.bootloader.tool=avrdude
[......]
leonardo.bootloader.tool=avrdude
[......]
```

When using the Arduino IDE, if the board uses a core reference, the platform.txt of the core platform is not used at all in defining the recipes for `erase` and `bootloader` actions. When using Arduino development software other than the Arduino IDE, the handling of properties from the core platform's platform.txt is done as usual.

Sketch debugging configuration

Starting from Arduino CLI 0.9.0 / Arduino IDE 2.x, sketch debugging support is available for platforms.

The debug action is triggered when the user clicks the Debug button in the Arduino IDE or runs the `arduino-cli debug` command.

Since opening a debug session requires the orchestration of numerous tools, the CLI/IDE will take care of that duty: differently from the upload actions, there is no need for the platform to provide debug recipes, the only requirement is to provide some debug configuration directives.

Debugger configuration directives

All the debug directives are grouped under the `debug.*` directives. Here is the complete list of the supported directives:

- `debug.executable`: is the absolute path to the compiled binary of the sketch
- `debug.toolchain`: is a unique identifier of the required toolchain, currently we support `gcc` (and compatible) only
- `debug.toolchain.path`: is the absolute path to the toolchain directory
- `debug.toolchain.prefix`: is the prefix of the toolchain (for example `arm-none-eabi-`)
- `debug.server`: is a unique identifier of the required debug server, currently we support only `openocd`
- `debug.svd_file`: is the absolute path to the SVD descriptor.

If the `debug.executable` property is not present or is empty debugging will not be allowed.

OpenOCD server specific configurations:

- `debug.server.openocd.path`: is the absolute path to the OpenOCD directory
- `debug.server.openocd.scripts_dir`: is the absolute path to the OpenOCD scripts directory
- `debug.server.openocd.scripts.N`: is a list of OpenOCD script files to run, where N is a number (a sequence of non-consecutive numbers is allowed)
- `debug.server.openocd.script`: if there is only one OpenOCD script to run, this directive con be used instead of the `debug.server.openocd.scripts.N` (this directive is ignored if `debug.server.openocd.scripts.N` is present)

Custom config for Cortext-debug plugin for Arduino IDE

The Arduino IDE uses cortex-debug plugin to start a debugging session. The IDE creates a `launch.json` file that is needed to start the debugging via the cortex-debug plugin. To give the platform developers more flexibility, it is allowed to pass any extra arbitrary setup to `launch.json` generated by the IDE. To allow this the directives under the group `debug.cortex-debug.custom.*` are converted into JSON and added to the generated `launch.json` as-is. Moreover, if a directive has a key with a numeric suffix, it is converted into a JSON array.

For example the following directives:

```
debug.cortex-debug.custom.postAttachCommands.0=set remote hardware-watchpoint-limit 2
debug.cortex-debug.custom.postAttachCommands.1=monitor reset halt
debug.cortex-debug.custom.postAttachCommands.2=monitor gdb_sync
debug.cortex-debug.custom.postAttachCommands.3=thb setup
debug.cortex-debug.custom.postAttachCommands.4=c
debug.cortex-debug.custom.overrideRestartCommands.0=monitor reset halt
debug.cortex-debug.custom.overrideRestartCommands.1=monitor gdb_sync
```

```
debug.cortex-debug.custom.overrideRestartCommands.2=thb setup
debug.cortex-debug.custom.overrideRestartCommands.3=c
```

will result in the following JSON to be merged in the Arduino IDE generated `launch.json`:

```
{
  "overrideRestartCommands": ["monitor reset halt", "monitor gdb_sync", "thb setup", "c"],
  "postAttachCommands": [
    "set remote hardware-watchpoint-limit 2",
    "monitor reset halt",
    "monitor gdb_sync",
    "thb setup",
    "c"
  ]
}
```

All the values are converted by default to a string in the resulting JSON. If another type is needed the value can be prefixed with the tags `[boolean]`, `[number]`, `[string]` or `[object]` to force a specific type in the JSON. Moreover the hierarchy of the properties may be used to build JSON objects. For example:

