Kaleidoscope

Keyboardio & Friends

May 06, 2024

INSTALLATION AND SETUP

Flexible firmware for computer keyboards.

This package contains the "core" of Kaleidoscope and a number of example firmware "Sketches"

If you're just getting started with the Keyboardio Model 01, the introductory docs are here and the source for the basic firmware package is here: https://github.com/keyboardio/Model01-Firmware. It's probably a good idea to start there, learn how to modify your keymap and maybe turn some modules on or off, and then come back to the full repository when you have more complex changes in mind.

CHAPTER

ONE

GETTING STARTED

For most folks, the right way to get started is to install the Arduino IDE with prebuilt Kaleidoscope support

1.1 Set up the Arduino IDE with Kaleidoscope support

Install Arduino and Kaleidoscope

The first thing you should do once you install Kaleidoscope is to *try building your keyboard's firmware*.

CHAPTER TWO

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

Arduino is one of the world's most widely used (and user friendly) platforms for programming "embedded" devices like the chip inside your keyboard.

To customize your keyboard's layout or functionality, the most robust and flexible option is to use the Arduino IDE.

If you're planning to modify Kaleidoscope itself or plan on developing Kaleidoscope plugins, you should be checking out the source code from our git repository instead. You can find instructions for that at https://github.com/keyboardio/Kaleidoscope

CHAPTER

THREE

SET UP THE ARDUINO IDE

Arduino's designers made it to be accessible to people at all skill levels, and Kaleidoscope is built on top of the Arduino platform because we share that goal. The easiest way to customize your keyboard's firmware is to use the Arduino IDE. Even if you expect to use the command line to compile your firmware, you'll still need to install Arduino, as they provide the compilers and libraries Kaleidoscope needs to work.

Using the IDE is is the easiest process for folks who are new to Arduino, or to programming generally. If you follow the instructions below step by step you should be fine. :-)

The right way to install Arduino is a little bit different depending on what operating system you use.

- *Install Arduino on macOS*
- *Install Arduino on Linux*
- *Install Arduino on Windows 10*
- *Install Arduino on FreeBSD*

3.1 Install Arduino on macOS

1. Download the Arduino IDE install package from https://www.arduino.cc/en/Main/Software

As of this writing, the latest version is v1.8.13, which you can download from https://www.arduino.cc/download_handler.php?f=/arduino-1.8.13-macosx.zip

1. Double-click "arduino-1.8.13-macos.zip" to uncompress the Arduino IDE.
2. Move Arduino.app from your `Downloads` folder to your Applications folder.
3. Double-click on Arduino.app to start it.

Next step: *Add keyboard support to Arduino*

3.2 Install Arduino on Linux

1. Install version 1.8.13 or newer of the Arduino IDE from:

```
Tar archive: http://arduino.cc/download
Flatpak:     https://flathub.org/apps/details/cc.arduino.arduinoide
Snap:        https://snapcraft.io/arduino
Arch:        sudo pacman -S arduino
```

Unfortunately, the version of the Arduino IDE packaged in Ubuntu is unmaintained and too old to use, and the version packaged in Debian has been heavily modified and might not be able to compile your keyboard's firmware.

2. Assuming you're using the tar archive, and untarring in the download directory:

```
$ cd ~/Downloads
$ tar xvf arduino-1.8.13-linux64.tar.xz
$ sudo mv arduino-1.8.13 /usr/local/arduino
$ cd /usr/local/arduino
$ sudo ./install.sh
```

3. On some linux distributions, ModemManager can prevent you from flashing or updating your keyboard by interfering with its virtual serial port. Additionally, by default, you may not have permissions to access your keyboard's serial port. `udev` is the Linux subsystem that managed both of these things. You should install our udev rules to manage access to your keyboard's serial port.

```
$ wget https://raw.githubusercontent.com/keyboardio/Kaleidoscope/master/etc/60-
↪kaleidoscope.rules
$ sudo cp 60-kaleidoscope.rules /etc/udev/rules.d
$ sudo /etc/init.d/udev reload
```

For Arch based distributions use the following command instead of `sudo /etc/init.d/udev reload`

```
$ sudo udevadm control --reload-rules && udevadm trigger
```

4. Next, disconnect and reconnect your keyboard so that your computer will apply the changes.

3.3 Install Arduino on Windows 10

Note: *This tutorial has been written using Windows 10.*

1. Download the Arduino IDE installation package from https://www.arduino.cc/en/Main/Software

As of this writing, the latest version is v1.8.13, which you can download from this URL:

https://www.arduino.cc/download_handler.php?f=/arduino-1.8.13-windows.exe

Note: *Some users have had difficulties with the Windows store version of the Arduino IDE. Please use the downloadable installation package.*

1. Open the installation package and follow the prompts to install the Arduino IDE.

Next step: *Add keyboard support to Arduino*

3.4 Install Arduino on FreeBSD

1. Install the following packages required by the build system: `bash`, `gmake`, `perl5`, `avrdude`, and `arduino18`.

```
$ sudo pkg install bash gmake perl5 avrdude arduino18
```

3.4.1 Flashing firmware as non-root.

1. If you want to flash your firmware as non-root, ensure your user has write access to the appropriate USB devices in devfs. By default, the devices are owned by `root:operator`, so put yourself in the `operator` group. You will also need to add yourself to the `dialer` group to use the modem device:

   ```
   $ sudo pw groupmod operator -m $USER
   $ sudo pw groupmod dialer -m $USER
   ```

2. Add devfs rules for write access for operator to USB devices:

   ```
   $ cat << EOM >> /etc/devfs.rules
   #
   # Allow operators access to usb devices.
   #
   [operator_usb=5]
   add path usbctl mode 0660 group operator
   add path 'usb/*' mode 0660 group operator
   add path 'ugen*' mode 0660 group operator
   EOM
   ```

3. Update `/etc/rc.conf` to use the new devfs rule as the system rule:

   ```
   $ sudo sysrc devfs_system_ruleset=operator_usb
   ```

4. Restart devfs:

   ```
   $ sudo service devfs restart
   ```

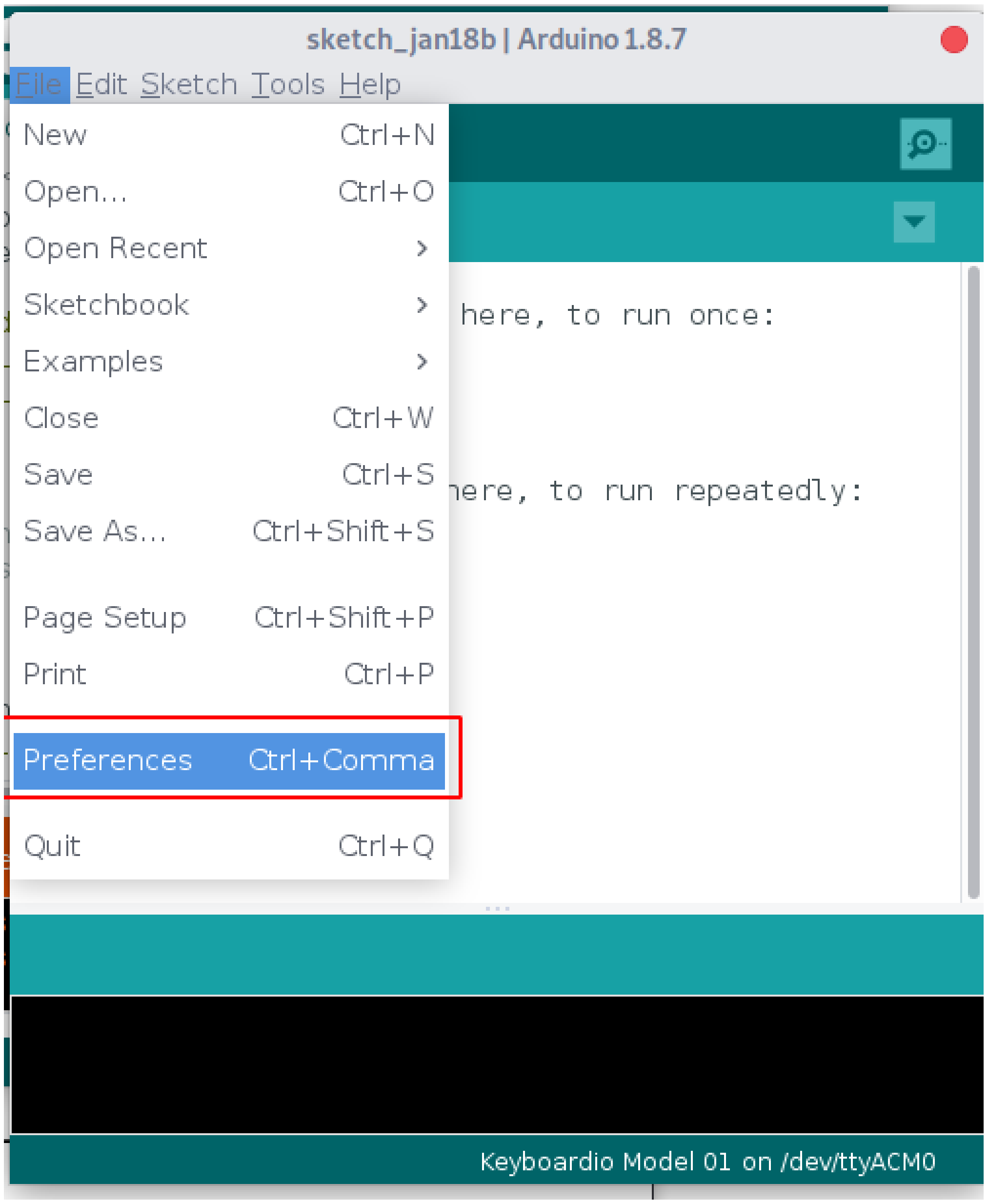
Next step: *Add keyboard support to Arduino*

CHAPTER

FOUR

ADD KEYBOARD SUPPORT TO ARDUINO

1. Open the Arduino IDE. It will open an empty "sketch" window.
2. On Windows or Linux: Open the "File" menu, and click on "Preferences." On a Mac: Open the "Arduino" menu, and click on "Preferences."



```
here, to run once:

here, to run repeatedly:
```

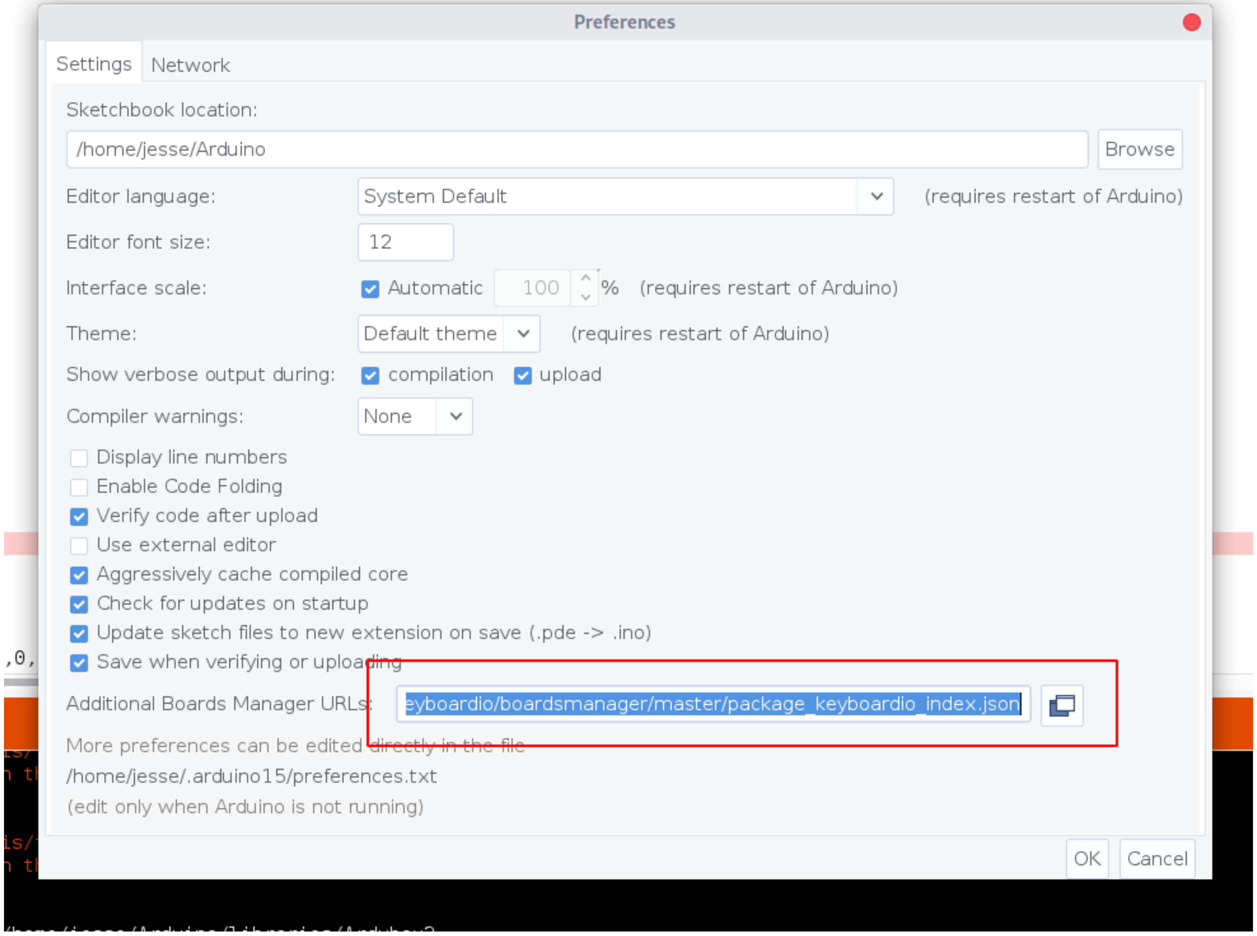
1. To use released versions of Kaleidoscope, paste the following url into the box labeled 'Additional Board Manager URLs':

   ```
   https://raw.githubusercontent.com/keyboardio/boardsmanager/master/package_
   ↪keyboardio_index.json
   ```

   If you would prefer to be able to install an 'up to the minute' build of the `master` branch of Kaleidoscope from git, use this URL:

   ```
   https://raw.githubusercontent.com/keyboardio/arduino-kaleidoscope-master/main/
   ↪package_kaleidoscope_master_index.json
   ```

   As a warning: the `master` builds may be less stable than release builds.


Preferences

Settings | Network

Sketchbook location: /home/jesse/Arduino

Editor language: System Default (requires restart of Arduino)

Editor font size: 12

Interface scale: [✓] Automatic 100 % (requires restart of Arduino)

Theme: Default theme (requires restart of Arduino)

Show verbose output during: [✓] compilation [✓] upload

Compiler warnings: None

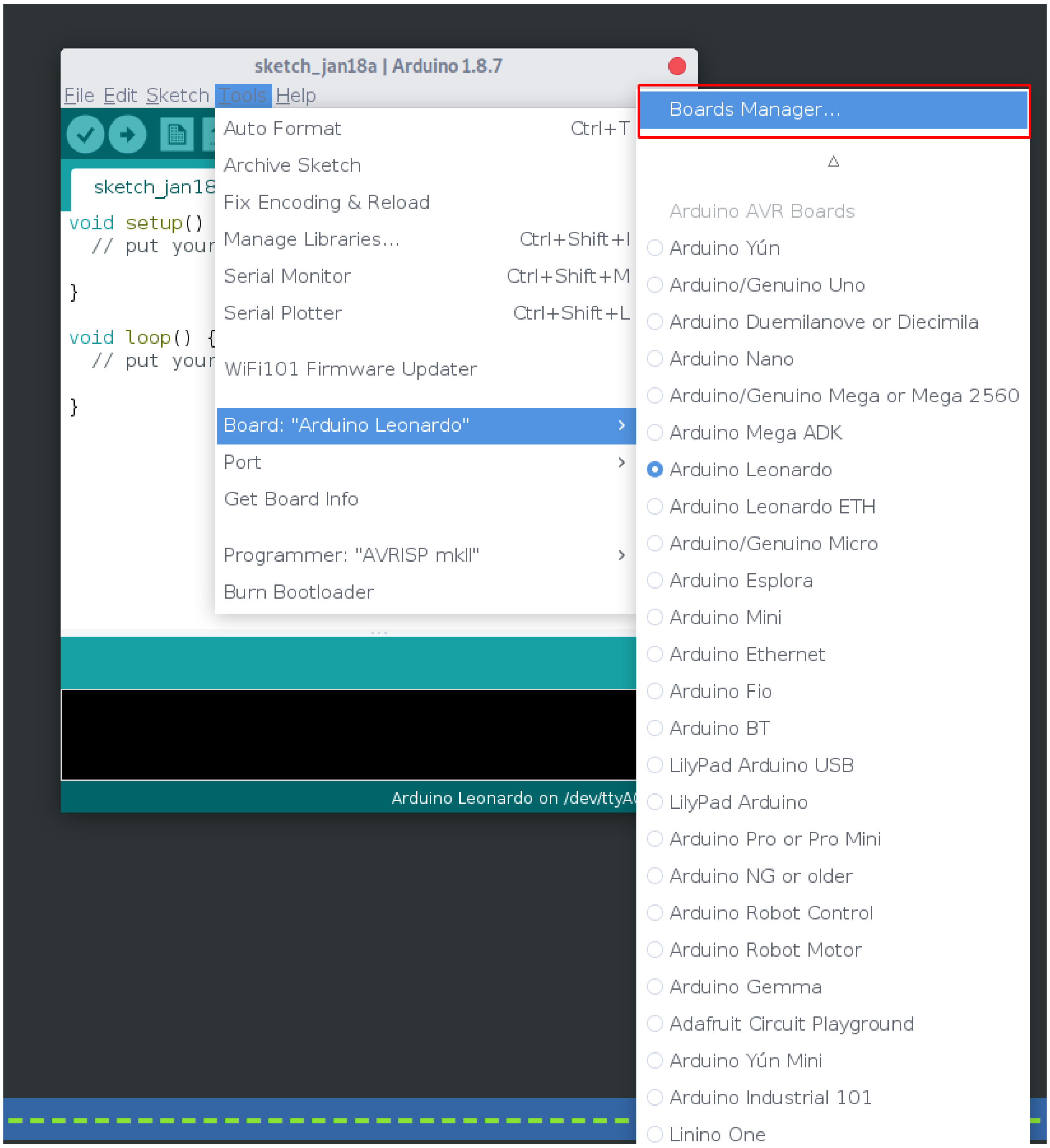
- [ ] Display line numbers
- [ ] Enable Code Folding
- [✓] Verify code after upload
- [ ] Use external editor
- [✓] Aggressively cache compiled core
- [✓] Check for updates on startup
- [✓] Update sketch files to new extension on save (.pde -> .ino)
- [✓] Save when verifying or uploading

Additional Boards Manager URLs: eyboardio/boardsmanager/master/package_keyboardio_index.json

More preferences can be edited directly in the file
/home/jesse/.arduino15/preferences.txt
(edit only when Arduino is not running)

OK Cancel

1. Click 'OK' to close the dialog
2. Open the 'Tools' menu, click on 'Board' and then click on 'Boards Manager'



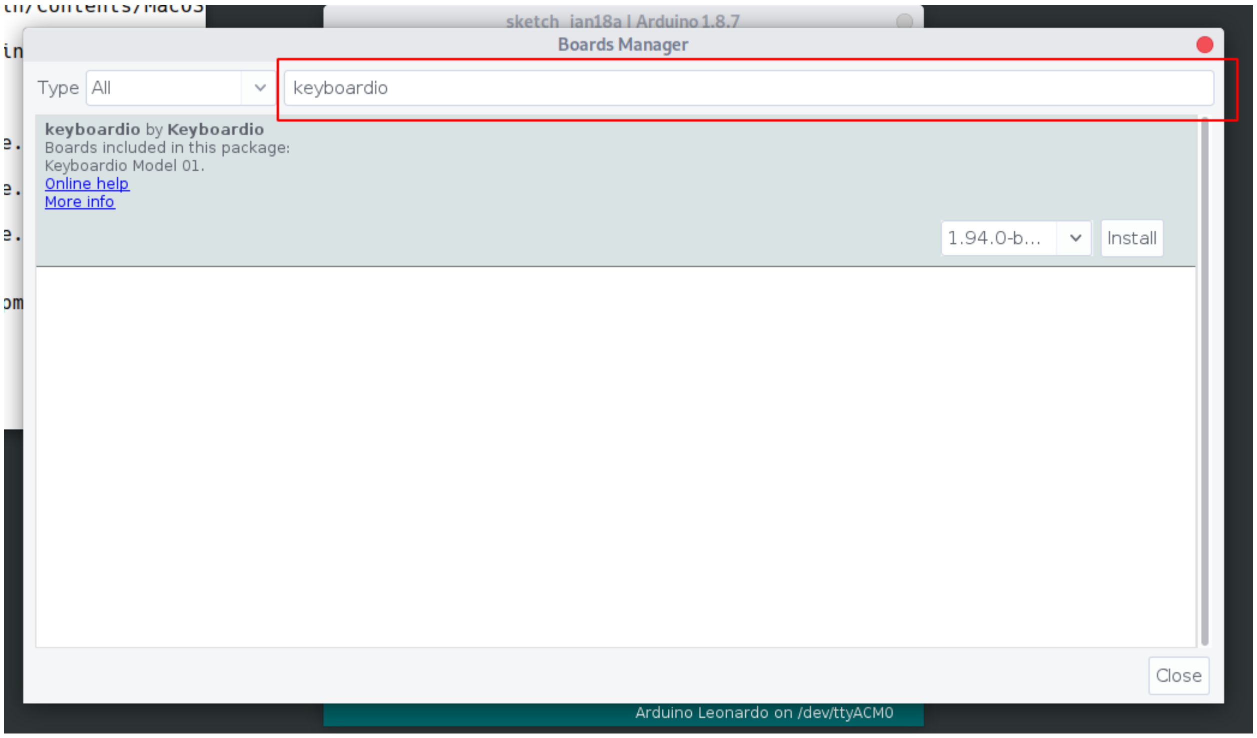
```
void setup()
  // put your

}

void loop() {
  // put your

}
```

Board: "Arduino Leonardo"

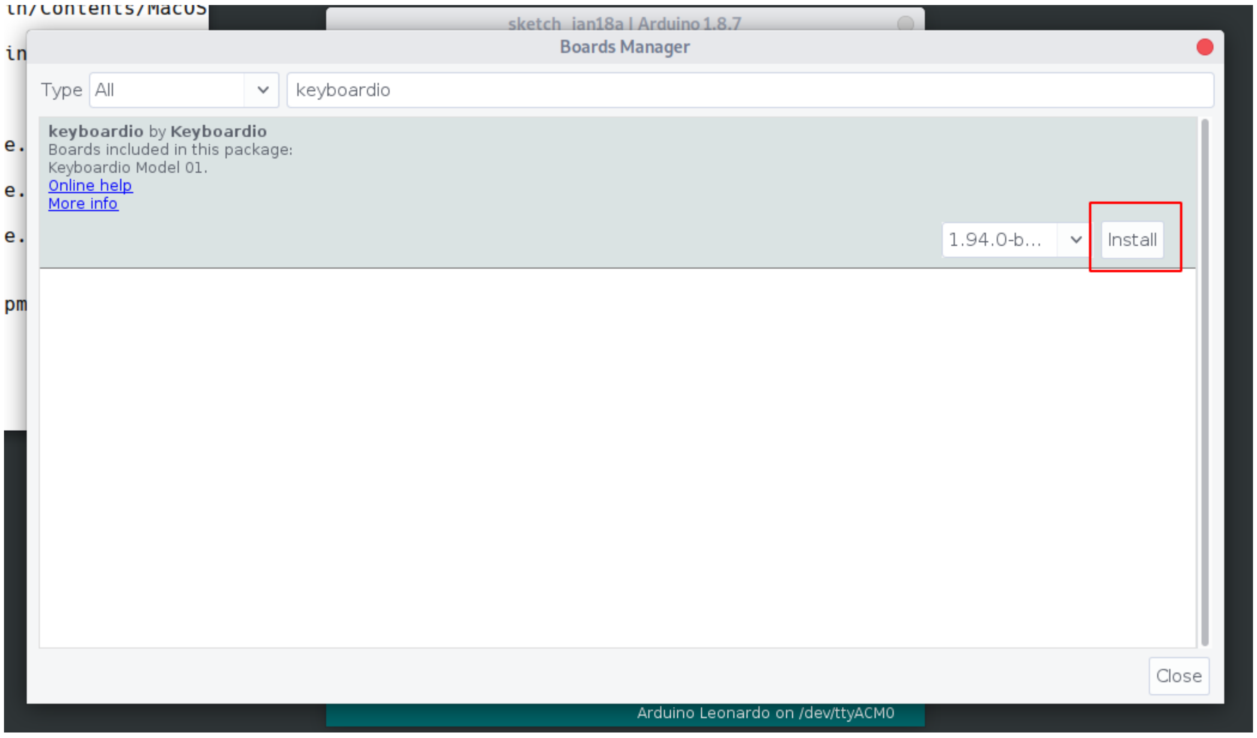
Programmer: "AVRISP mkII"

Arduino Leonardo on /dev/ttyAC

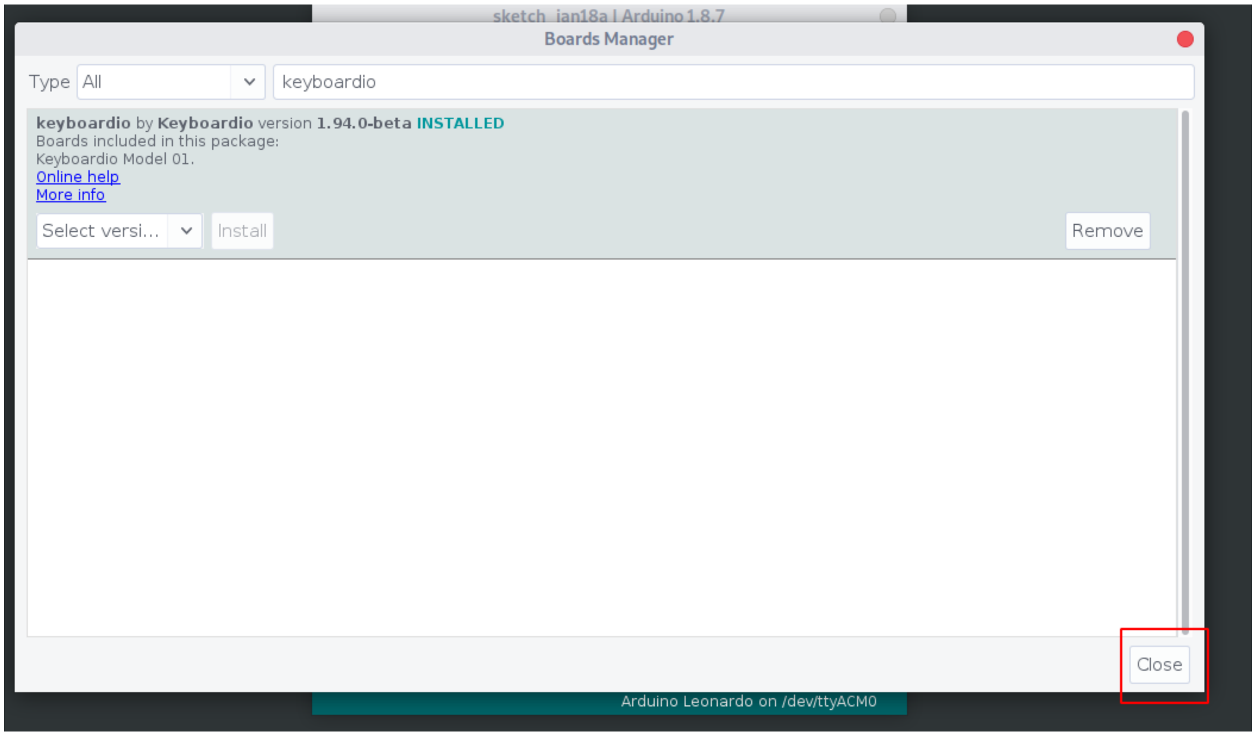
1. Type ‘Keyboardio’ into the search box.



1. You will see an entry that says "keyboardio by Keyboardio" click on it to select it, and then click 'Install'.



1. Once the install completes, click "Close".



keyboardio by Keyboardio version 1.94.0-beta INSTALLED
Boards included in this package:
Keyboardio Model 01.
Online help
More info

Next up, you might want to *build the latest version of your keyboard's firmware*

CHAPTER

FIVE

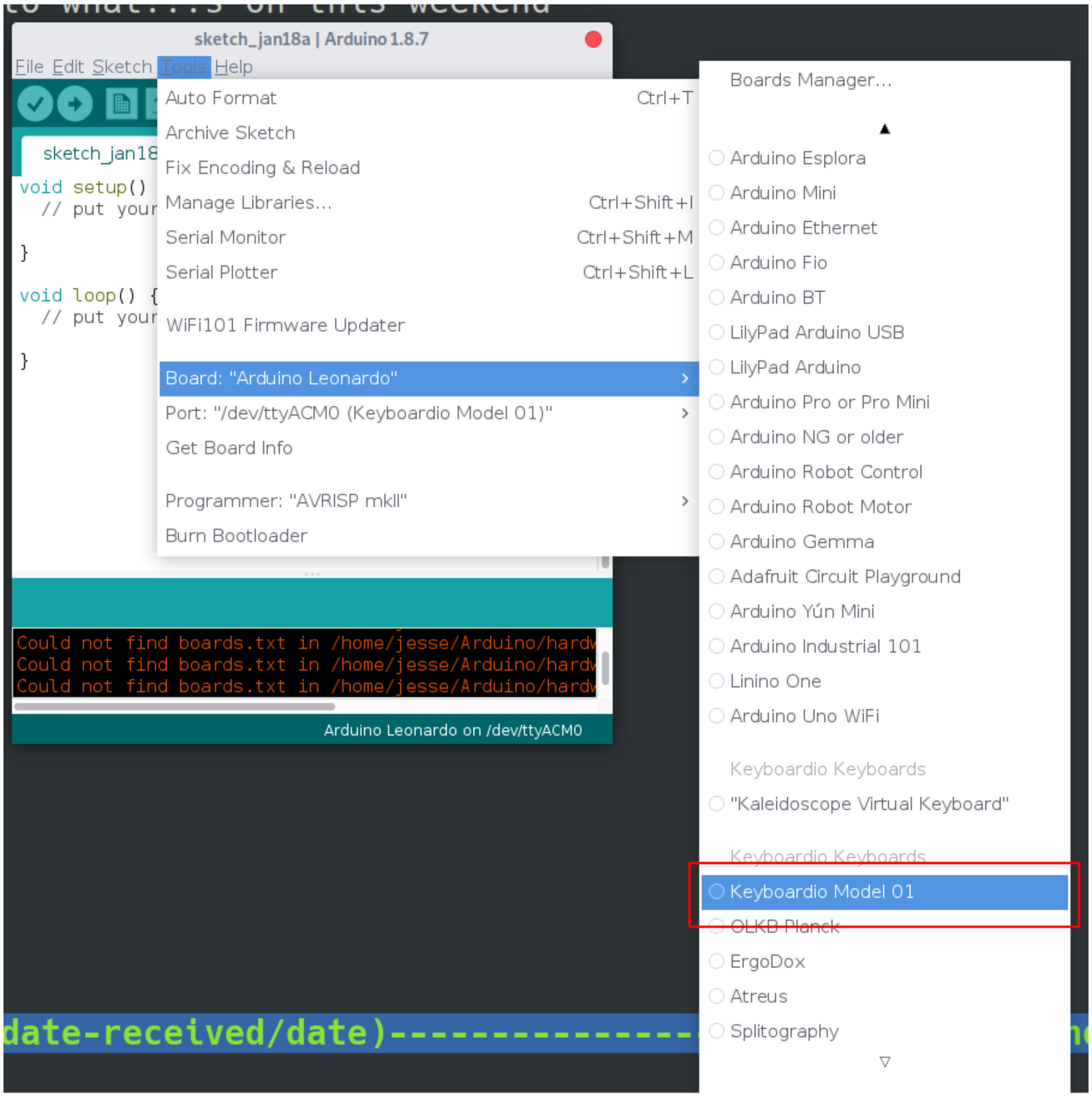
BUILD AND INSTALL THE LATEST FIRMWARE FOR YOUR KEYBOARD

CHAPTER

SIX

SELECT YOUR KEYBOARD

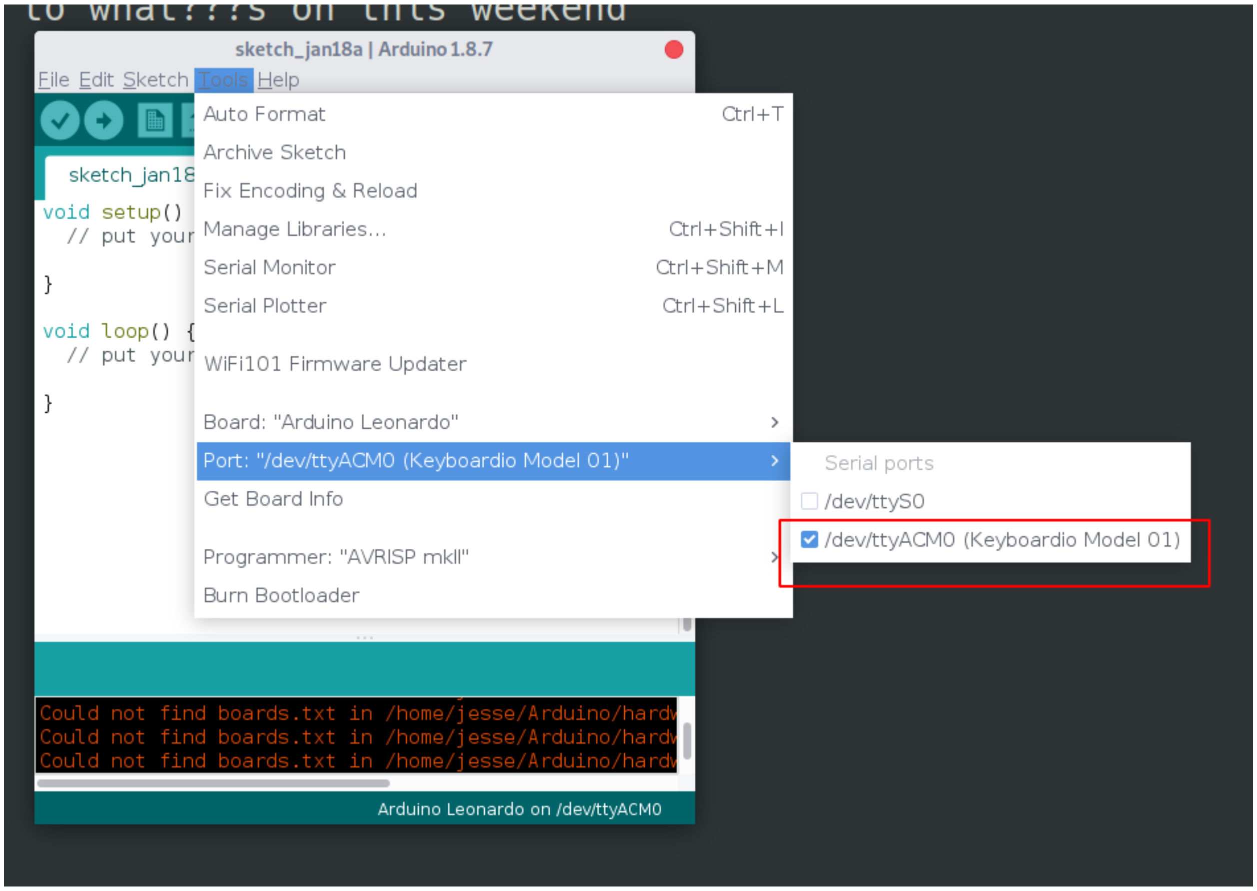
1. Open the ‘Tools’ menu, click on ‘Board’ and then click on the name of your keyboard. In the screenshot, we picked ‘Keyboardio Model 01’. (You may have to scroll through a long list of other boards to get there.)



```
sketch_jan18
void setup()
  // put your
}

void loop() {
  // put your

}
```

```
Could not find boards.txt in /home/jesse/Arduino/hardw
Could not find boards.txt in /home/jesse/Arduino/hardw
Could not find boards.txt in /home/jesse/Arduino/hardw
```

1. Open the ‘Tools’ menu, click on “Port > “. If your keyboard is not already selected, click on it to select it. (If there is only one option such as “COM3” try it, it’s almost certainly the correct port.)


sketch_jan18a | Arduino 1.8.7
File Edit Sketch Tools Help
sketch_jan18
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 Firmware Updater
Board: "Arduino Leonardo"
Port: "/dev/ttyACM0 (Keyboardio Model 01)"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
/dev/ttyS0
/dev/ttyACM0 (Keyboardio Model 01)
Arduino Leonardo on /dev/ttyACM0

```
void setup()
  // put your

}

void loop() {
  // put your

}
```

```
Could not find boards.txt in /home/jesse/Arduino/hardw
Could not find boards.txt in /home/jesse/Arduino/hardw
Could not find boards.txt in /home/jesse/Arduino/hardw
```

Next step: *Install the latest firmware on your keyboard*

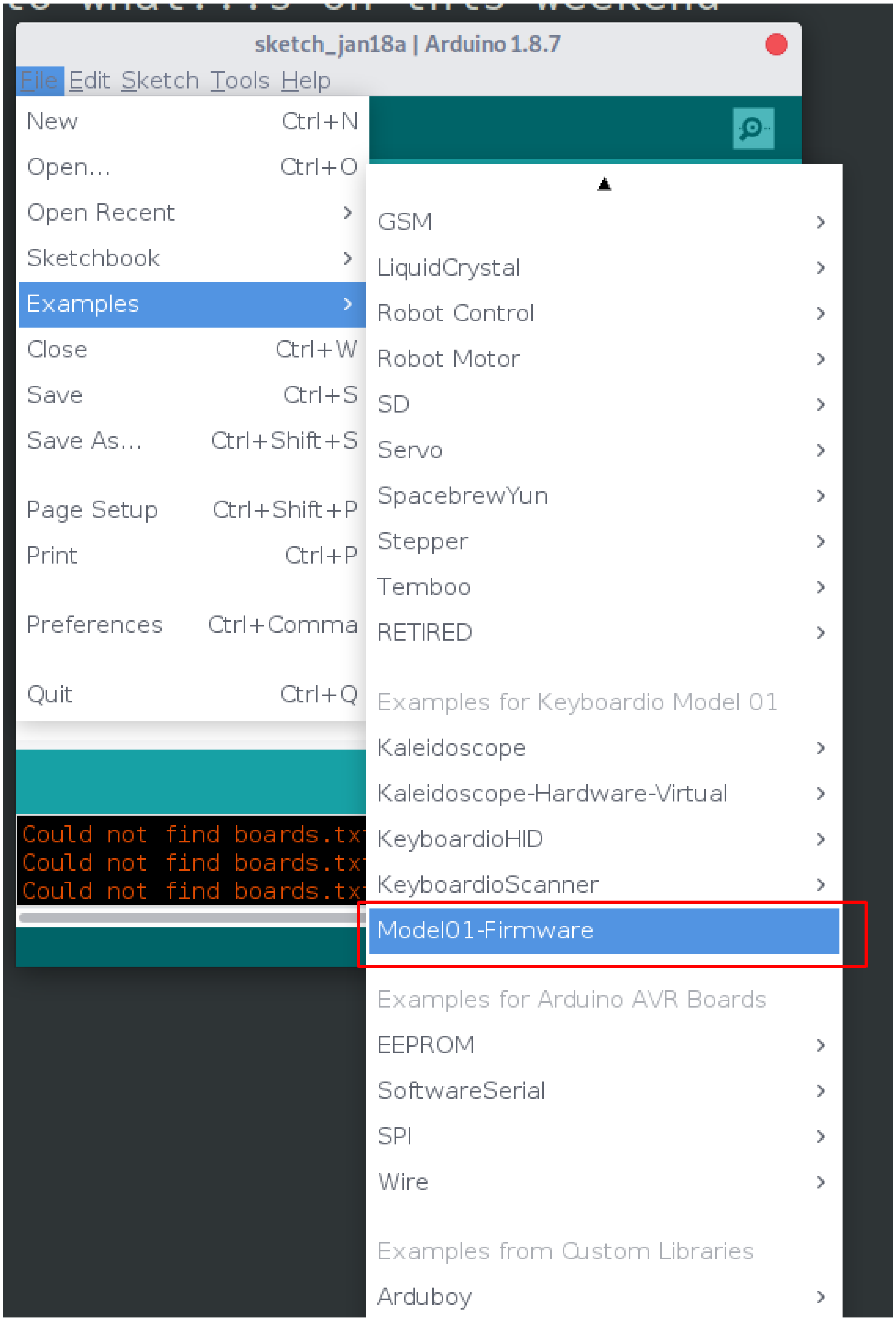
CHAPTER

SEVEN

INSTALL THE LATEST DEFAULT FIRMWARE ON YOUR KEYBOARD

To load the firmware, open the Arduino IDE’s “File” menu, and click on the “Examples” submenu.