```
debug.cortex-debug.custom.aBoolean=[boolean]true
debug.cortex-debug.custom.aNumber=[number]10
debug.cortex-debug.custom.anotherNumber=[number]10.20
debug.cortex-debug.custom.anObject=[object]{"key":"value", "boolean":true}
debug.cortex-debug.custom.anotherObject.key=value
debug.cortex-debug.custom.anotherObject.boolean=[boolean]true
```

will result in the following JSON:

```
{
  "aBoolean": true,
  "aNumber": 10,
  "anotherNumber": 10.2,
  "anObject": {
    "boolean": true,
    "key": "value"
  },
  "anotherObject": {
    "boolean": true,
    "key": "value"
  }
}
```

Additional debugger config selection via `debug.additional_config` directive.

It is possible to use any sub-tree of the platform configuration to override the debugger configuration using the directive `debug.additional_config=CONFIG_PREFIX`. This rule will use the configuration under `CONFIG_PREFIX.*` to override the current `debug.*` config.

This change allows a more convenient rationalization and selection of the configs to apply to the debugger. For example, we could factor common parts of a configuration in the platform.txt file:

```
# CONFIG 1
debug-overrides.esp32.cortex-debug.custom.name=Arduino on ESP32
debug-overrides.esp32.cortex-debug.custom.request=attach
debug-overrides.esp32.cortex-debug.custom.postAttachCommands.0=set remote hardware-watchpoint-limit 2
debug-overrides.esp32.cortex-debug.custom.postAttachCommands.1=monitor reset halt
debug-overrides.esp32.cortex-debug.custom.postAttachCommands.2=monitor gdb_sync
debug-overrides.esp32.cortex-debug.custom.postAttachCommands.3=thb setup
debug-overrides.esp32.cortex-debug.custom.postAttachCommands.4=c
debug-overrides.esp32.cortex-debug.custom.overrideRestartCommands.0=monitor reset halt
debug-overrides.esp32.cortex-debug.custom.overrideRestartCommands.1=monitor gdb_sync
debug-overrides.esp32.cortex-debug.custom.overrideRestartCommands.2=thb setup
debug-overrides.esp32.cortex-debug.custom.overrideRestartCommands.3=c

# CONFIG 2
debug-overrides.esp32s2.cortex-debug.custom.name=Arduino on ESP32-S2
debug-overrides.esp32s2.cortex-debug.custom.request=attach
debug-overrides.esp32s2.cortex-debug.custom.postAttachCommands.0=set remote hardware-watchpoint-limit 2
debug-overrides.esp32s2.cortex-debug.custom.postAttachCommands.1=monitor reset halt
debug-overrides.esp32s2.cortex-debug.custom.postAttachCommands.2=monitor gdb_sync
debug-overrides.esp32s2.cortex-debug.custom.postAttachCommands.3=thb setup
debug-overrides.esp32s2.cortex-debug.custom.postAttachCommands.4=c
debug-overrides.esp32s2.cortex-debug.custom.overrideRestartCommands.0=monitor reset halt
debug-overrides.esp32s2.cortex-debug.custom.overrideRestartCommands.1=monitor gdb_sync
debug-overrides.esp32s2.cortex-debug.custom.overrideRestartCommands.2=thb setup
debug-overrides.esp32s2.cortex-debug.custom.overrideRestartCommands.3=c
```

and choose which one to use depending on the board in the boards.txt file:

```
myboard.name=My Board with esp32
myboard.debug.additional_config=debug-overrides.esp32

anotherboard.name=My Board with esp32s2
anotherboard.debug.additional_config=debug-overrides.esp32s2
...
```

Another possibility is to compose the configuration using another variable present in the board configuration, for example if in the `platform.txt` we add:

```
debug.additional_config=debug-overrides.{build.mcu}
```

we may use the `build.mcu` value as a "selector" for the board-specific debug configuration that is overlapped to the global debug configuration:

```
myboard.name=My Board with esp32
myboard.build.mcu=esp32
...