If you’re using a Keyboardio Model 01, Scroll down to ‘Model01-Firmware’:



```
Could not find boards.tx
Could not find boards.tx
Could not find boards.tx
```

If you're using another keyboard, you should find it under Examples -> Kaleidoscope -> Devices -> (Your keyboard maker) -> (Your keyboard)

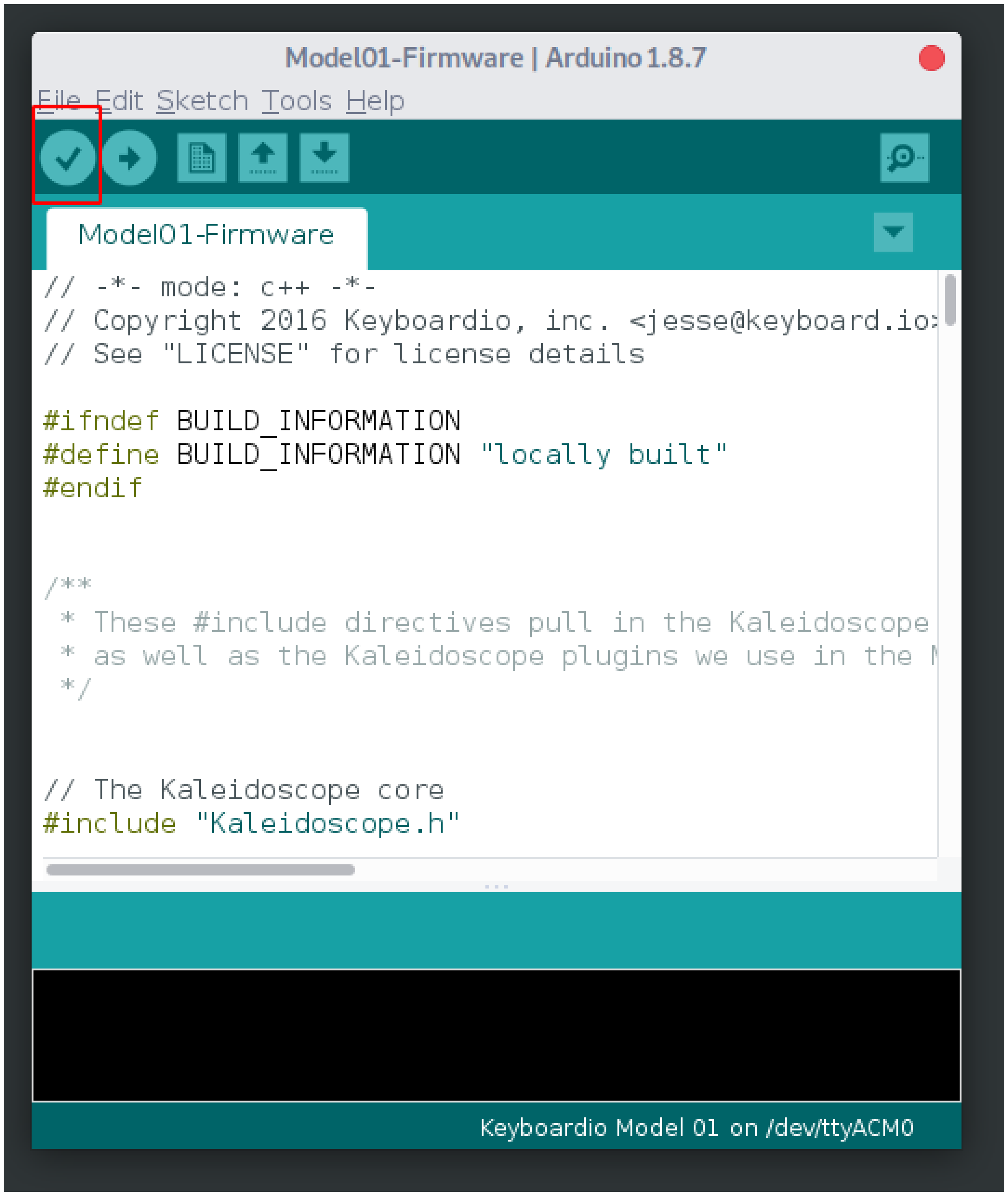
After you pick your example sketch, Arduino wil open a new window showing the sketch's source code above a black message log section and a green status message bar at the bottom. The message log displays detailed information about what Arduino is doing.

Note: *We recommend that you install the default firmware at least once before you start to make changes. This gives you a chance to ensure that the firmware update process works.*

CHAPTER EIGHT

BUILD THE FIRMWARE

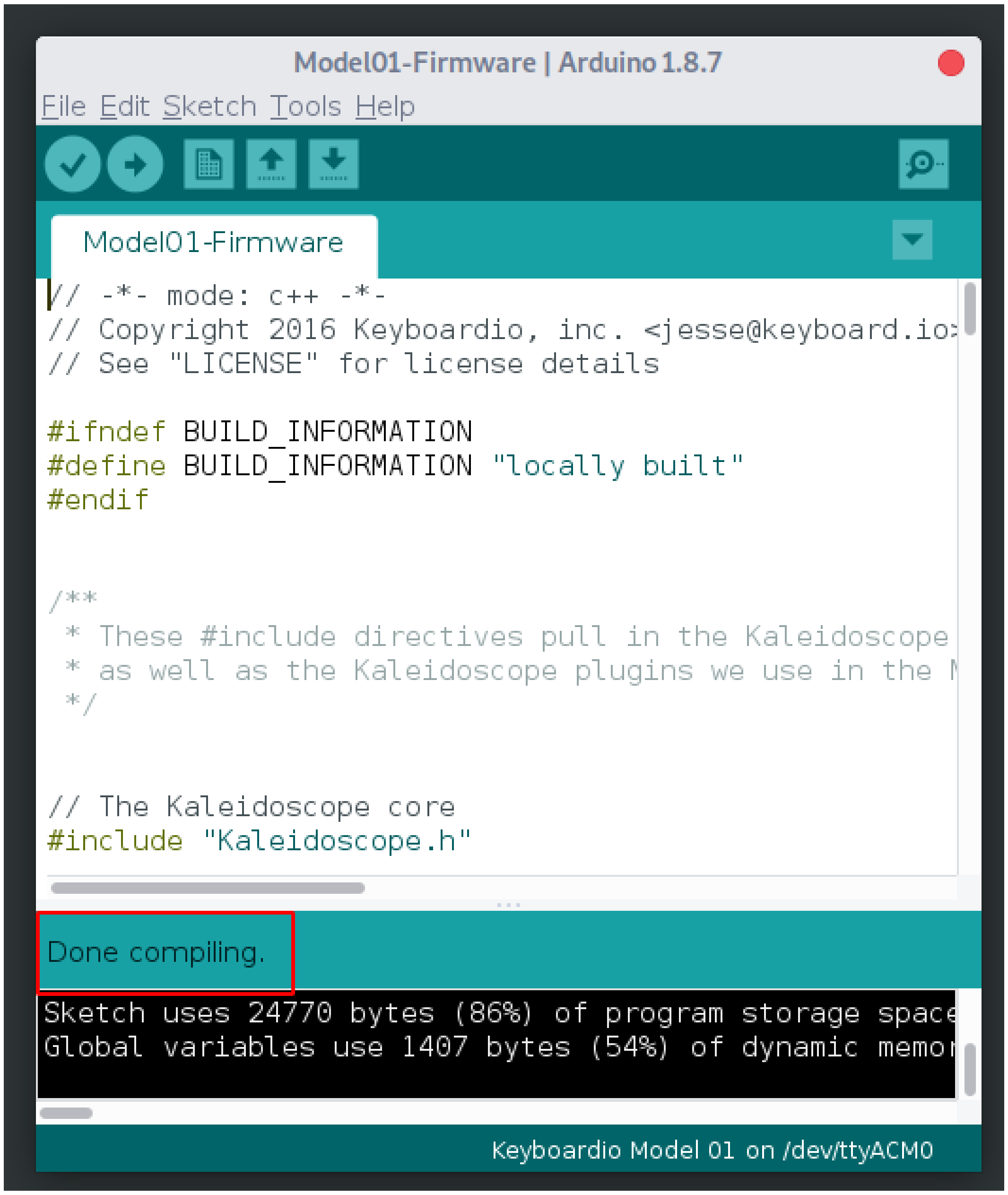
Click the check mark icon below “File” to build your firmware.


Model01-Firmware | Arduino 1.8.7
File Edit Sketch Tools Help
Model01-Firmware

```
// -*- mode: c++ -*-
// Copyright 2016 Keyboardio, inc. <jesse@keyboard.io>
// See "LICENSE" for license details

#ifndef BUILD_INFORMATION
#define BUILD_INFORMATION "locally built"
#endif


/**
 * These #include directives pull in the Kaleidoscope
 * as well as the Kaleidoscope plugins we use in the M
 */


// The Kaleidoscope core
#include "Kaleidoscope.h"
```

Keyboardio Model 01 on /dev/ttyACM0

If the firmware builds successfully, Arduino reports “Done Compiling” in the green status bar.



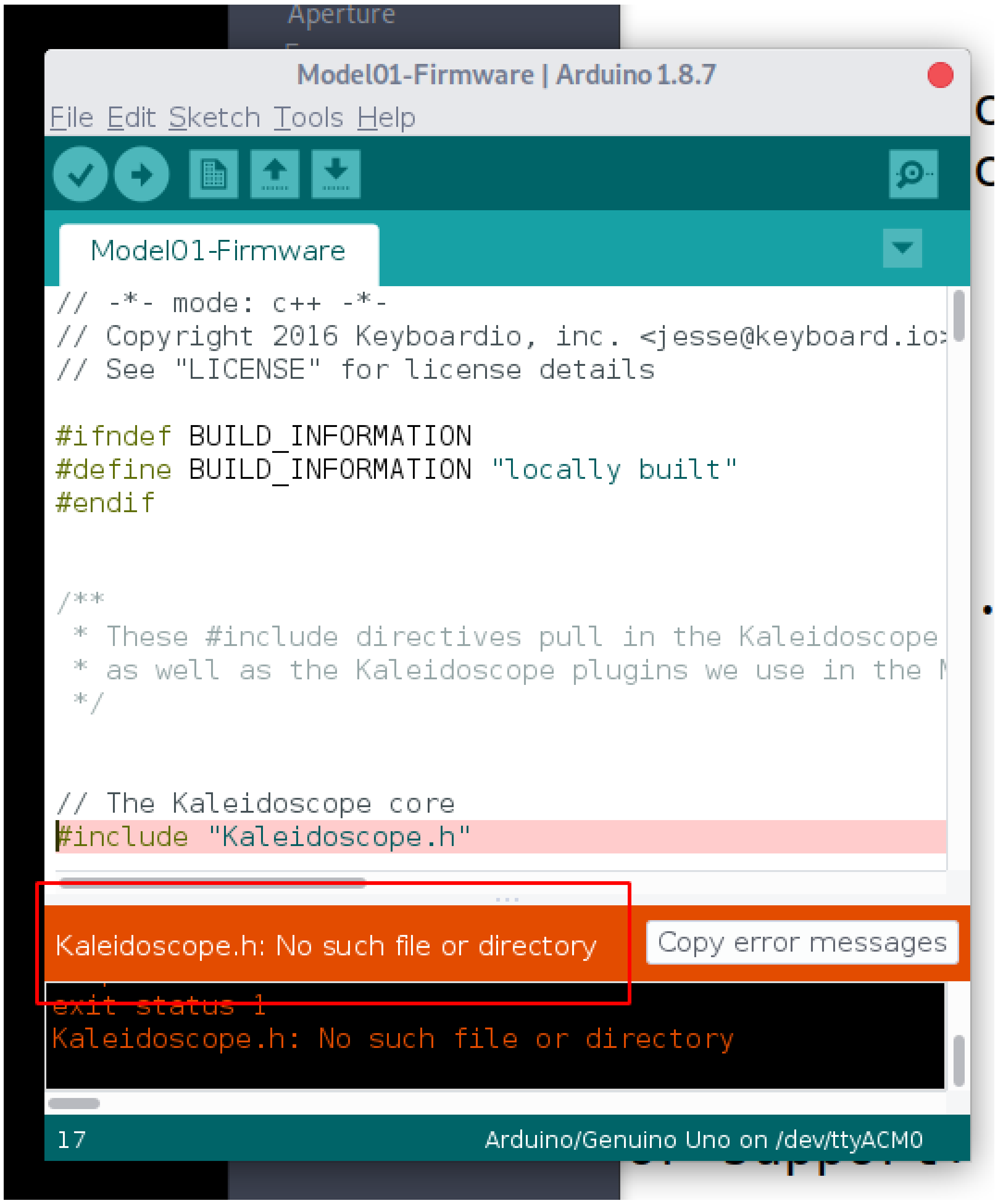
```
// -*- mode: c++ -*-
// Copyright 2016 Keyboardio, inc. <jesse@keyboard.io>
// See "LICENSE" for license details

#ifndef BUILD_INFORMATION
#define BUILD_INFORMATION "locally built"
#endif


/**
 * These #include directives pull in the Kaleidoscope
 * as well as the Kaleidoscope plugins we use in the M
 */


// The Kaleidoscope core
#include "Kaleidoscope.h"
```

Done compiling.

```
Sketch uses 24770 bytes (86%) of program storage space
Global variables use 1407 bytes (54%) of dynamic memor
```

Keyboardio Model 01 on /dev/ttyACM0

If something goes wrong, the status bar turns orange and displays an error message. Additionally, there may be text in the black message log with more details about the error. At this point, it may be helpful to expand the message log so that you can see a bit more about what's going on.


Model01-Firmware | Arduino 1.8.7
File Edit Sketch Tools Help
Model01-Firmware
Copy error messages
17
Arduino/Genuino Uno on /dev/ttyACM0

```
// -*- mode: c++ -*-
// Copyright 2016 Keyboardio, inc. <jesse@keyboard.io>
// See "LICENSE" for license details

#ifndef BUILD_INFORMATION
#define BUILD_INFORMATION "locally built"
#endif



/**
 * These #include directives pull in the Kaleidoscope
 * as well as the Kaleidoscope plugins we use in the 
 */



// The Kaleidoscope core
#include "Kaleidoscope.h"
```

```
Kaleidoscope.h: No such file or directory
```

```
exit status 1
Kaleidoscope.h: No such file or directory
```

CHAPTER

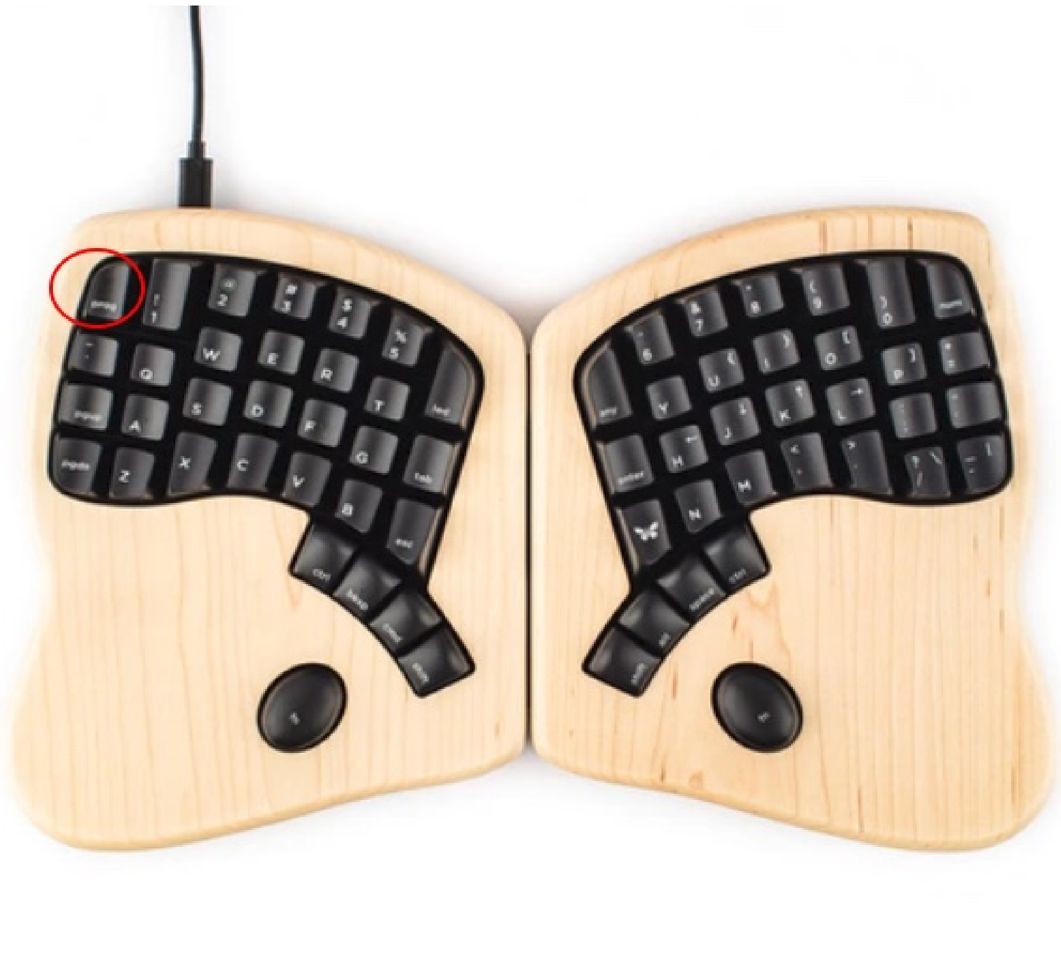
NINE

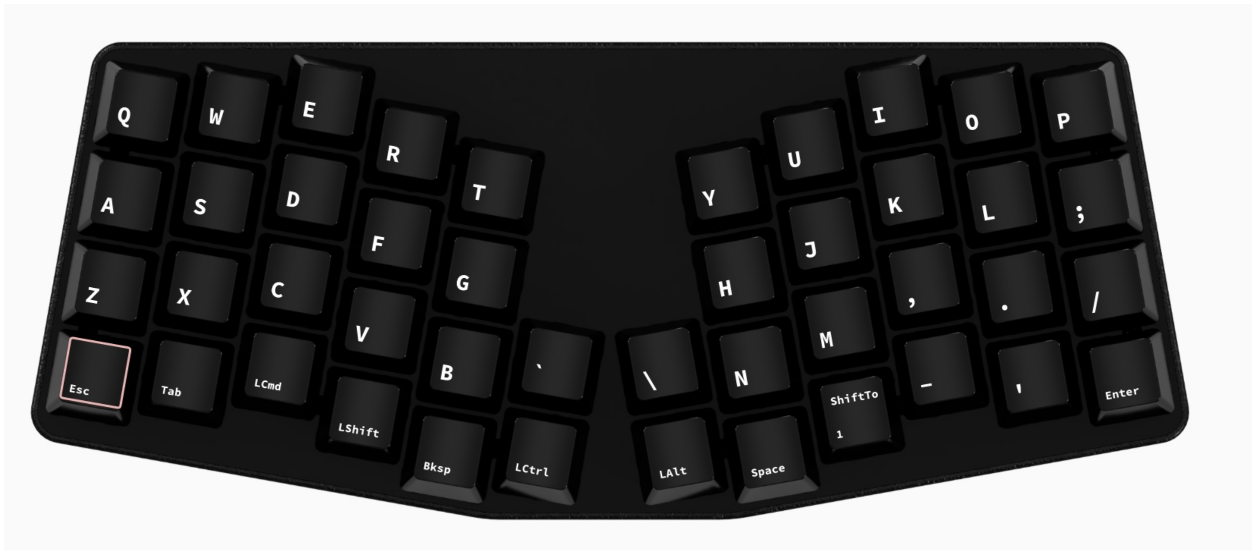
INSTALL THE FIRMWARE

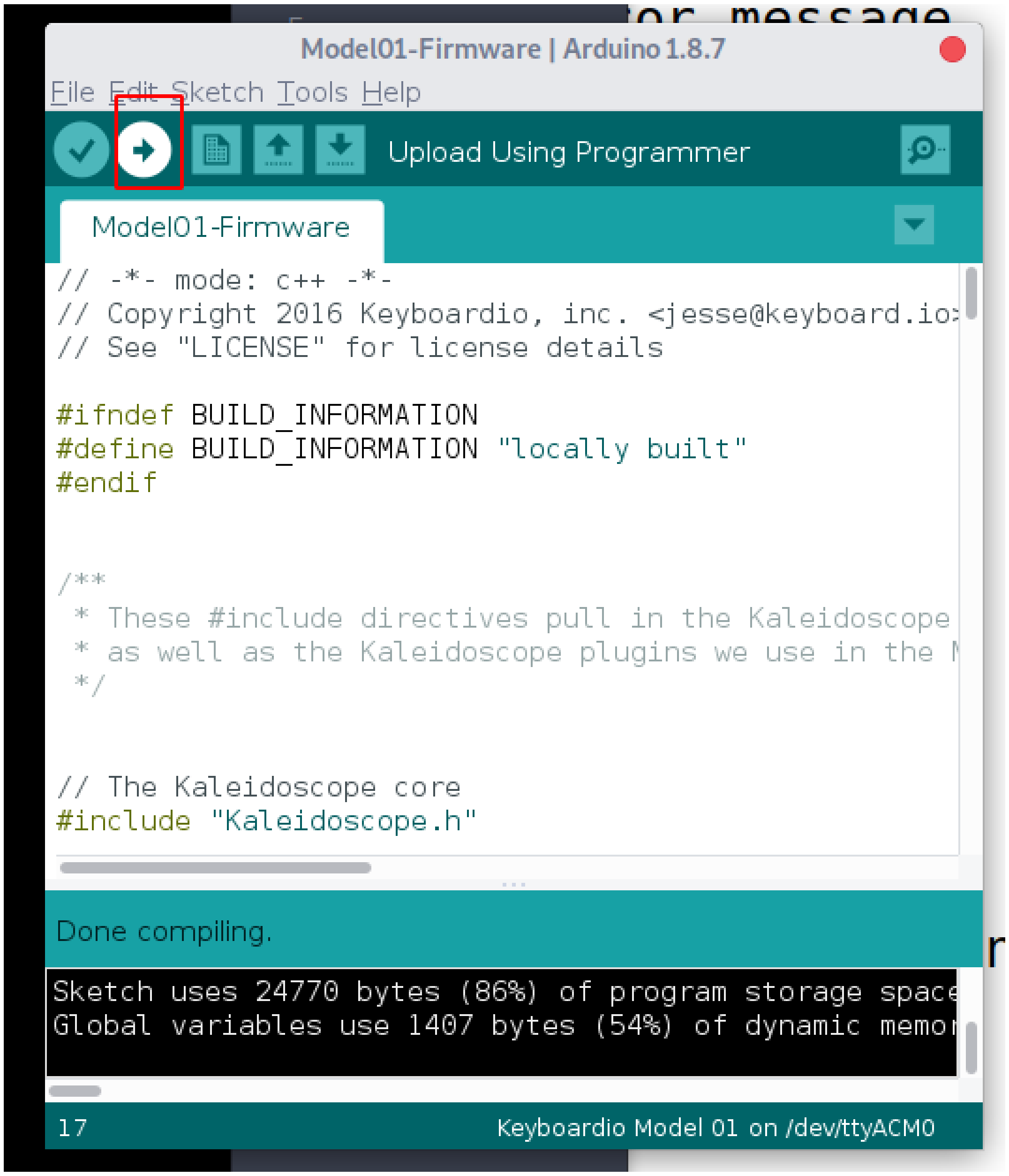
(If you are updating the firmware on a Keyboardio Model 100 for the first time, at this point, you may need to disconnect the keyboard from your computer, hold down the `Prog` key and plug the keyboard back in to put it into programming mode. Once you plug the keyboard back in, the `Prog` key will glow red. You may then continue with this tutorial.)

If your keyboard has a programming interlock key, you'll need to hold it down now. On the Keyboardio Model 01, this is the `Prog` key. On the Keyboardio Atreus, this is the `Esc` key.

Without releasing that key, click on the "right arrow" button in the sketch window menu bar. This starts the firmware installation process.




Q
W
E
R
T
Y
U
I
O
P
A
S
D
F
G
H
J
K
L
;
Z
X
C
V
B
`
\
N
M
,
.
/
Esc
Tab
LCmd
LShift
Bksp
LCtrl
LAlt
Space
ShiftTo 1
_
'
Enter



```
// -*- mode: c++ -*-
// Copyright 2016 Keyboardio, inc. <jesse@keyboard.io>
// See "LICENSE" for license details

#ifndef BUILD_INFORMATION
#define BUILD_INFORMATION "locally built"
#endif


/**
 * These #include directives pull in the Kaleidoscope
 * as well as the Kaleidoscope plugins we use in the M
 */


// The Kaleidoscope core
#include "Kaleidoscope.h"
```

Done compiling.

```
Sketch uses 24770 bytes (86%) of program storage space
Global variables use 1407 bytes (54%) of dynamic memor
```

If the process is successful, Arduino will tell you that in the status area. Some keyboards may also use LEDs to report their results. For example, the Model 01's LED's flash red across the board as the firmware is installed, and then the "LED" key glows blue.

On Windows, you may also see the message "the device Model 01 is undergoing additional configuration."

If you have any trouble flashing your keyboard's firmware, check to see if the issue is addressed on the Troubleshooting Firmware Upload Issues page. Note that if you already have any customized data in the keyboard's EEPROM, any layout differences between the keyboard's original and current firmware may cause issues. See *Using EEPROM* for ways to correct any problems.

CHAPTER

TEN

FOR USERS

10.1 Layers

Layers are an integral part of Kaleidoscope, but a part that is perhaps harder to master than many other things in the firmware. On these pages, we'll make an attempt at explaining layers, what you can do with them, how, and a few common use-cases.

We'll start with a quick use-case guide, before diving deeper into explanations!

10.1.1 How do I. . . ?

How do I switch to a layer, so I can type multiple keys from there?

You can use `LockLayer(n)` or `MoveToLayer(n)`, depending on whether you want other layers to be active at the same time or not. `LockLayer(n)` allows you to build up a stack of layers, while with `MoveToLayer(n)` only the selected layer will be active, without any stacking.

How do I do make layer switching act similar to modifiers?

If you want the layer switch to be active only while the key is held, like in the case of modifiers, the `ShiftToLayer(n)` method does just that.

While switching layers this way is similar to how modifiers work, there are subtle differences. For a longer explanation, see *later*.

10.1.2 Layer theory

First of all, the most important thing to remember is that layers are like a piece of foil, you can place many of them on top of each other, and see through uncovered parts. In other words, you can have multiple layers all active at the same time! As we'll see a few paragraphs later, this can be a very powerful thing.

To better explain how this works in practice, lets look at what layer-related keys we can place on the keymap first. Armed with that knowledge, we'll then explore a few use-cases.

10.1.3 Layer keys

- `LockLayer(n)`: Locking a layer will activate it when the key toggles on, and the layer will remain active until unlocked (with `UnlockLayer(n)` or by pressing `LockLayer(n)` again), even if we release the layer key meanwhile. Think of it like a `Caps lock` or `Num lock` key.
- `ShiftToLayer(n)`: Unlike `LockLayer`, this only activates the layer until the key is held. Once the key is released, the layer deactivates. This behaviour is very similar to that of modifiers.
- `MoveToLayer(n)`: Moving to a layer is very similar to locking it, the only exception is that moving disables all other layers, so only the moved to layer will be active. This allows us to have a less powerful, but simpler way of dealing with layers, as we'll see below.
- `Key_KeymapNext_Momentary` / `Key_KeymapPrevious_Momentary`: These activate the next or the previous layer, momentarily, like `ShiftToLayer(n)`. What it considers `next`, is one layer higher than the currently highest active layer. Similarly, `previous` is one below the currently highest active layer.

10.1.4 Use cases

Locked layers

Locked layers are most useful when you'll want to spend more time on the target layer. One such case is the numpad: when using it, we usually want to enter longer numbers, or use the mathematical operator keys as well. Just imagine hitting a layer lock key, and the right half of your keyboard turning into a numpad! It's closer than the numpad on traditional full-size keyboards, thus less hand movement is required!

Shifted layers

There are many great examples for shifted layers, such as a symbols layer. Let's say we have a number of often used symbols which we want easy access to, preferably near the home row. For example, the various parentheses, brackets and the like are often used in programming. Having them on the home row is incredibly convenient. In most cases, we only need this layer for a very short time, for a symbol or two. As such, locking the layer would be counter-productive. Instead, we use a layer shift key, like if it was a modifier.

As a concrete example, let's imagine a small, ortholinear keyboard, like the Planck. On the bottom row, on the right side of the space bar, we'd have a layer shift key (lets call that `Fn` for now), that takes us to the symbol layer. On the symbol layer, we'd have `{`, `}`, `[`, `]`, `(`, and `)` on the home row. To input `{`, we'd press `Fn + d`, for example. This is still two presses, very much like `Shift + [`, but the keys are more convenient, because we use stronger fingers to press them.

Another - and perhaps an even better - example would be a navigation layer, with cursor keys laid over `WASD`. The reason why this would be a better example, is because in this case, we often want to use modifiers along with the cursor keys, such as `Shift` or `Control`. With a shifted layer, if we have transparent keys at positions where the modifiers are on the base layer, we don't have to repeat the modifier layout on the shifted layer! This makes it easier to experiment with one's layout, because if we move modifiers, we only have to do that on one layer.

Moving to layers

Moving to a layer is very similar to locking one. The only difference is that moving disables all other layers. This in turn, has consequences: we can't return to the previous layer state by repeating the same key. Unlocking a layer that has been activated by `MoveToLayer(n)` will instead cause Kaleidoscope to fall back to the default base layer.

The major advantage of moving to a layer - as opposed to locking one - is the cognitive load. With moving, there is no transparency.[^1] There is only one layer active at any given time. It's a simpler concept to grasp.

10.1.5 Layers, transparency, and how lookup works

The thing that confuses many people about layers is that they can have transparency. What even is a transparent key? Remember the first paragraphs: layers are like a foil. They're see-through, unless parts of it are obstructed. They're like overrides. Any layer you place on top of the existing stack, will override keys in the layers below.

When you have multiple layers active, to figure out what a key does, the firmware will first look at the key position on the most recently activated layer, and see if there's a non-transparent key there. If there is, it will use that. If there isn't, it will start walking backwards on the stack of *active* layers to find the highest one with a non-transparent key. The first one it finds is whose key it will use. If it finds none, then a transparent key will act like a blank one, and do nothing.[^1]

It is important to note that transparent keys are looked up from active layers only, from most recently activated to least. Lets consider that we have three layers, 0, 1, and 2. On a given position, we have a non-transparent key on layers 0 and 1, but the same position is transparent on layer 2. If we have layer 0 and 2 active, the key will be looked up from layer 0, because layer 2 is transparent. If we activate layer 1 too, it will be looked up from there, since layer 1 is higher in the stack than layer 0. In this case, since we activated layer 1 most recently, layer 2 wouldn't even be looked at.

As we just saw, another important factor is that layers are ordered by their order of activation. Whether you activate layer 1 or 2 first, matters. Lets look at another example: we have three layers, 0, 1, and 2. On a given position, we have a non-transparent key on every layer. If we have just layer 0 active, it will be looked up from there. If we activate layer 2, then the firmware will look there first. If we activate layer 1 as well, then - since now layer 1 is the most recently activated layer - the firmware will look the code up from layer 1, without looking at layer 2. It would only look at layer 2 if the key was transparent on layer 1.

[^1]: Except that the base layer is always active implicitly, so if all active layers are transparent for a particular key, its value will come from the base layer.

10.2 Core plugin overview

This is an annotated list of some of Kaleidoscope's most important core plugins. You may also want to consult the *automatically generated list of all plugins bundled with Kaleidoscope*.

You can find a list of third-party plugins not distributed as part of Kaleidoscope on the forums.

10.2.1 EEPROM-Keymap

EEPROM-Keymap Documentation

While keyboards usually ship with a keymap programmed in, to be able to change that keymap, without flashing new firmware, we need a way to place the keymap into a place we can update at run-time, and which persists across reboots. Fortunately, we have a bit of EEPROM on the keyboard, and can use it to store either the full keymap (and saving space in the firmware then), or store an overlay there. In the latter case, whenever there is a non-transparent key on the overlay, we will use that instead of the keyboard default.

In short, this plugin allows us to change our keymaps, without having to compile and flash new firmware. It does so through the use of the Focus plugin.

10.2.2 Escape-OneShot

Escape-OneShot Documentation

Turn the Esc key into a special key, that can cancel any active OneShot effect - or act as the normal Esc key if none are active. For those times when one accidentally presses a one-shot key, or change their minds.

10.2.3 Leader

Leader Documentation

Leader keys are a kind of key where when they are tapped, all following keys are swallowed, until the plugin finds a matching sequence in the dictionary, it times out, or fails to find any possibilities. When a sequence is found, the corresponding action is executed, but the processing still continues. If any key is pressed that is not the continuation of the existing sequence, processing aborts, and the key is handled normally.

This behaviour is best described with an example. Suppose we want a behaviour where `LEAD u` starts unicode input mode, and `LEAD u h e a r t` should result in a heart symbol being input, and we want `LEAD u 0 0 e 9 SPC` to input é, and any other hex code that follows `LEAD u`, should be handled as-is, and passed to the host. Obviously, we can't have all of this in a dictionary.

So we put `LEAD u` and `LEAD u h e a r t` in the dictionary only. The first will start unicode input mode, the second will type in the magic sequence that results in the symbol, and then aborts the leader sequence processing. With this setup, if we type `LEAD u 0`, then `LEAD u` will be handled first, and start unicode input mode. Then, at the 0, the plugin notices it is not part of any sequence, so aborts leader processing, and passes the key on as-is, and it ends up being sent to the host. Thus, we covered all the cases of our scenario!

10.2.4 Macros

Macros Documentation

Macros are a standard feature on many keyboards and powered ones are no exceptions. Macros are a way to have a single key-press do a whole lot of things under the hood: conventionally, macros play back a key sequence, but with Kaleidoscope, there is much more we can do. Nevertheless, playing back a sequence of events is still the primary use of macros.

Playing back a sequence means that when we press a macro key, we can have it play pretty much any sequence. It can type some text for us, or invoke a complicated shortcut - the possibilities are endless!

10.2.5 MagicCombo

MagicCombo Documentation

The MagicCombo extension provides a way to perform custom actions when a particular set of keys are held down together. The functionality assigned to these keys are not changed, and the custom action triggers as long as all keys within the set are pressed. The order in which they were pressed do not matter.

This can be used to tie complex actions to key chords.

10.2.6 OneShot

OneShot Documentation

One-shots are a new kind of behaviour for your standard modifier and momentary layer keys: instead of having to hold them while pressing other keys, they can be tapped and released, and will remain active until any other key is pressed. In short, they turn `Shift, A` into `Shift+A`, and `Fn, 1` to `Fn+1`. The main advantage is that this allows us to place the modifiers and layer keys to positions that would otherwise be awkward when chording. Nevertheless, they still act as normal when held, that behaviour is not lost.

Furthermore, if a one-shot key is tapped two times in quick succession, it becomes sticky, and remains active until disabled with a third tap. This can be useful when one needs to input a number of keys with the modifier or layer active, and still does not wish to hold the key down. If this feature is undesirable, unset the `OneShot.double_tap_sticky property` (see later).

To make multi-modifier, or multi-layer shortcuts possible, one-shot keys remain active if another one-shot of the same type is tapped, so `Ctrl, Alt, b` becomes `Ctrl+Alt+b`, and `L1, L2, c` is turned into `L1+L2+c`.

10.2.7 Qukeys

Qukeys Documentation

A Qukey is a key that has two possible values, usually a modifier and a printable character. The name is a play on the term “qubit” (short for “quantum bit”) from quantum computing. The value produced depends on how long the key press lasts, and how it is used in combination with other keys (roughly speaking, whether the key is “tapped” or “held”).

The *primary* value (a printable character) of a Qukey is output if the key is “tapped” (i.e. quickly pressed and released). If it is held long enough, it will instead produce the Qukey’s *alternate* value (usually a modifier). It will also produce that alternate value if a subsequent key is tapped after the initial keypress of the Qukey, even if both keys are released before the time it takes to produce the alternate value on its own. This makes it feasible for most people to use Qukeys on home-row keys, without slowing down typing. In this configuration, it can become very comfortable to use modifier combinations, without needing to move one’s hands from the home position at all.

Qukeys can be defined to produce any two keys, including other plugin keys and keys with modifier flags applied. For example, one could define a Qukey to produce `Shift + 9` when tapped, and a OneShot `Ctrl` when held.

It is also possible to use Qukeys like SpaceCadet (see below), by setting the primary value to a modifier, and the alternate value to a printable character (e.g. `(`). In that case, the behavior is reversed, and the alternate value will only be used if the key is pressed and released without any rollover to a subsequent key press.

10.2.8 ShapeShifter

ShapeShifter Documentation

ShapeShifter is a plugin that makes it considerably easier to change what symbol is input when a key is pressed together with `Shift`. If one wants to rearrange the symbols on the number row for example, without modifying the layout on the operating system side, this plugin is where one can turn to.

What it does, is very simple: if any key in its dictionary is found pressed while `Shift` is held, it will press another key instead of the one triggering the event. For example, if it sees `Shift + 1` pressed together, which normally results in a `!`, it will press `4` instead of `1`, inputting `$`.

10.2.9 SpaceCadet

SpaceCadet Documentation

Space Cadet is a way to make it more convenient to input parens - those `(` and `)` things -, symbols that a lot of programming languages use frequently. If you are working with Lisp, you are using these all the time.

What it does, is that it turns your left and right `Shift` keys into parens if you tap and release them, without pressing any other key while holding them. Therefore, to input, say, `(print foo)`, you don't need to press `Shift`, hold it, and press `9` to get a `(`, you simply press and release `Shift`, and continue writing. You use it as if you had a dedicated key for parens!

But if you wish to write capital letters, you hold it, as usual, and you will not see any parens when you release it. You can also hold it for a longer time, and it still would act as a `Shift`, without the parens inserted on release: this is useful when you want to augment some mouse action with `Shift`, to select text, for example.

After getting used to the Space Cadet style of typing, you may wish to enable this sort of functionality on other keys, as well. Fortunately, the Space Cadet plugin is configurable and extensible to support adding symbols to other keys. Along with `(` on your left `Shift` key and `)` on your right `Shift` key, you may wish to add other such programming mainstays as `{` to your left-side `cmd` key, `}` to your right-side `alt` key, `[` to your left `Control` key, and `]` to your right `Control` key. You can map the keys in whatever way you may wish to do, so feel free to experiment with different combinations and discover what works best for you!

10.2.10 TapDance

TapDance Documentation

Tap-dance keys are general purpose, multi-use keys, which trigger a different action based on the number of times they were tapped in sequence. As an example to make this clearer, one can have a key that inputs `A` when tapped once, inputs `B` when tapped twice, and lights up the keyboard in Christmas colors when tapped a third time.