anotherboard.name=My Board with esp32s2
anotherboard.build.mcu=esp32s2
...
```

Optimization level for debugging

The compiler optimization level that is appropriate for normal usage will often not provide a good experience while debugging. For this reason, it may be helpful to use different compiler flags when compiling a sketch for use with the debugger. The flags for use when compiling for debugging can be defined via the **compiler.optimization_flags.debug** property, and those for normal use via the **compiler.optimization_flags.release** property. The **compiler.optimization_flags** property will be defined according to one or the other depending on the Arduino Pro IDE's **Sketch > Optimize for Debugging** setting or `arduino-cli compile`'s `--optimize-for-debug` option.

Custom board options

It can sometimes be useful to provide user selectable configuration options for a specific board. For example, a board could be provided in two or more variants with different microcontrollers, or may have different crystal speed based on the board model, and so on...

When using Arduino CLI, the option can be selected via the FQBN, or using the `--board-options` flag

In the Arduino IDE the options add extra menu items under the "Tools" menu.

In Arduino Web Editor, the options are displayed in the "Flavours" menu.

Let's see an example of how a custom option is implemented. The board used in the example is the Arduino Duemilanove. This board was produced in two models, one with an ATmega168 microcontroller and another with an ATmega328P.
We are going then to define a custom option, using the "cpu" MENU_ID, that allows the user to choose between the two different microcontrollers.

We must first define a set of **menu.MENU_ID=Text** properties. **Text** is what is displayed on the GUI for every custom menu we are going to create and must be declared at the beginning of the boards.txt file:

```
menu.cpu=Processor
[.....]
```

in this case, the menu name is "Processor".
Now let's add, always in the boards.txt file, the default configuration (common to all processors) for the duemilanove board:

```
menu.cpu=Processor
[.....]
duemilanove.name=Arduino Duemilanove
duemilanove.upload.tool=avrdude
duemilanove.upload.protocol=arduino
duemilanove.build.f_cpu=16000000L
duemilanove.build.board=AVR_DUEMILANOVE
duemilanove.build.core=arduino
duemilanove.build.variant=standard
[.....]
```

Now let's define the possible values of the "cpu" option:

```
[.....]
duemilanove.menu.cpu.atmega328=ATmega328P
[.....]
duemilanove.menu.cpu.atmega168=ATmega168
[.....]
```

We have defined two values: "atmega328" and "atmega168".
Note that the property keys must follow the format **BOARD_ID.menu.MENU_ID.OPTION_ID=Text**, where **Text** is what is displayed under the "Processor" menu in the IDE's GUI.
Finally, the specific configuration for each option value:

```
[.....]
## Arduino Duemilanove w/ ATmega328P
duemilanove.menu.cpu.atmega328=ATmega328P
duemilanove.menu.cpu.atmega328.upload.maximum_size=30720
duemilanove.menu.cpu.atmega328.upload.speed=57600
duemilanove.menu.cpu.atmega328.build.mcu=atmega328p

## Arduino Duemilanove w/ ATmega168
duemilanove.menu.cpu.atmega168=ATmega168
duemilanove.menu.cpu.atmega168.upload.maximum_size=14336
duemilanove.menu.cpu.atmega168.upload.speed=19200
duemilanove.menu.cpu.atmega168.build.mcu=atmega168
[.....]
```

Note that when the user selects an option value, all the "sub properties" of that value are copied in the global configuration. For example, when the user selects "ATmega168" from the "Processor" menu, or uses the FQBN `arduino:avr:duemilanove:cpu=atmega168` with Arduino CLI, the configuration under atmega168 is made available globally:

```
duemilanove.menu.cpu.atmega168.upload.maximum_size     =>   upload.maximum_size
duemilanove.menu.cpu.atmega168.upload.speed            =>   upload.speed
duemilanove.menu.cpu.atmega168.build.mcu               =>   build.mcu
```

There is no limit to the number of custom menus that can be defined.

Referencing another core, variant or tool

The Arduino platform referencing system allows using components of other platforms in cases where it would otherwise be necessary to duplicate those components. This feature allows us to reduce the minimum set of files needed to define a new "hardware" to just the boards.txt file.

Core reference

Inside the boards.txt we can define a board that uses a core provided by another vendor/maintainer using the syntax **VENDOR_ID:CORE_ID**. For example, if we want to define a board that uses the "arduino" core from the "arduino" vendor we should write:

```
[....]
myboard.name=My Wonderful Arduino Compatible board
myboard.build.core=arduino:arduino
[....]
```

Note that we don't need to specify any architecture since the same architecture of "myboard" is used, so we just say "arduino:arduino" instead of "arduino:avr:arduino".

The platform.txt settings are inherited from the referenced core platform, thus there is no need to provide a platform.txt unless there are some specific properties that need to be overridden.

The bundled libraries from the referenced platform are used, thus there is no need for the referencing platform to bundle those libraries. If libraries are provided, the list of available libraries is the sum of the two libraries, where the referencing platform has priority over the referenced platform.

The programmers from the referenced platform are made available, thus there is no need for the referencing platform to define those programmers. If the referencing platform does provide its own programmer definitions, the list of available programmer is the sum of the programmers of the two platforms. In Arduino IDE 1.8.12 and older, all programmers of all installed platforms were made available.

Variant reference

In the same way we can use a variant defined on another platform using the syntax **VENDOR_ID:VARIANT_ID**:

```
[....]
myboard.build.variant=arduino:standard
[....]
```

Note that, unlike core references, other resources (platform.txt, bundled libraries, programmers) are *not* inherited from the referenced platform.

Tool references

Tool recipes defined in the platform.txt of other platforms can also be referenced using the syntax **VENDOR_ID:TOOL_ID**:

```
[....]
myboard.upload.tool=arduino:avrdude
myboard.bootloader.tool=arduino:avrdude
[....]
```

When using Arduino CLI or Arduino IDE 2.x (but not Arduino IDE 1.x), properties used in the referenced tool recipe may be overridden in the referencing platform's platform.txt.

Note that, unlike core references, referencing a tool recipe does *not* result in any other resources being inherited from the referenced platform.

Platform Terminology

Because boards can reference cores, variants and tools in different platforms, this means that a single build or upload can use data from up to four different platforms. To keep this clear, the following terminology is used:

- The "board platform" is the platform that defines the currently selected board (e.g. the platform that contains the board.txt the board is defined in.
- The "core platform" is the the platform that contains the core to be used.
- The "variant platform" is the platform that contains the variant to be used.
- The "tool platform" is the platform that contains the tool used for the current operation.

In the most common case: a board platform without any references, all of these will refer to the same platform.

Note that the above terminology is not in widespread use, but was invented for clarity within this document. In the actual Arduino CLI code, the "board platform" is called `targetPlatform`, the "core platform" is called `actualPlatform`, the others are pretty much nameless.

boards.local.txt

Introduced in Arduino IDE 1.6.6. This file can be used to override properties defined in `boards.txt` or define new properties without modifying `boards.txt`. It must be placed in the same folder as the `boards.txt` it supplements.

Platform bundled libraries

Arduino libraries placed in the platform's `libraries` subfolder are accessible when a board of the platform, or of a platform that references the platform's core, is selected. When any other board is selected, the platform bundled libraries are inaccessible.

These are often architecture-specific libraries (e.g., SPI, Wire) which must be implemented differently for each architecture.

Platform bundled libraries may be used to provide specialized versions of libraries which use the dependency resolution system to override built-in libraries.

For more information, see the Arduino library specification.

keywords.txt

As of Arduino IDE 1.6.6, per-platform keywords can be defined by adding a keywords.txt file to the platform's architecture folder. These keywords are only highlighted in the Arduino IDE when one of the boards of that platform are selected. This file follows the same format as the keywords.txt used in libraries.

Post-install script

After Boards Manager finishes installation of a platform, it checks for the presence of a script named:

- `post_install.bat` - when running on Windows
- `post_install.sh` - when running on any non-Windows operating system

If present, the script is executed.

This script may be used to configure the user's system for the platform, such as installing drivers.

The circumstances under which the post-install script will run are different depending on which Arduino development software is in use:

- **Arduino IDE 1.x**: runs the script when the installed platform is signed with Arduino's private key.
- **Arduino IDE 2.x**: runs the script for any installed platform.
- **Arduino CLI**: (since 0.12.0) runs the script for any installed platform when Arduino CLI is in "interactive" mode. This behavior can be configured

Pre-uninstall script

Before Boards Manager starts uninstalling a platform, it checks for the presence of a script named:

- `pre_uninstall.bat` - when running on Windows