This behaviour is most useful in cases where we have a number of things we perform rarely, where tapping a single key repeatedly is not counter-productive. Such cases include - for example - multimedia forward / backward keys: forward on single tap, backward on double. Of course, one could use modifiers to achieve a similar effect, but that's two keys to use, this is only one. We can also hide some destructive functionality behind a number of taps: reset the keyboard after 4 taps, and light up LEDs in increasingly frightful colors until then.

How does it work?

To not interfere with normal typing, tap-dance keys have two ways to decide when to call an action: they either get interrupted, or they time out. Every time a tap-dance key is pressed, the timer resets, so one does not have to finish the whole tapping sequence within a short time limit. The tap-dance counter continues incrementing until one of these cases happen.

When a tap-dance key is pressed and released, and nothing is pressed on the keyboard until the timeout is reached, then the key will time out, and trigger an action. Which action, depends on the number of times it has been tapped up until this point.

When a tap-dance key is pressed and released, and another key is hit before the timer expires, then the tap-dance key will trigger an action first, perform it, and only then will the firmware continue handling the interrupting key press. This is to preserve the order of keys pressed.

In both of these cases, the `tapDanceAction` will be called, with `tapDanceIndex` set to the index of the tap-dance action (as set in the keymap), the `tapCount`, and tapDanceAction set to either `kaleidoscope::TapDance::Interrupt`, or `kaleidoscope::TapDance::Timeout`. If we continue holding the key, then as long as it is held, the same function will be called with tapDanceAction set to `kaleidoscope::TapDance::Hold`. When the key is released, after either an Interrupt or Timeout action was triggered, the function will be called with tapDanceAction set to `kaleidoscope::TapDance::Release`.

These actions allow us to create sophisticated tap-dance setups, where one can tap a key twice and hold it, and have it repeat, for example.

There is one additional value the tapDanceAction parameter can `take: kaleidoscope::TapDance::Tap`. It is called with this argument for each and every tap, even if no action is to be triggered yet. This is so that we can have a way to do some side-effects, like light up LEDs to show progress, and so on.

10.2.11 TopsyTurvy

TopsyTurvy Documentation

TopsyTurvy is a plugin that inverts the behaviour of the Shift key for some selected keys. That is, if configured so, it will input `!` when pressing the `1` key without `Shift`, but with the modifier pressed, it will input the original `1` symbol.

10.3 Using EEPROM

10.3.1 Why Use EEPROM?

While we've done our best to make it easy to change how your keyboard works by changing your firmware & re-flashing it, sometimes it would be convenient to be able to make changes without having to go through that rigamarole. Maybe you'd like to be able to use a GUI like Chrysalis to configure your keyboard layout or LED themes, or maybe your sketch is getting very complicated and you're looking for a way to save program memory. In either case, you'll want to use EEPROM to store your settings.

10.3.2 What is EEPROM?

EEPROM stands for "Electronic Erasable Programmable Read-Only Memory" and is one of the three memory mediums your keyboard has. The other two are RAM, which is used for variables when running your code, and program memory, which is used for storing the program, as well as some other select pieces of data (if you're curious, the bit in your sketch where it says `PROGMEM` indicates that a variable is being stored in program memory instead of RAM). RAM we want to keep as free as we can, since running our code will need some RAM to work. While we can put stuff in PROGMEM, your code itself will take up some room there, so it may be useful to store things elsewhere. EEPROM provides us with another place to store things that can free up RAM and PROGMEM. Additionally, by leveraging a few plugins, we can store configuration in EEPROM and allow a GUI tool on the connected computer to change settings on the keyboard!

10.3.3 Move Settings to EEPROM

There are a few important Kaleidoscope plugins for putting settings in EEPROM:

- Kaleidoscope-Focus - This plugin is what enables communication between your keyboard and programs running on your computer; all the following plugins require you to be using this if you want to be able to change your settings from the computer without re-flashing.
- Kaleidoscope-EEPROM-Settings - This is a plugin that doesn't do much by itself, but most of the other EEPROM plugins will need active to be able to make use of EEPROM storage.
- Kaleidoscope-EEPROM-Keymap - This plugin uses Focus and EEPROM-Settings to allow either overriding or fully replacing the programmed-in keymap without reflashing (by means of a program like Chrysalis running on your computer).
- Kaleidoscope-Colormap - This plugin allows you to use a computer-side program to set a (static – i.e. the keys won't change colour over time) LED theme for each layer.

All these plugins have minimal installation that can be found in their respective READMEs. After following the instructions for each and adding them together, you should be able to download a program that knows how to communicate with the keyboard (i.e. Chrysalis and you can start customizing settings without having to do any more programming!

10.4 Migrating EEPROM contents between firmware changes

When you flash new firmware that adds or removes plugins utilizing EEPROM storage, your configuration stored on the keyboard is likely to break. This is, because the stored data from the old firmware does not align with the storage layout of the new firmware.

> Don't worry, your config is not gone - flashing the old firmware will provide you a functioning configuration again.

To work around this breakage, the configuration can be extracted via *Focus commands* before flashing - and restored after flashing. Currently this is not part of the flashing process, but can easily be done with helper scripts `eeprom-backup.sh` and `eeprom-restore.sh`.

10.5 What can go on your keymap

Eventually there should be a helpful table here with good definitions for the common codes. In the meantime, you can check these files for all the codes the Keyboardio supports:

- Most of the common keyboard key codes are here:

  key_defs/keyboard.h

- Key codes for system tasks like shutting down, switching windows, and moving through menus are here:

  key_defs/sysctl.h

- A wide range of key codes for controlling consumer electronics, most of which are probably not relevant, are in this file:

  key_defs/consumerctl.h

10.5.1 In-keymap chorded keys

In addition, the keys in `key_defs/keyboard.h` can be augmented with modifier macros: `LCTRL()`, `LSHIFT()`, `LALT()`, `LGUI()` and `RALT()` to add chorded keys to your keymap. For example `LCTRL(LALT(Key_Delete))` can be used to add control-alt-delete as a single key to your keymap, should you wish. The innermost bracket must be of the standard format as taken from the above key definitions, and all other modifiers must be from the aforementioned list, and in that format. This does allow you to create single keys for multiple modifiers, e.g. `LCTRL(LALT(LSHIFT(Key_LeftGui)))`, when held, would have the effect of all left modifiers at once. These modifier macros only work for standard keys! When applied to any key provided by a plugin, they will have no effect.

10.5.2 Combination modifier/layer shift keys

The `ML()` preprocessor macro can be used to define a key which will act as both a keyboard modifier and a layer shift while the key is held. For example, `ML(LeftShift, 3)` will act as both `ShiftToLayer(3)` and `Key_LeftShift`. Any of the eight modifier keys can be used, and is specified without the `Key_` prefix. The layer shift can be to any layer in the range 0-31.

10.6 Core LED Effects

This is the list of the stable LED effects in the core libraries.

LED-ActiveModColor

A very simple plugin, that lights up the LED in white under any active modifier, for the duration of its activity. Also supports one-shots.

Kaleidoscope-LEDEffects

The LEDEffects plugin provides a selection of LED effects, each of them fairly simple, simple enough to not need a plugin of their own. There are a number of different effects included in the package, all of them are available once including the header, and one's free to choose any number of them. Kaleidoscope-LEDEffect-BootGreeting

If you want to have your keyboard signal when it turns on, but you don't want to use any more complicated LED modes, this plugin is for you. It will make the `LEDEffectNext` key on your keymap slowly breathe for about ten seconds after plugging the keyboard in (without blocking the normal functionality of the keyboard, of course).

Kaleidoscope-LEDEffect-Breathe

Provides a breathing effect for the keyboard. Breathe in, breathe out.

Kaleidoscope-LEDEffect-Chase

A simple LED effect where one color chases another across the keyboard and back, over and over again. Playful colors they are.

Kaleidoscope-LEDEffect-Rainbow

Two colorful rainbow effects are implemented by this plugin: one where the rainbow waves through the keys, and another where the LEDs breathe though the colors of a rainbow. The difference is that in the first case, we have all the rainbow colors on display, and it waves through the keyboard. In the second case, we have only one color at a time, for the whole board, and the color cycles through the rainbow's palette.

Kaleidoscope-LEDEffect-SolidColor

This plugin provides tools to build LED effects that set the entire keyboard to a single color. For show, and for backlighting purposes.

LED-Stalker

Demoed in the backer update, this adds an effect that stalks your keys: whenever a key is pressed, the LED under it lights up, and the slowly fades away once the key is released. This provides a kind of trailing effect.

There are two color schemes currently: Haunt, which is a white-ish, ghostly color that follows your fingers, and BlazingTrail, demoed in the video, which lights your keyboard on fire. It looks much better in real life.

10.7 How to write a Kaleidoscope plugin

This is a brief guide intended for those who want to write custom Kaleidoscope plugins. It covers basic things you'll need to know about how Kaleidoscope calls plugin event handlers, and how it will respond to actions taken by those plugins.

10.7.1 What can a plugin do?

There are many things that Kaleidoscope plugins are capable of, from LED effects, serial communication with the host, altering HID reports, and interacting with other plugins. It's useful to break these capabilities down into some broad categories, based on the types of input a plugin can respond to.

- Key events (key switches toggling on and off)
- Focus commands (sent to the keyboard from software on the host via the serial port)
- LED updates
- Keymap layer changes
- Timers

10.7.2 An example plugin

To make a Kaleidoscope plugin, we create a subclass of the `kaleidoscope::Plugin` class, usually in the `kaleidoscope::plugin` namespace:

```
namespace kaleidoscope {
namespace plugin {

class MyPlugin : public Plugin {};

} // namespace kaleidoscope
} // namespace plugin
```

This code can be placed in a separate C++ source file, but it's simplest to just define it right in the sketch's *.ino file for now.

By convention, we create a singleton object named like the plugin's class in the global namespace. This is typical of Arduino code.

```
kaleidoscope::plugin::MyPlugin MyPlugin;
```

Next, in order to connect that plugin to the Kaleidoscope event handler system, we need to register it in the call to the preprocessor macro `KALEIDOSCOPE_INIT_PLUGINS()` in the sketch:

```
KALEIDOSCOPE_INIT_PLUGINS(MyPlugin, OtherPlugin);
```

To make our plugin do anything useful, we need to add [[event-handler-hooks]] to it. This is how Kaleidoscope delivers input events to its registered plugins. Here's an example:

```
class MyPlugin : public Plugin {
 public:
  EventHandlerResult onKeyEvent(KeyEvent &event);
};
```

This will result in `MyPlugin.onKeyEvent()` being called (along with other plugins' `onKeyEvent()` methods) when Kaleidoscope detects a key state change. This function returns one of three `EventHandlerResult` values:

- `EventHandlerResult::OK` indicates that Kaleidoscope should proceed on to the event handler for the next plugin in the chain.
- `EventHandlerResult::ABORT` indicates that Kaleidoscope should stop processing immediately, and treat the event as if it didn't happen.
- `EventHandlerResult::EVENT_CONSUMED` stops event processing like `ABORT`, but records that the key is being held.

The `onKeyEvent()` method takes one argument: a reference to a `KeyEvent` object, which is a simple container for these essential bits of information:

- `event.addr` — the physical location of the keyswitch, if any
- `event.state` — a bitfield containing information on the current and previous state of the keyswitch (from which we can find out if it just toggled on or toggled off)
- `event.key` — a 16-bit `Key` value containing the contents looked up from the sketch's current keymap (if the key just toggled on) or the current live value of the key (if the key just toggled off)

Because the `KeyEvent` parameter is passed by (mutable) reference, our plugin's `onKeyEvent()` method can alter the components of the event, causing subsequent plugins (and, eventually, Kaleidoscope itself) to treat it as if it was a different event. In practice, except in very rare cases, the only member of a `KeyEvent` that a plugin should alter is `event.key`. Here's a very simple `onKeyEvent()` handler that changes all `X` keys into `Y` keys:

```
EventHandlerResult onKeyEvent(KeyEvent &event) {
  if (event.key == Key_X)
    event.key = Key_Y;
  return EventHandlerResult::OK;
}
```

The difference between `ABORT` & `EVENT_CONSUMED`

Here's a plugin that will suppress all `X` key events:

```
EventHandlerResult onKeyEvent(KeyEvent &event) {
  if (event.key == Key_X)
    return EventHandlerResult::ABORT;
  return EventHandlerResult::OK;
}
```

Here's an almost identical plugin that has an odd failure mode:

```
EventHandlerResult onKeyEvent(KeyEvent &event) {
  if (event.key == Key_X)
    return EventHandlerResult::EVENT_CONSUMED;
  return EventHandlerResult::OK;
}
```

In this case, when an `X` key is pressed, no Keyboard HID report will be generated and sent to the host, but the key will still be recorded by Kaleidoscope as "live". If we hold that key down and press a `Y` key, we will suddenly see both `x` *and* `y` in the output on the host. This is because returning `ABORT` suppresses the key event entirely, as if it never happened, whereas `EVENT_CONSUMED` signals to Kaleidoscope that the key should still become "live", but that no further processing is necessary. In this case, since we want to suppress all `X` keys entirely, we should return `ABORT`.

A complete in-sketch plugin

Here's an example of a very simple plugin, defined as it would be in a firmware sketch (e.g. a `*.ino` file):

```
namespace kaleidoscope {
namespace plugin {

class KillX : public Plugin {
 public:
  EventHandlerResult onKeyEvent(KeyEvent &event) {
    if (event.key == Key_X)
      return EventHandlerResult::ABORT;
    return EventHandlerResult::OK;
  }
};

} // namespace kaleidoscope
} // namespace plugin

kaleidoscope::plugin::KillX;
```

On its own, this plugin won't have any effect unless we register it later in the sketch like this:

```
KALEIDOSCOPE_INIT_PLUGINS(KillX);
```

Note: `KALEIDOSCOPE_INIT_PLUGINS()` should only appear once in a sketch, with a list of all the plugins to be registered.

10.7.3 Plugin registration order

Obviously, the `KillX` plugin isn't very useful. But more important, it's got a potential problem. Suppose we had another plugin defined, like so:

```
namespace kaleidoscope {
namespace plugin {

class YtoX : public Plugin {
 public:
  EventHandlerResult onKeyEvent(KeyEvent &event) {
    if (event.key == Key_Y)
      event.key = Key_X;
    return EventHandlerResult::OK;
  }
};

} // namespace kaleidoscope
} // namespace plugin

kaleidoscope::plugin::YtoX;
```

`YtoX` changes any `Y` key to an `X` key. These two plugins both work fine on their own, but when we put them together, we get some undesirable behavior. Let's try it this way first:

```
KALEIDOSCOPE_INIT_PLUGINS(YtoX, KillX);
```

This registers both plugins' event handlers with Kaleidoscope, in order, so for each `KeyEvent` generated in response to a keyswitch toggling on or off, `YtoX.onKeyEvent(event)` will get called first, then `KillX.onKeyEvent(event)` will get called.

If we press `X`, the `YtoX` plugin will effectively ignore the event, allowing it to pass through to `KillX`, which will abort the event.

If we press `Y`, `YtoX.onKeyEvent()` will change `event.key` from `Key_Y` to `Key_X`. Then, `KillX.onKeyEvent()` will abort the event. As a result, both `X` and `Y` keys will be suppressed by the combination of the two plugins.

Now, let's try the same two plugins in the other order:

```
KALEIDOSCOPE_INIT_PLUGINS(KillX, YtoX);
```

If we press `X`, its keypress event will get aborted by `KillX.onKeyEvent()`, and that key will not become live, so when it gets released, the event generated won't have the value `Key_X`, but will instead by `Key_Inactive`, which will not result in anything happening, either from the plugins or from Kaleidoscope itself.

Things get interesting if we press and release `Y`, though. First, `KillX.onKeyEvent()` will simply return `OK`, allowing `YtoX.onKeyEvent()` to change `event.key` from `Key_Y` to `Key_X`, causing that `Key_X` to become live, and sending its keycode to the host in the Keyboard USB HID report. That's all as expected, but then we release the key, and that's were it goes wrong.

`KillX.onKeyEvent()` doesn't distinguish between presses and releases. When a key toggles off, rather than looking up that key's value in the keymap, Kaleidoscope takes it from the live keys array. That means that `event.key` will be `Key_X` when `KillX.onKeyEvent()` is called, which will result in that event being aborted. And when an event is aborted, the key's entry in the live keys array doesn't get updated, so Kaleidoscope will treat it as if the key is still held after release. Thus, far from preventing the keycode for `X` getting to the host, it keeps that key pressed forever! The `X` key becomes "stuck on" because the plugin suppresses both key *presses* and key *releases*.

Differentiating between press and release events

There is a solution to this problem, which is to have `KillX` suppress `Key_X` toggle-on events, but not toggle-off events:

```
EventHandlerResult KillX::onKeyEvent(KeyEvent &event) {
  if (event.key == Key_X && keyToggledOn(event.state))
    return EventHandlerResult::ABORT;
  return EventHandlerResult::OK;
}
```

Kaleidoscope provides `keyToggledOn()` and `keyToggledOff()` functions that operate on the `event.state` bitfield, allowing plugins to differentiate between the two different event states. With this new version of the `KillX` plugin, it won't keep an `X` key live, but it will stop one from *becoming* live.

Our two plugins still yield results that depend on registration order in `KALEIDOSCOPE_INIT_PLUGINS()`, but the bug where the `X` key becomes "stuck on" is gone.

It is very common for plugins to only act on key toggle-on events, or to respond differently to toggle-on and toggle-off events.

10.7.4 Timers

Another thing that many plugins need to do is handle timeouts. For example, the OneShot plugin keeps certain keys live for a period of time after those keys are released. Kaleidoscope provides some infrastructure to help us keep track of time, starting with the `afterEachCycle()` "event" handler function.

The `onKeyEvent()` handlers only get called in response to keyswitches toggling on and off (or as a result of plugins calling `Runtime.handleKeyEvent()`). If the user isn't actively typing for a period, its `onKeyEvent()` handler won't get called at all, so it's not very useful to check timers in that function. Instead, if we need to know if a timer has expired, we need to do it in a function that gets called regularly, regardless of input. The `afterEachCycle()` handler gets called once per cycle, guaranteed.

This is what an `afterEachCycle()` handler looks like:

```
EventHandlerResult afterEachCycle() {
  return EventHandlerResult::OK;
}
```

It returns an `EventHandlerResult`, like other event handlers, but this one's return value is ignored by Kaleidoscope; returning `ABORT` or `EVENT_CONSUMED` has no effect on other plugins.

In addition to this, we need a way to keep track of time. For this, Kaleidoscope provides the function `Runtime.millisAtCycleStart()`, which returns an unsigned integer representing the number of milliseconds that have elapsed since the keyboard started. It's a 32-bit integer, so it won't overflow until about one month has elapsed, but we usually want to use as few bytes of RAM as possible on our MCU, so most timers store only as many bytes as needed, usually a `uint16_t`, which overflows after about one minute, or even a `uint8_t`, which is good for up to a quarter of a second.

We need to use an integer type that's at least as big as the longest timeout we expect to be used, but integer overflow can still give us the wrong answer if we check it by naïvely comparing the current time to the time at expiration, so Kaleidoscope provides a timeout-checking service that's handles the integer overflow properly: `Runtime.hasTimeExpired(start_time, timeout)`. To use it, your plugin should store a timestamp when the timer begins, using `Runtime.millisAtCycleStart()` (usually set in response to an event in `onKeyEvent()`). Then, in its `afterEachCycle()` call `hasTimeExpired()`:

```
namespace kaleidoscope {
namespace plugin {

class MyPlugin : public Plugin {
 public:
  constexpr uint16_t timeout = 500;

  EventHandlerResult onKeyEvent(KeyEvent &event) {
    if (event.key == Key_X && keyToggledOn(event.state)) {
      start_time_ = Runtime.millisAtCycleStart();
      timer_running_ = true;
    }
    return EventHandlerResult::OK;
  }

  EventHandlerResult afterEachCycle() {
    if (Runtime.hasTimeExpired(start_time_, timeout)) {
      timer_running_ = false;
      // do something...
    }
    return EventHandlerResult::OK;
  }
```

(continues on next page)

(continued from previous page)

```
 private:
  bool timer_running_ = false;
  uint16_t start_time_;
};

} // namespace kaleidoscope
} // namespace plugin

kaleidoscope::plugin::MyPlugin;
```

In the above example, the private member variable `start_time_` and the constant `timeout` are the same type of unsigned integer (`uint16_t`), and we've used the additional boolean `timer_running_` to keep from checking for timeouts when `start_time_` isn't valid. This plugin does something (unspecified) 500 milliseconds after a `Key_X` toggles on.

10.7.5 Creating additional events

Another thing we might want a plugin to do is generate "extra" events that don't correspond to physical state changes. An example of this is the Macros plugin, which might turn a single keypress into a series of HID reports sent to the host. Let's build a simple plugin to illustrate how this is done, by making a key type a string of characters, rather than a single one.

For the sake of simplicity, let's make the key `H` result in the string `Hi!` being typed (from the point of view of the host computer). To do this, we'll make a plugin with an `onKeyEvent()` handler (because we want it to respond to a particular keypress event), which will call `Runtime.handleKeyEvent()` to generate new events sent to the host.

The first thing we need to understand to do this is how to use the `KeyEvent()` constructor to create a new `KeyEvent` object. For example:

```
KeyEvent event = KeyEvent(KeyAddr::none(), IS_PRESSED, Key_H);
```

This creates a `KeyEvent` where `event.addr` is an invalid address that doesn't correspond to a physical keyswitch, `event.state` has only the `IS_PRESSED` bit set, but not `WAS_PRESSED`, which corresponds to a key toggle-on event, and `event.key` is set to `Key_H`.

We can then cause Kaleidoscope to process this event, including calling plugin handlers, by calling `Runtime.handleKeyEvent(event)`:

```
  EventHandlerResult onKeyEvent(KeyEvent &event) {
    if (event.key == Key_H && keyToggledOn(event.state)) {

      // Create and send the `H` (shift + h)
      KeyEvent new_event = KeyEvent(KeyAddr::none(), IS_PRESSED, LSHIFT(Key_H));
      Runtime.handleKeyEvent(new_event);

      // Change the key value and send `i`
      new_event.key = Key_I;
      Runtime.handleKeyEvent(new_event);

      // Change the key value and send `!` (shift + 1)
      new_event.key = LSHIFT(Key_1);
      Runtime.handleKeyEvent(new_event);
```

(continues on next page)

(continued from previous page)

```
    return EventHandlerResult::ABORT;
  }
  return EventHandlerResult::OK;
}
```

A few shortcuts were taken with this plugin that are worth pointing out. First, you may have noticed that we didn't send any key *release* events, just three presses. This works, but there's a small chance that it could cause problems for some plugin that's trying to match key presses and releases. To be nice (or pedantic, if you will), we could also send the matching release events, but this is probably not necessary in this case, because we've used an invalid key address (`KeyAddr::none()`) for these generated events. This means that Kaleidoscope will not be recording these events as held keys. If we had used valid key addresses (corresponding to physical keyswitches) instead, it would be more important to send matching release events to keep keys from getting "stuck" on. For example, we could just use the address of the `H` key that was pressed:

```
EventHandlerResult onKeyEvent(KeyEvent &event) {
  if (event.key == Key_H && keyToggledOn(event.state)) {

    KeyEvent new_event = KeyEvent(event.addr, IS_PRESSED, LSHIFT(Key_H));
    Runtime.handleKeyEvent(new_event);

    new_event.key = Key_I;
    Runtime.handleKeyEvent(new_event);

    new_event.key = LSHIFT(Key_1);
    Runtime.handleKeyEvent(new_event);

    return EventHandlerResult::ABORT;
  }
  return EventHandlerResult::OK;
}
```

This new version has the curious property that if the `H` key is held long enough, it will result in repeating `!!!!` characters on the host, until the key is released, which will clear it. In fact, instead of creating a whole new `KeyEvent` object, we could further simplify this plugin by simply modifying the `event` object that we already have, instead:

```
EventHandlerResult onKeyEvent(KeyEvent &event) {
  if (event.key == Key_H && keyToggledOn(event.state)) {
    event.key = LSHIFT(Key_H);
    Runtime.handleKeyEvent(event);

    event.key = Key_I;
    Runtime.handleKeyEvent(event);

    event.key = LSHIFT(Key_1);
  }
  return EventHandlerResult::OK;
}
```

Note that, with this version, we've only sent two extra events, then changed the `event.key` value, and returned `OK` instead of `ABORT`. This is basically the same as the above pluging that turned `Y` into `X`, but with two extra events sent first.

As one extra precaution, it would be wise to mark the generated event(s) as "injected" to let other plugins know that these events should be ignored. This is a convention that is used by many existing Kaleidoscope plugins. We do this by setting the `INJECTED` bit in the `event.state` variable:

```
EventHandlerResult onKeyEvent(KeyEvent &event) {
  if (event.key == Key_H && keyToggledOn(event.state)) {
    event.state |= INJECTED;

    event.key = LSHIFT(Key_H);
    Runtime.handleKeyEvent(event);

    event.key = Key_I;
    Runtime.handleKeyEvent(event);

    event.key = LSHIFT(Key_1);
  }
  return EventHandlerResult::OK;
}
```

If we wanted to be especially careful, we could also add the corresponding release events:

```
EventHandlerResult onKeyEvent(KeyEvent &event) {
  if (event.key == Key_H && keyToggledOn(event.state)) {
    event.key = LSHIFT(Key_H);
    event.state = INJECTED | IS_PRESSED;
    Runtime.handleKeyEvent(event);
    event.state = INJECTED | WAS_PRESSED;
    Runtime.handleKeyEvent(event);

    event.key = Key_I;
    event.state = INJECTED | IS_PRESSED;
    Runtime.handleKeyEvent(event);
    event.state = INJECTED | WAS_PRESSED;
    Runtime.handleKeyEvent(event);

    event.key = LSHIFT(Key_1);
    event.state = INJECTED | IS_PRESSED;
  }
  return EventHandlerResult::OK;
}
```

Avoiding infinite loops

One very important consideration for any plugin that calls `Runtime.handleKeyEvent()` from an `onKeyEvent()` handler is recursion. `Runtime.handleKeyEvent()` will call all plugins' `onKeyEvent()` handlers, including the one that generated the event. Therefore, we need to take some measures to short-circuit the resulting recursive call so that our plugin doesn't cause an infinite loop.

Suppose the example plugin above was changed to type the string `hi!` instead of `Hi!`. When sending the first generated event, with `event.key` set to `Key_H`, our plugin would recognize that event as one that should be acted on, and make another call to `Runtime.handleKeyEvent()`, which would again call `MyPlugin.onKeyEvent()`, and so on until the MCU ran out of memory on the stack.

The simplest mechanism used by many plugins that mark their generated events "injected" is to simply ignore all generated events:

```
  EventHandlerResult onKeyEvent(KeyEvent &event) {
    if ((event.state & INJECTED) != 0)
      return EventHandlerResult::OK;

    if (event.key == Key_H && keyToggledOn(event.state)) {
      event.state |= INJECTED;

      event.key = LSHIFT(Key_H);
      Runtime.handleKeyEvent(event);

      event.key = Key_I;
      Runtime.handleKeyEvent(event);

      event.key = LSHIFT(Key_1);
    }
    return EventHandlerResult::OK;
  }
```

There are other techniques to avoid inifinite loops, employed by plugins whose injected events should be processed by other plugins, but since most of those will be using the `onKeyswitchEvent()` handler instead of `onKeyEvent()`, we'll cover that later in this guide.

10.7.6 Physical keyswitch events

Most plugins that respond to key events can do their work using the `onKeyEvent()` handler, but in some cases, it's necessary to use the `onKeyswitchEvent()` handler instead. These event handlers are strictly intended for physical keyswitch events, and plugins that implement the `onKeyswitchEvent()` handler must abide by certain rules in order to work well with each other. As a result, such a plugin is a bit more complex, but there are helper mechanisms to make things easier:

```
#include "kaleidoscope/KeyEventTracker.h"

namespace kaleidoscope {
namespace plugin {

class MyKeyswitchPlugin : public Plugin {
 public:
  EventHandlerResult onKeyswitchEvent(KeyEvent &event) {
    if (event_tracker_.shouldIgnore(event))
      return EventHandlerResult::OK;
    // Plugin logic goes here...
    return EventHandlerResult::OK;
  }
 private:
  KeyEventTracker event_tracker_;
};

} // namespace kaleidoscope
} // namespace plugin

kaleidoscope::plugin::MyKeyswitchPlugin;
```

We've just added a `KeyEventTracker` object to our plugin, and made the first line of its `onKeyswitchEvent()`

handler call that event tracker's `shouldIgnore()` method, returning `OK` if it returns `true` (thereby ignoring the event). Every plugin that implements `onKeyswitchEvent()` should follow this template to avoid plugin interaction bugs, including possible infinite loops.

The main reason for this event tracker mechanism is that plugins with `onKeyswitchEvent()` handlers often delay events because some aspect of those events (usually `event.key`) needs to be determined by subsequent events or timeouts. To do this, event information is stored, and the event is later regenerated by the plugin, which calls `Runtime.handleKeyswitchEvent()` so that the other `onKeyswitchEvent()` handlers can process it.

We need to prevent infinite loops, but simply marking the regenerated event `INJECTED` is no good, because it would prevent the other plugins from acting on it, so we instead keep track of a monotonically increasing event id number and use the `KeyEventTracker` helper class to ignore events that our plugin has already recieved, so that when the plugin regenerates an event with the same event id, it (and all the plugins before it) can ignore that event, but the subsequent plugins, which haven't seen that event yet, will recongize it as new and process the event accordingly.

Regenerating stored events

When a plugin that implements `onKeyswitchEvent()` regenerates a stored event later so that it can be processed by the next plugin in the chain, it must use the correct event id value (the same one used by the original event). This is an object of type `EventId`, and is retrieved by calling `event.id()` (unlike the other properties of a `KeyEvent` object the event id is not directly accessible).

```
KeyEventId stored_id = event.id();
```

When reconstructing an event to allow it to proceed, we then use the four-argument version of the `KeyEvent` constructor:

```
KeyEvent event = KeyEvent(addr, state, key, stored_id);
```

In the above, `addr` and `state` are usually also the same as the original event's values, and `key` is most often the thing that changes. If your plugin wants a keymap lookup to take place, the value `Key_Undefined` can be used instead of explicitly doing the lookup itself.

10.7.7 Controlling LEDs

10.7.8 HID reports

10.7.9 Layer changes

10.8 Bundled plugins

10.8.1 AutoShift

AutoShift allows you to type shifted characters by long-pressing a key, rather than chording it with a modifier key.

Using the plugin

Using the plugin with its defaults is as simple as including the header, and enabling the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-AutoShift.h>

KALEIDOSCOPE_INIT_PLUGINS(AutoShift);
```

With AutoShift enabled, when you first press a key that AutoShift acts on, its output will be delayed. If you hold the key down long enough, you will see the shifted symbol appear in the output. If you release the key before the timeout, the output will be unshifted.

Turning AutoShift on and off

The `AutoShift` object provides three methods for turning itself on and off:

- To turn the plugin on, call `AutoShift.enable()`.
- To turn the plugin off, call `AutoShift.disable()`.
- To toggle the plugin's state, call `AutoShift.toggle()`.

Note: Disabling the AutoShift plugin does not affect which `Key` categories it will affect when it is re-enabled.

Setting the AutoShift long-press delay

To set the length of time AutoShift will wait before adding the `shift` modifier to the key's output, use `AutoShift.setTimeout(t)`, where `t` is a number of milliseconds.

Configuring which keys get auto-shifted

AutoShift provides a set of key categories that can be independently added or removed from the set of keys that will be auto-shifted when long-pressed:

- `AutoShift.letterKeys()`: Letter keys
- `AutoShift.numberKeys()`: Number keys (number row, not numeric keypad)
- `AutoShift.symbolKeys()`: Other printable symbols
- `AutoShift.arrowKeys()`: Navigational arrow keys
- `AutoShift.functionKeys()`: All function keys (F1-F24)
- `AutoShift.printableKeys()`: Letters, numbers, and symbols
- `AutoShift.allKeys()`: All non-modifier USB Keyboard keys

These categories are restricted to USB Keyboard-type keys, and any modifier flags attached to the key is ignored when determining if it will be auto-shifted. Any of the above expressions can be used as the `category` parameter in the functions described below.

- To include a category in the set that will be auto-shifted, call `AutoShift.enable(category)`
- To remove a category from the set that will be auto-shifted, call `AutoShift.disable(category)`

- To set the full list of categories that will be auto-shifted, call `AutoShift.setEnabled(categories)`, where `categories` can be either a single category from the above list, or list of categories combined using the `|` (bitwise-or) operator (e.g. `AutoShift.setEnabled(AutoShift.letterKeys() | AutoShift.numberKeys())`).

Advanced customization of which keys get auto-shifted

If the above categories are not sufficient for your auto-shifting needs, it is possible to get even finer-grained control of which keys are affected by AutoShift, by overriding the `isAutoShiftable()` method in your sketch. For example, to make AutoShift only act on keys `A` and `Z`, include the following code in your sketch:

```
bool AutoShift::isAutoShiftable(Key key) {
  if (key == Key_A || key == key_Z)
    return true;
  return false;
}
```

As you can see, this method takes a `Key` as its input and returns either `true` (for keys eligible to be auto-shifted) or `false` (for keys AutoShift will leave alone).

Plugin compatibility

If you're using AutoShift in a sketch that also includes the Qukeys and/or SpaceCadet plugins, make sure to register AutoShift after those other plugins in order to prevent auto-shifts from getting clobbered. The recommended order is as follows:

```
KALEIDOSCOPE_INIT_PLUGINS(Qukeys, SpaceCadet, AutoShift)
```

It's not generally recommended to use AutoShift on the same key(s) handled by either Qukeys or SpaceCadet, as this can result in confusing behaviour.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.2 CharShift

CharShift allows you to independently assign symbols to shifted and unshifted positions of keymap entries. Either or both symbols can be ones that normally requires the `shift` modifier, and either or both symbols can be ones normally produced without it.

For example you can configure your keyboard so that a single key produces `,` when pressed unshifted, but `;` when pressed with `shift` held. Or `(` unshifted, and `[` shifted. Or +/* — all without changing your OS keyboard layout.

Using the plugin

Using the plugin with its defaults is as simple as including the header, and enabling the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-CharShift.h>

KALEIDOSCOPE_INIT_PLUGINS(CharShift);
```

Further configuration is required, of course; see below.

Note: CharShift should be registered in `KALEIDOSCOPE_INIT_PLUGINS()` after any plugin that changes the event's `Key` value to that of an CharShift key.

Configuring CharShift keys

To use CharShift, we must first define `KeyPair` objects, which can then be referenced by entries in the keymap. This is easiest to do by using the `CS_KEYS()` preprocessor macro in the sketch's `setup()` function, as follows:

```
void setup() {
  Kaleidoscope.setup();
  CS_KEYS(
    kaleidoscope::plugin::CharShift::KeyPair(Key_Comma, Key_Semicolon),          ␣
↪ // `,`/`;`
    kaleidoscope::plugin::CharShift::KeyPair(Key_Period, LSHIFT(Key_Semicolon)), ␣
↪ // `.`/`:`
    kaleidoscope::plugin::CharShift::KeyPair(LSHIFT(Key_9), Key_LeftBracket),    ␣
↪ // `(`/`[`
    kaleidoscope::plugin::CharShift::KeyPair(LSHIFT(Key_Comma), LSHIFT(Key_LeftBracket)),
↪ // `<`/`{`
  );
}
```

The first argument to the `KeyPair()` constructor is the value for when the key is pressed without `shift` held, the second is what you'll get if a `shift` modifier is being held when the key toggles on. If that second (”upper”) value doesn't have the `shift` modifier flag (i.e. `LSHIFT()`) applied to it, the held `shift` modifier will be suppressed when the key is pressed, allowing the "unshifted" symbol to be produced.

These `KeyPairs` can be referred to in the sketch's keymap by using the `CS()` preprocessor macro, which takes an integer argument, referring to items in the `CS_KEYS()` array, starting with zero. With the example above, an entry of `CS(2)` will output `(` when pressed without `shift`, and `[` if `shift` is being held.

Adding CharShift keys in Chrysalis

As of this writing, CharShift keys can't be defined in Chrysalis; they can only be defined in a custom sketch (see above). This doesn't mean that you can't use them in Chrysalis-defined keymaps, however. To add a CharShift key in Chrysalis, select `Custom key code`, and add the offset `53631` to the index number of the CharShift key.

In other words, where you would use `CS(2)` in a Kaleidoscope sketch, you would need to use `53633` (`53631 + 2`) as the custom key code in Chrysalis. Any CharShift keys referenced in this way still need to be defined in a custom Kaleidoscope sketch (see above), but they can still be used in a Chrysalis keymap.

In general, the formula for the Chrysalis custom key code corresponding to the CharShift key with index `N` is:

`CS(N) 53631 + N`

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.3 Chord

Concept

This Kaleidoscope plugin allows you to define a chord of keys on your keyboard which, when pressed simultaneously, produce a single keycode. This differs from MagicCombo in that the individual keys making up a chord are suppressed, producing only the singular result.

Setup

- Include the header file:

```
#include <Kaleidoscope-Chord.h>
```

- Use the plugin in the `KALEIDOSCOPE_INIT_PLUGINS` macro:

```
KALEIDOSCOPE_INIT_PLUGINS(Chord);
```

And define some chords in `setup` such as:

```
CHORDS(
  CHORD(Key_J, Key_K), Key_Escape,
  CHORD(Key_D, Key_F), Key_LeftShift,
  CHORD(Key_S, Key_D), TOPSY(Semicolon),
  CHORD(Key_S, Key_D, Key_F), Key_Spacebar,
)
```

As can be seen from the example, chords can be overlapping or subsets of each other, and can result in regular keys, modifier keys or special keys (such as a TopsyTurvey key). The resulting key will be held for as long as the last key pressed in the chord is held.

Configuration

`.setTimeout(timeout)`

Sets the time (in milliseconds) after which a key or set of keys that could be part of a larger chord is pressed before the pressed keys are resolved. It's generally not necessary to explicitly wait for this timeout, since as soon as a key is pressed that could not be part of a chord with existing key presses, the existing keys will resolve. For instance, with the example above, pressing and holding S, D, L in quick succession would result in a held Shift + L. It's only if you wanted to type Shift + F, that you'd need to add a pause (S, D, wait for timeout, F), since otherwise it would be interpreted as a space.

Defaults to `50`.

Further reading

The *example* can help to learn how to use this plugin.

10.8.4 Colormap

The `Colormap` extension provides an easier way to set up a different - static - color map per-layer. This means that we can set up a map of colors for each key, on a per-layer basis, and whenever a layer becomes active, the color map for that layer is applied. Colors are picked from a 16-color palette, provided by the LED-Palette-Theme plugin. The color map is stored in `EEPROM`, and can be easily changed via the FocusSerial plugin, which also provides palette editing capabilities.

It is also possible to set up a default palette and colormap, using the `DefaultColormap` plugin, also provided by this package. See below for its documentation.

Using the extension

To use the extension, include the header, tell it the number of layers you have, register the `Focus` hooks, and it will do the rest. We'll also set up a default for both the palette, and the colormap.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-Colormap.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-LED-Palette-Theme.h>

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          LEDControl,
                          LEDPaletteTheme,
                          ColormapEffect,
                          DefaultColormap,
                          Focus);

PALETTE(
 /* A list of 16 cRGB colors... */
)

COLORMAPS(
 [0] = COLORMAP(
  // List of palette indexes for each key, using the same layout
  // as the `KEYMAP` macro does for keys.
 ),
 [1] = COLORMAP_STACKED(
  // List of palette indexes for each key, using the same layout
  // as the `KEYMAP_STACKED` macro does for keys.
 )
)

void setup() {
  Kaleidoscope.setup();
```

(continues on next page)

(continued from previous page)

```
  ColormapEffect.max_layers(1);
  DefaultColormap.setup();
}
```

The PALETTE and COLORMAPS macros are only used for the DefaultColormap plugin, ColormapEffect itself makes no use of them. The PALETTE must always contain a full 16-color palette. COLORMAPS can define colormaps for as many layers as one wishes, but the DefaultColormap plugin will only copy over as many as ColormapEffect is configured to support.

Plugin methods

The extension provides an ColormapEffect singleton object, with a single method:

.max_layers(max)

Tells the extension to reserve space in EEPROM for up to max layers. Can only be called once, any subsequent call will be a no-op.

Also provided is an optional DefaultColormap plugin, with two methods:

.setup()

Intended to be called from the setup() method of the sketch, it checks if the ColormapEffect plugin is initialized, and if not, then copies the palette and the colormap over from the firmware to EEPROM.

.install()

Same as .setup() above, but without the initialized check. Intended to be used when one wants to restore the colormap to factory settings.

Focus commands

colormap.map

Without arguments, prints the color map: palette indexes for all layers.

With arguments, updates the color map with new indexes. One does not need to give the full map, the plugin will process as many arguments as available, and ignore anything past the last key on the last layer (as set by the .max_layers() method).

If the DefaultColormap plugin is also in use, an additional focus command is made available:

`colormap.install`

Copies the default colormap and palette built into the firmware into EEPROM, effectively performing a factory reset for both.

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial
- Kaleidoscope-LED-Palette-Theme

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.5 Colormap-Overlay

This plugin provides an easier way to apply color to specific combinations of keys and layers, regardless of the active LED mode. Colors are picked from a 16-color palette, provided by the LED-Palette-Theme plugin. The overlays are stored in `EEPROM`, and can be easily changed via the FocusSerial plugin, which also provides palette editing capabilities.

It is also possible to set up a default palette and overrides, using the `PALETTE` macro provided by the LED-Palette-Theme package and the `COLORMAP_OVERLAYS` provided by this package. See below for its documentation.

Using the extension

To use the extension, include the headers and, optionally, register the `Focus` hooks. Use the macros mentioned above to set up a default for both the palette and colormap overlays. Note that layers and key addresses are all zero-indexed, and key addresses rows are top to bottom and columns are left to right. For the key coordinates refer to the relevant header file:

- Model 01
- Model 100
- Atreus
- Imago

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Palette-Theme.h>
#include <Kaleidoscope-Colormap-Overlay.h>
#include <Kaleidoscope-FocusSerial.h>

PALETTE(
 /* A list of 16 cRGB colors... */
)

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          LEDControl,
                          ColormapOverlay,
```

(continues on next page)

(continued from previous page)

```
                          DefaultPalette,
                          Focus);

void setup() {
  Kaleidoscope.setup();

  COLORMAP_OVERLAYS(
    // List of overlays, using kaleidoscope::plugin::Overlay
    // kaleidoscope::plugin::Overlay({layer}, {key address}, {palette index})
    // A special value `ColormapOverlay::layer_wildcard` can be used in place of
    // a layer number to apply the color overlay on all layers
  )

  ColormapOverlay.setup();
  DefaultPalette.setup();
}
```

Plugin methods

The extension only has a single method:

`.setup()`

Intended to be called from the `setup()` method of the sketch, it reserves the required space in EEPROM.

10.8.6 Cycle

If you ever wanted a key that works like keys on old cell phones, when you press a key and it cycles through a number of options in a sequence, then the cycling key is what you are looking for. It is a bit different than on cell phones of old, as it is a separate key, that works in combination of other keys: you press a key, then the cycle key, and the cycle key will replace the previously input symbol with another. Keep tapping the cycle key, and it will replace symbols with new ones, in a loop.

Using the plugin

To use the plugin, we need to include the header, and declare the behaviour used. Then, we need to place a cycle key or two on the keymap. And finally, we need to implement the *cycleAction* function that gets called each time the cycling key triggers.

```
#include <Kaleidoscope-Cycle.h>

// Somewhere in the keymap:
Key_Cycle

// later in the Sketch:
void cycleAction(Key previous_key, uint8_t cycle_count) {
  bool is_shifted = previous_key.getFlags() & SHIFT_HELD;
  if (previous_key.getKeyCode() == Key_A.getKeyCode() && is_shifted)
```

(continues on next page)

(continued from previous page)

```
      cycleThrough (LSHIFT(Key_A), LSHIFT(Key_B), LSHIFT(Key_C));
  if (previous_key.getKeyCode() == Key_A.getKeyCode() && !is_shifted)
      cycleThrough (Key_A, Key_B, Key_C);
}

KALEIDOSCOPE_INIT_PLUGINS(Cycle);

void setup() {
  Kaleidoscope.setup();
}
```

Keymap markup

`Key_Cycle`

The key code for the cycle key. There can be as many of this on the keymap, as many one wants, but they all behave the same. There is little point in having more than one on each side.

Plugin methods

The plugin provides a `Cycle` object, but to implement the actions, we need to define a function (*`cycleAction`*) outside of the object. A handler, of sorts. The object also provides a helper method to replace the previous symbol with another. The plugin also provides one macro that is particularly useful, and in most cases, should be used over the `.replace()` method explained below.

`cycleThrough(keys...)`

Cycles through all the possibilities given in `keys` (starting from the beginning once it reached the end). This should be used from the *`cycleAction`* function, once it is determined what sequence to cycle through.

To make the cycling loop complete, the first element of the `keys` list should be the one that - when followed by the Cycle key - triggers the action.

`.replace(key)`

Deletes the previous symbol (by sending a `Backspace`), and inputs the new one. This is used by `cycleThrough()` above, behind the scenes.

The recommended method is to use the macro, but in special circumstances, this function can be of direct use as well.

Overrideable methods

`cycleAction(previous_key, cycle_count)`

The heart and soul of the plugin, that must be defined in the Sketch. It will be called whenever the cycling key triggers, and the two arguments are the last key pressed (not counting repeated taps of the cycling key itself), and the number of times the cycling key has been pressed.

It is up to us to decide what to do, and when. But the most common - and expected - action is to call `cycleThrough()` with a different sequence for each key we want to use together with the cycling key.

Dependencies

- Kaleidoscope-Ranges

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.7 CycleTimeReport

A development and debugging aid, this plugin will measure average mainloop times (in microseconds) and print it to `Serial` periodically. While not the most reliable way to measure the speed of processing, it gives a reasonable indication nevertheless.

Using the plugin

The plugin comes with reasonable defaults (see below), and can be used out of the box, without any further configuration:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-CycleTimeReport.h>

KALEIDOSCOPE_INIT_PLUGINS(CycleTimeReport);

void setup () {
  Kaleidoscope.serialPort().begin(9600);
  Kaleidoscope.setup ();
}
```

Plugin methods

The plugin provides a single object, `CycleTimeReport`, with the following methods:

`.setReportInterval(interval)`

Sets the length of time between reports to `interval` milliseconds. The default is `1000`, so it will report once per second.

`.report(mean_cycle_time)`

Reports the average (mean) cycle time since the previous report. This method is called automatically, once per report interval (see above). By default, it does so over `Serial`.

It can be overridden, to change how the report looks, or to make the report toggleable, among other things.

It takes no arguments, and returns nothing.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.8 DefaultLEDModeConfig

The `DefaultLEDModeConfig` plugin provides a way to set a default LED mode, the LED mode the device starts up with active, via Focus.

By default the first LED mode enabled will be the active one, unless set otherwise in `setup()`. To make this configurable, without having to reorder the LED modes, this plugin provides the necessary tools to accomplish that.

Using the plugin

The example below shows how to use the plugin, including setting up a LED mode other than the first to use as a default in case EEPROM is uninitialized.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-DefaultLEDModeConfig.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>
#include <Kaleidoscope-FocusSerial.h>

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          LEDControl,
                          LEDOff,
                          LEDRainbowEffect,
                          LEDRainbowWaveEffect,
                          Focus,
                          DefaultLEDModeConfig);

void setup() {
  Kaleidoscope.setup();

  DefaultLEDModeConfig.activateLEDModeIfUnconfigured(
    &LEDRainbowWaveEffect
```

(continues on next page)

(continued from previous page)

```
  );
}
```

Plugin methods

The plugin provides a singleton object called `DefaultLEDModeConfig`, with a single method:

`.activateLEDModeIfUnconfigured(&LEDModePlugin)`

> Activates the LED mode pointed to by `&LEDModePlugin` if and only if the EEPROM slice of the plugin is unconfigured. This lets us set a default LED mode without persisting it into storage, or hard-coding it.

Focus commands

`led_mode.default`

> Without arguments, prints the default LED mode's index.
>
> If an argument is given, it must be the index of the LED mode we wish to set as the default.

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

10.8.9 Kaleidoscope-Devel-ArduinoTrace

A development and debugging aid, this plugin imports and initializes an embedded copy of the ArduinoTrace library from https://github.com/bblanchon/ArduinoTrace

It is primarly intended for use on our simulator, though in theory, it should work when run on normal hardware, too

Using the plugin

The plugin comes with reasonable defaults (see below), and can be used out of the box, without any further configuration:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Devel-ArduinoTrace.h>

/* ... */

void setup () {
  Kaleidoscope.setup ();
  TRACE()
}
```

(continues on next page)

(continued from previous page)

```
void someMethod(uint8_t value) {
        uint8_t other_value;

        TRACE()
        DUMP(value)
        other_value = someOtherMethod(value);
        DUMP(other_value)
}
```

Running in the simulator, you should see output like:

```
basic-keypress.ino:492: void setup()
Runtime.cpp:51: void kaleidoscope::Runtime_::loop()
Runtime.cpp:53: millis_at_cycle_start_ = 4
```

While this plugin is primarily intended to be used in the Kaleidoscope simulator, it should work on actual hardware. On the simulator, output is directed to DebugStderr. On hardware, it defaults to Serial.

To configure ArduinoTrace, there are a number of constants you can `#define` before you `#include` the plugin. They're documented upstream.

Plugin methods

This plugin does not itself offer up any API methods or use any plugin hooks, instead exposing the "TRACE" and "DUMP" macros provided by ArduinoTrace

Further reading

Have a look at the docs for ArduinoTrace on GitHub.

10.8.10 DynamicMacros

Dynamic macros are similar to Macros, but unlike them, they can be re-defined without compiling and flashing new firmware: one can change dynamic macros via Focus, using a tool like Chrysalis.

Dynamic macros come with certain limitations, however: unlike the built-in macros, dynamic ones do not support running custom code, they can only play back a sequence of events (keys, mousekeys, etc), and do so whenever one presses the dynamic macro key.

You can define up to 32 dynamic macros, there is no limit on their length, except the amount of storage available on the keyboard.

Using the plugin

To use the plugin, we need to include the header, initialize the plugin with `KALEIDOSCOPE_INIT_PLUGINS()`, and reserve storage space for the macros. This is best illustrated with an example:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROMSettings.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-DynamicMacros.h>

KALEIDOSCOPE_INIT_PLUGINS(
  EEPROMSettings,
  Focus,
  DynamicMacros
);

void setup() {
  Kaleidoscope.setup();

  DynamicMacros.reserve_storage(128);
}
```

Keymap markup

`DM(id)`

Places a dynamic macro key on the keymap, with the `id` number (0 to 31) as identifier. Pressing the key will immediately run the associated dynamic macro.

Plugin methods

The plugin provides a `DynamicMacros` object, with the following methods and properties available:

`.reserve_storage(size)`

Reserves `size` bytes of storage for dynamic macros. This must be called from the `setup()` method of your sketch, otherwise dynamic macros will not function.

`.play(macro_id)`

Plays back a macro, specified by `macro_id`.

`MACRO` steps

The plugin supports the same macro steps as the Macros plugin, please refer to the documentation therein.

Focus commands

The plugin provides two Focus commands: `macros.map` and `macros.trigger`.

`macros.map [macros...]`

Without arguments, displays all the stored macros. Each macro is terminated by an end marker (`MACRO_ACTION_END`), and the last macro is followed by an additional marker. The plugin will send back the entire dynamic macro storage space, even the data after the final marker.

With arguments, it replaces the current set of dynamic macros with the newly given ones. Macros are terminated by an end marker, and the last macro must be terminated by an additional one.

In both cases, the data sent or expected is a sequence of 8-bit values, a memory dump.

`macros.trigger macro_id`

Runs the dynamic macro associated with `macro_id` immediately. This can be used to test macros without having to place them on the keymap.

Dependencies

- Kaleidoscope-MacroSupport
- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

10.8.11 DynamicTapDance

The `DynamicTapDance` plugin allows one to set up TapDance keys without the need to compile and flash new firmware: one can change dynamic dances via Focus, using a tool like Chrysalis.

Dynamic dances come with certain limitations, however: unlike the built-in ones, dynamic ones do not support running custom code. They can only choose a key from a list of possibilities. Given a list of keys, the plugin will choose the one corresponding to the number of taps on the key, just like `TapDance` itself does.

Basically, this plugin allows us to store `tapDanceActionKeys` key lists in the on-board memory of our keyboard.

You can define up to 16 dynamic dances, there is no limit on their length, except the amount of storage available on the keyboard. You can even mix them with built-in dances! But the total number of tap-dances is 16.

Using the plugin

To use the plugin, we need to include the header, tell the firmware to `use` the plugin, and reserve storage space for the dances. This is best illustrated with an example:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROMSettings.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-TapDance.h>
#include <Kaleidoscope-DynamicTapDance.h>

KALEIDOSCOPE_INIT_PLUGINS(
  EEPROMSettings,
  Focus,
  TapDance,
  DynamicTapDance
);

void tapDanceAction(uint8_t tap_dance_index, KeyAddr key_addr, uint8_t tap_count,␣
↪kaleidoscope::plugin::TapDance::ActionType tap_dance_action) {
  DynamicTapDance.dance(tap_dance_index, key_addr, tap_count, tap_dance_action);
}

void setup() {
  Kaleidoscope.setup();

  // 0 is the amount of built-in dances we have.
  // 128 is how much space (in bytes) we reserve for dances.
  DynamicTapDance.setup(0, 128);
}
```

Plugin methods

The plugin provides a `DynamicTapDance` object, with the following methods and properties available:

`.setup(builtin_dances, size)`

Reserves `size` bytes of storage for dynamic dances. This must be called from the `setup()` method of your sketch, otherwise dynamic tap-dances will not function.

The `builtin_dances` argument tells the plugin how many built-in dances there are.

`.dance(index, key_addr, tap_count, tap_dance_action)`

Performs a given dance (`index`) made on the key at `key_addr` address, which has been tapped `tap_count` times, and the action to perform is `tap_dance_action`.

This mirrors the overrideable `tapDanceAction()` method of TapDance, and is intended to be called from therein.

Focus commands

The plugin provides one Focus command: `tapdance.map`.

`tapdance.map [dances...]`

Without arguments, displays all the stored dances. Each dance is terminated by an end marker (`0`, aka `Key_NoKey`), and the last dance is followed by an additional marker. The plugin will send back the entire dynamic tap-dance storage space, even data after the final marker.

With arguments, it replaces the current set of dynamic dances with the newly given ones. Dances are terminated by an end marker, and the last dance must be terminated by an additional one. It is up to the caller to make sure these rules are obeyed.

In both cases, the data sent or expected is a sequence of 16-bit values, a memory dump.

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial
- Kaleidoscope-TapDance

10.8.12 EEPROM-Keymap

While keyboards usually ship with a keymap programmed in, to be able to change that keymap, without flashing new firmware, we need a way to place the keymap into a place we can update at run-time, and which persists across reboots. Fortunately, we have a bit of `EEPROM` on the keyboard, and can use it to store either the full keymap (and saving space in the firmware then), or store additional layers there.

In short, this plugin allows us to change our keymaps, without having to compile and flash new firmware. It does so through the use of the FocusSerial plugin.

By default, the plugin extends the keymap in PROGMEM: it will only look for keys in EEPROM if looking up from a layer that's higher than the last one in PROGMEM. This behaviour can be changed either via `Focus` (see below), or by calling `EEPROMSettings.use_eeprom_layers_only` (see the EEPROMSettings documentation for more information).

Using the plugin

Using the plugin is reasonably simple: after including the header, enable the plugin, and configure how many layers at most we want to store in `EEPROM`. There are other settings one can tweak, but these two steps are enough to get started with.

Once these are set up, we can update the keymap via Focus.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Keymap.h>
#include <Kaleidoscope-FocusSerial.h>

KALEIDOSCOPE_INIT_PLUGINS(EEPROMKeymap,
                          Focus);

void setup() {
  Kaleidoscope.setup();

  EEPROMKeymap.setup(1);
}
```

Plugin methods

The plugin provides the `EEPROMKeymap` object, which has the following method:

`.setup(layers)`

Reserve space in EEPROM for up to `layers` layers, and set up the key lookup mechanism.

Focus commands

The plugin provides three Focus commands: `keymap.default`, `keymap.custom`, and `keymap.useCustom`.

`keymap.default`

Display the default keymap from PROGMEM. Each key is printed as its raw, 16-bit keycode.

Unlike `keymap.custom`, this does not support updating, because PROGMEM is read-only.

`keymap.custom [codes...]`

Without arguments, display the custom keymap stored in EEPROM. Each key is printed as its raw, 16-bit keycode.

With arguments, it updates as many keys as given. One does not need to set all keys, on all layers: the command will start from the first key on the first layer (in EEPROM, which might be different than the first layer!), and go on as long as it has input. It will not go past the number of layers in EEPROM.

`keymap.onlyCustom [0|1]`

Without arguments, returns whether the firmware uses both the default and the custom layers (the default, `0`) or custom (EEPROM-stored) layers only (`1`).

With an argument, sets whether to use custom layers only, or extend the built-in layers instead.

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.13 EEPROM-Keymap-Programmer

Inspired by a similar feature on other keyboards, the `EEPROM-Keymap-Programmer` plugin implements an on-device keymap re-arrangement / re-coding system. There are two modes of operation: in one, we need to press a key we want to change, then another to copy from. In the other, we press a key to change, and then input a key code (terminated by any non-number key).

The two modes of operation

It is worth looking at the two separately, to better understand how they work, and what they accomplish:

Copy mode

In `COPY` mode, the plugin will use both the built-in, default keymap, and the override stored in `EEPROM`. When we select a key to override, we need to tap another, which will be used as the source. The source key's code will be looked up from the built-in keymap. For example, lets say we want to swap `A` and `B` for some odd reason. We can do this by triggering the keymap programmer mode, then tapping `A` to select it as the destination, then `B` as the source. The plugin will look up the keycode in the built-in keymap for the key in `B`'s location, and replace the location of `A` in the override with it. Next, we press the `B` key to select it as the destination, and we press the key that used to be `A` (but is now `B` too) to select it as a source. Because source keys are looked up in the built-in keymap, the plugin will find it is `A`.

Obviously, this method only works if we have a built-in keymap, and it does not support copying from another layer. It is merely a way to rearrange simple things, like alphanumerics.

Code mode

In CODE mode, instead of selecting a source key, we need to enter a code: press numbers to input the code, and any non-number key to end the sequence. Thus, when entering keymap programmer mode, and selecting, say, the A key, then tapping 5 SPACE will set the key to B (which has the keycode of 5).

This allows us to use keycodes not present on the built-in keymap, at the expense of having to know the keycode, and allowing no mistakes.

Using the plugin

Adding the functionality of the plugin to a Sketch is easier the usage explained above, though it requires that the EEPROM-Keymap plugin is also used, and set up appropriately.

Once the prerequisites are dealt with, all we need to do is to use the plugin, and find a way to trigger entering the keymap programmer mode. One such way is to use a macro, as in the example below:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Keymap.h>
#include <Kaleidoscope-EEPROM-Keymap-Programmer.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-Macros.h>

const macro_t *macroAction(uint8_t macroIndex, uint8_t keyState) {
  if (macroIndex == 0 && keyToggledOff(keyState)) {
    EEPROMKeymapProgrammer.activate();
  }

  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          EEPROMKeymapProgrammer,
                          EEPROMKeymap,
                          Macros);

void setup() {
  Kaleidoscope.setup();

  Layer.getKey = EEPROMKeymap.getKey;

  EEPROMKeymap.max_layers(1);
  EEPROMSettings.seal();
}
```

The plugin should be used as early as possible, otherwise other plugins that hook into the event system may start processing events before the programmer can take over.

Plugin methods

The plugin provides the `EEPROMKeymapProgrammer` object, which has the following methods and properties:

`.activate()`

Activates the keymap programmer. This is the function one needs to call from - say - a macro, to enter the edit state.

`.mode`

Set this property to the mode to use for editing: either `kaleidoscope::EEPROMKeymapProgrammer::COPY`, or `kaleidoscope::EEPROMKeymapProgrammer::CODE`.

Defaults to `kaleidoscope::EEPROMKeymapProgrammer::CODE`.

Focus commands

The plugin provides a single `Focus` hook: `FOCUS_HOOK_KEYMAP_PROGRAMMER`, which in turn provides the following command:

`keymap.toggleProgrammer`

Toggles the programmer mode on or off.

Dependencies

- Kaleidoscope-EEPROM-Keymap

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.14 EEPROM-Settings

To be able to reliably store persistent configuration in `EEPROM`, we need to be able to split up the available space for plugins to use. We also want to make sure that we notice when the `EEPROM` contents and the firmware are out of sync. This plugin provides the tools to do that.

It does not guard against errors, it merely provides the means to discover them, and let the firmware Sketch handle the case in whatever way it finds reasonable. It's a building block, and not much else. All Kaleidoscope plugins that need to store data in `EEPROM` are encouraged to make use of this library.

Using the plugin

There are a few steps one needs to take to use the plugin: we must first register it, then either let other plugins request slices of EEPROM, or do so ourselves. And finally, seal it, to signal that we are done setting up. At that point, we can verify whether the contents of the EEPROM agree with our firmware.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>

static uint16_t settingsBase;
static struct {
  bool someSettingFlag;
} testSettings;

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings, /* Other plugins that use EEPROM... */);

void setup () {
  Kaleidoscope.setup();

  settingsBase = EEPROMSettings.requestSlice(sizeof(testSettings));

  EEPROMSettings.seal();

  if (!EEPROMSettings.isValid()) {
    // Handle the case where the settings are out of sync...
    // Flash LEDs, for example.

    return;
  }

  Kaleidoscope.storage().get(settingsBase, testSettings);
}
```

Plugin methods

The plugin provides the EEPROMSettings object, which has the following methods:

`requestSlice(size)`

Requests a slice of the `EEPROM`, and returns the starting address (or 0 on error, including when the request arrived after sealing the layout).

Should only be called **before** calling `seal()`.

`default_layer([id])`

Sets (or returns, if called without an ID) the default layer. When the keyboard boots up, it will automatically switch to the configured layer - if any.

Setting it to `126` or anything higher disables the automatic switching.

`ignoreHardcodedLayers([true|false])`

Controls whether the hardcoded layers (in `PROGMEM`) are ignored or not.

When not ignored, the custom layes (in `EEPROM`) extend the hardcoded ones. When ignored, they replace the hardcoded set.

Returns the setting if called without arguments, changes it to the desired value if called with a boolean flag.

This setting is exposed to Focus via the `keymap.onlyCustom` command implemented by the [EEPROM-Keymap][EEPROM-Keymap.md] plugin.

Defaults to `false`.

`seal()`

Seal the `EEPROM` layout, so no new slices can be requested. The CRC checksum is considered final at this time, and the `isValid()`, `crc()`, `used()` and `version()` methods can be used from this point onwards.

If not called explicitly, the layout will be sealed automatically after `setup()` in the sketch finished.

`update()`

Updates the `EEPROM` header with the current status quo, including the version and the CRC checksum.

This should be called when upgrading from one version to another, or when fixing up an out-of-sync case.

`isValid()`

Returns whether the `EEPROM` header is valid, that is, if it has the expected CRC checksum.

Should only be called after calling `seal()`.

`invalidate()`

Invalidates the `EEPROM` header. Use when the version does not match what the firmware would expect. This signals to other plugins that the contents of `EEPROM` should not be trusted.

`version()`

Returns the current version of the EEPROM settings. It's the version of the settings only, not that of the whole layout - the CRC covers that.

This is for internal use only, end-users should not need to care about it.

`crc()`

Returns the CRC checksum of the layout. Should only be used after calling `seal()`.

`used()`

Returns the amount of space requested so far.

Should only be used after calling `seal()`.

Focus commands

The plugin provides two - optional - Focus command plugins: `FocusSettingsCommand` and `FocusEEPROMCommand`. These must be explicitly added to `KALEIDOSCOPE_INIT_PLUGINS` if one wishes to use them. They provide the following commands:

`settings.defaultLayer`

Sets or returns (if called without arguments) the ID of the default layer. If set, the keyboard will automatically switch to the given layer when connected. Setting it to `126` or anything higher disables the automatic switching.

This is the Focus counterpart of the `default_layer()` method documented above.

`settings.crc`

Returns the actual, and the expected checksum of the settings.

`settings.valid?`

Returns either `true` or `false`, depending on whether the sealed settings are to be considered valid or not.

`settings.version`

Returns the version of the settings.

`eeprom.contents`

Without argument, displays the full contents of the `EEPROM`, including the settings header.

With arguments, the command updates as much of the `EEPROM` as arguments are provided. It will discard any unnecessary arguments.

`eeprom.free`

Returns the amount of free bytes in `EEPROM`.

`eeprom.erase`

Erases the entire `EEPROM`, and reboots the keyboard to make sure the erase is picked up by every single plugin.

Dependencies

- (Kaleidoscope-FocusSerial)[Kaleidoscope-FocusSerial.md]

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.15 Escape-OneShot

Turn the `Esc` key into a special key, that can cancel any active `OneShot` effect - or act as the normal `Esc` key if none are active, or if any of them are still held. For those times when one accidentally presses a one-shot key, or change their minds.

Additionally, the special `Key_OneShotCancel` key will also count as a oneshot cancel key, would one want a dedicated key for the purpose.

Using the plugin

To use the plugin, one needs to include the header, and activate it. No further configuration is necessary.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-OneShot.h>
#include <Kaleidoscope-Escape-OneShot.h>

KALEIDOSCOPE_INIT_PLUGINS(OneShot,
                          EscapeOneShot);

void setup () {
  Kaleidoscope.setup ();
}
```

If one wishes to configure the plugin at run-time via Focus, the optional `EscapeOneShotConfig` plugin must also be enabled:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-OneShot.h>
#include <Kaleidoscope-Escape-OneShot.h>

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          Focus,
                          OneShot,
                          EscapeOneShot,
                          EscapeOneShotConfig);

void setup () {
  Kaleidoscope.setup ();
}
```

The plugin only makes sense when using one-shot keys.

Plugin methods

The plugin provides the `EscapeOneShot` object, which has the following public configuration methods:

`.setCancelKey(key)`

Changes the `Key` value that will trigger deactivation of one-shot (including sticky) keys. The default is to use `Key_Escape` (the normal `Esc` key), but if you would rather have a dedicated key (so that you can use `Key_Escape` in combination with one-shot modifiers), there is the special `Key_OneShotCancel`, which will not have any side effects.

`.getCancelKey(key)`

Returns the `Key` value that will trigger deactivation of one-shot (including sticky) keys.

Focus commands

The plugin provides a single Focus command: `escape_oneshot.cancel_key`.

`escape_oneshot.cancel_key [keycode]`

Without an argument, returns the raw 16-bit keycode of the cancel key set.

With an argument - a raw 16-bit keycode -, sets the cancel key to the one corresponding to the given code.

Dependencies

- Kaleidoscope-OneShot

Optional dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.16 FingerPainter

The `FingerPainter` plugin provides an elaborate `LED` mode, in which one's able to paint with their fingers: when edit mode is toggled on, keys will - instead of performing their normal function - cycle through the global palette - as provided by the LED-Palette-Theme plugin -, one by one for each tap.

This allows us to edit the theme with the keyboard only, without any special software (except to toggle edit mode on and off).

Using the plugin

To use the plugin, just include the header, add it to the list of used plugins.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Palette-Theme.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-FingerPainter.h>
#include <Kaleidoscope-FocusSerial.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
```

(continues on next page)

(continued from previous page)

```
                          EEPromSettings,
                          LEDPaletteTheme,
                          FingerPainter,
                          Focus);

void setup() {
  Kaleidoscope.setup();
}
```

Plugin methods

The plugin provides the `FingerPainter` object, which provides no public methods.

Focus commands

`fingerpainter.clear`

Clears the canvas, so that one can start a new painting.

`fingerpainter.toggle`

Toggles the painting mode on and off.

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial
- Kaleidoscope-LED-Palette-Theme
- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.17 FirmwareDump

This plugin provides a single Focus command: `firmware.dump`, which dumps the firmware's executable code. One might rightfully wonder what purpose this serves when the source code is available, but rest assured, there is one: in case one wants to temporarily replace their firmware, then put it back on, without having to carry the HEX file around, this command makes that possible: dump the contents, turn them into HEX, and it can be re-flashed at any point. We get a HEX file on-demand, and don't have to carry it around!

The intended primary user of this feature is Chrysalis.

Using the plugin

To use the plugin, include the header, and add it to your list of plugins:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-FirmwareDump.h>

KALEIDOSCOPE_INIT_PLUGINS(FocusSerial, FirmwareDump);

void setup () {
  Kaleidoscope.setup();
}
```

Focus commands

The plugin provides a single Focus command:

`firmware.dump`

Dumps the entire firmware (bootloader not included), even the unused parts.

Dependencies

- [Kaleidoscope-FocusSerial][Kaleidoscope-FocusSerial.md]

10.8.18 FirmwareVersion

Implements a new focus command - version - that simply prints the version set up at compile time.

Using the plugin

To use the plugin, first define the version to be printed, then include the header, and activate the plugin.

```
#define KALEIDOSCOPE_FIRMWARE_VERSION "0.1.2"

#include <Kaleidoscope.h>
#include <Kaleidoscope-FirmwareVersion.h>
#include <Kaleidoscope-FocusSerial.h>

KALEIDOSCOPE_INIT_PLUGINS(Focus,
                          FirmwareVersion);

void setup () {
  Kaleidoscope.setup ();
}
```

Focus commands

The plugin provides a single Focus command: `version`.

`version`

Prints the version configured at build time.

Dependencies

- Kaleidoscope-FocusSerial

10.8.19 FocusSerial

Bidirectional communication for Kaleidoscope. With this plugin enabled, plugins that implement the `onFocusEvent` hook will start responding to Focus commands sent via `Serial`, allowing bidirectional communication between firmware and host.

This plugin is an upgrade of the former Kaleidoscope-Focus plugin. See the UPGRADING.md document for information about how to transition to the new system.

Using the plugin

This plugin is **not** meant to be used by the end-user (apart from setting it up to use plugin-provided hooks), but by plugin authors instead. As an end user, you just need to use Focus-enabled plugins like you normally would, and once `FocusSerial` is enabled, their commands will be available too.

Nevertheless, a very simple example is shown below:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-FocusSerial.h>

namespace kaleidoscope {
class FocusTestCommand : public Plugin {
 public:
  EventHandlerResult onNameQuery() {
    return ::Focus.sendName(F("FocusTestCommand"));
  }

  EventHandlerResult onFocusEvent(const char *input) {
    const char *cmd = PSTR("test");

    if (::Focus.inputMatchesHelp(input))
      return ::Focus.printHelp(cmd);

    if (!::Focus.inputMatchesCommand(input, cmd))
      return EventHandlerResult::OK;

    ::Focus.send(F("Congratulations, the test command works!"));
    return EventHandlerResult::EVENT_CONSUMED;
  }
```

(continues on next page)

(continued from previous page)

```
};
}

kaleidoscope::FocusTestCommand FocusTestCommand;

KALEIDOSCOPE_INIT_PLUGINS(Focus, FocusTestCommand);

void setup () {
  Kaleidoscope.setup ();
}
```

Plugin methods

The plugin provides the `Focus` object, with a couple of helper methods aimed at developers. Terminating the response with a dot on its own line is handled implicitly by `FocusSerial`, one does not need to do that explicitly.

`.inputMatchesHelp(input)`

Returns `true` if the given `input` matches the `help` command. To be used at the top of `onFocusEvent()`, followed by `.printHelp(...)`.

`.printHelp(...)`

Given a series of strings (stored in `PROGMEM`, via `PSTR()`), prints them one per line. Assumes it is run as part of handling the `help` command. Returns `EventHandlerResult::OK`.

`.inputMatchesCommand(input, command)`

Returns `true` if the `input` matches the expected `command`, false otherwise. A convenience function over `strcmp_P()`.

`.send(...)`

`.sendRaw(...)`

Sends a list of variables over the wire. The difference between `.send()` and `.sendRaw()` is that the former puts a space between each variable, the latter does not. If one just wants to send a list of things, use the former. If one needs more control over the formatting, use the latter. In most cases, `.send()` is the recommended method to use.

Both of them take a variable number of arguments, of almost any type: all built-in types can be sent, `cRGB`, `Key` and `bool` too in addition. For colors, `.send()` will write them as an `R G B` sequence; `Key` objects will be sent as the raw 16-bit keycode; and `bool` will be sent as either the string `true`, or `false`.

`.sendName(F("..."))`

To be used with the `onNameQuery()` hook, this sends the plugin name given, followed by a newline, and returns `EventHandlerResult::OK`, so that `onNameQuery()` hooks can be implemented in a single line with the help of this function.

`.read(variable)`

Depending on the type of the variable passed by reference, reads a 8 or 16-bit unsigned integer, a `Key`, or a `cRGB` color from the wire, into the variable passed as the argument.

`.peek()`

Returns the next character on the wire, without reading it. Subsequent reads will include the peeked-at byte too.

`.isEOL()`

Returns whether we're at the end of the request line.

`.COMMENT`

When sending something to the host that is not a response to a request, prefix the response lines with this.

`.SEPARATOR`

To be used when using `.sendRaw`, when one needs complete control over where separators are inserted into the response.

Wire protocol

`Focus` uses a simple, textual, request-response-based wire protocol.

Each request has to be on one line, anything before the first space is the command part (if there is no space, just a newline, then the whole line will be considered a command), everything after are arguments. The plugin itself only parses until the end of the command part, argument parsing is left to the various hooks. If there is anything left on the line after hooks are done processing, it will be ignored.

Responses can be multi-line, but most aren't. Their content is also up to the hooks, `Focus` does not enforce anything, except a trailing dot and a newline. Responses should end with a dot on its own line.

Apart from these, there are no restrictions on what can go over the wire, but to make the experience consistent, find a few guidelines below:

- Commands should be namespaced, so that the plugin name, or functionality comes first, then the sub-command or property. Such as `led.theme`, or `led.setAll`.
- One should not use setters and getters, but a single property command instead. One, which when called without arguments, will act as a getter, and as a setter otherwise.
- Namespaces should be lowercase, while the commands within them camel-case.

- Do as little work in the hooks as possible. While the protocol is human readable, the expectation is that tools will be used to interact with the keyboard.
- As such, keep formatting to the bare minimum. No fancy table-like responses.
- In general, the output of a getter should be copy-pasteable to a setter.
- Messages sent to the host, without a request, should be prefixed with a hash mark (`Focus.COMMENT`).

These are merely guidelines, and there can be - and are - exceptions. Use your discretion when writing Focus hooks.

Example

In the examples below, < denotes what the host sends to the keyboard, > what the keyboard responds.

```
< test
> Congratulations, the test command works!
> .
```

```
< help
> help
> test
> palette
> .
```

```
< palette
> 0 0 0 128 128 128 255 255 255
> .
< palette 0 0 0 128 128 128 255 255 255
> .
```

Further reading

- The `focus-send` script in the Kaleidoscope repo make use of this protocol.
- Starting from the *example* is the recommended way of getting started with the plugin.

10.8.20 GhostInTheFirmware

Born out of the desire to demo LED effects on the keyboard without having to touch it by hand (which would obstruct the video), the `GhostInTheFirmware` plugin allows one to inject events at various delays, by telling it which keys to press. Unlike macros, these press keys at given positions, as if they were pressed by someone typing on it - the firmware will not see the difference.

Given a sequence (with press- and delay times), the plugin will walk through it once activated, and hold the key for the specified amount, release it, and move on to the next after the delay time.

Using the plugin

To use the plugin, one needs to include the header, and configure it with a list of key coordinates, a press time, and a delay time quartett. One also needs a way to trigger starting the sequence, and a macro is the most convenient way for that.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-GhostInTheFirmware.h>
#include <Kaleidoscope-Macros.h>

const macro_t *macroAction(uint8_t macro_id, KeyEvent& event) {
  if (macro_id == 0 && keyToggledOn(event.state))
    GhostInTheFirmware.activate();

  return MACRO_NONE;
}

static const kaleidoscope::plugin::GhostInTheFirmware::GhostKey ghost_keys[] PROGMEM = {
  {KeyAddr(0, 0), 200, 50},
  {KeyAddr::none(), 0, 0}
};

KALEIDOSCOPE_INIT_PLUGINS(GhostInTheFirmware,
                          Macros);

void setup() {
  Kaleidoscope.setup ();

  GhostInTheFirmware.ghost_keys = ghost_keys;
}
```

The plugin won't be doing anything until its `activate()` method is called - hence the macro.

Plugin methods

The plugin provides the `GhostInTheFirmware` object, which has the following methods and properties:

`.activate()`

Start playing back the sequence. Best called from a macro.

`.ghost_keys`

Set this property to the sequence of keys to press, by assigning a sequence to this variable. Each element is a `GhostKey` object, comprised of a `KeyAddr` (the location of a key on the keyboard), a duration of the key press (in milliseconds), and a delay after the key release until the next one is pressed (also in milliseconds).

This `ghost_keys` array *MUST* end with the sentinal value of `{KeyAddr::none(), 0, 0}` to ensure that GhostInTheFirmware doesn't read past the end of the array.

The sequence *MUST* reside in `PROGMEM`.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.21 Heatmap

The Heatmap plugin provides a LED effect, that displays a heatmap on the keyboard. The LEDs under each key will have a color according to how much use they see. Fewer used keys will have deep blue colors, that gradually turns lighter, then green, to yellow, and finally red for the most used keys. The heatmap is not updated on every key press, but periodically. It's precision is also an approximation, and not a hundred percent exact. Nevertheless, it is a reasonable estimate.

Using the plugin

The plugin comes with reasonable defaults pre-configured, all one needs to do is include the header, and make sure the plugin is in use:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-Heatmap.h>

static const cRGB heat_colors[] PROGMEM = {
  {  0,   0,   0}, // black
  {255,  25,  25}, // blue
  { 25, 255,  25}, // green
  { 25,  25, 255}  // red
};

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, HeatmapEffect);

void setup() {
  Kaleidoscope.setup ();

  HeatmapEffect.heat_colors = heat_colors;
  HeatmapEffect.heat_colors_length = 4;
}
```

This sets up the heatmap to update every second (by default). It also registers a new LED effect, which means that if you have not set up any other effects, then Heatmap will likely be the default. You may not want that, so setting up at least one other LED effect, such as LEDOff is highly recommended.

Plugin methods

The plugin provides a HeatmapEffect object, which has the following methods and properties:

`.activate()`

When called, immediately activates the Heatmap effect. Mostly useful in the `setup()` method of the Sketch, or in macros that are meant to switch to the heatmap effect, no matter where we are in the list.

`.update_delay`

The number of milliseconds to wait between updating the heatmap. Updating the heatmap incurs a significant performance penalty, and should not be done too often. Doing it too rarely, on the other hand, make it much less useful. One has to strike a reasonable balance.

Defaults to *1000*.

`.heat_colors`

A cRGB array describing the gradian of colors that will be used, from colder to hoter keys. E.g. with `heat_colors = {{100, 0, 0}, {0, 100, 0}, {0, 0, 100}}`, a key with a temperature of 0.8 (0=coldest, 1=hotest), will end up with a color `{0, 40, 60}`.

Defaults to `{{0, 0, 0}, {25, 255, 25}, {25, 255, 255}, {25, 25, 255}}` (black, green, yellow, red)

`.heat_colors_length`

Length of the `heat_colors` array.

Defaults to *4*

Dependencies

- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.22 HostOS

The `HostOS` extension is not all that useful in itself, rather, it is a building block other plugins and extensions can use to not repeat the same guesswork and logic.

The goal is to have a single place that remembers the host OS, whether set by the end-user in a Sketch, or via a macro, or some other way. This information can then be reused by other plugins.

See the Unicode extension for an example about how to use `HostOS` in practice.

Using the extension

The extension provides a `HostOS` singleton object.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-HostOS.h>

void someFunction() {
  if (HostOS.os() == kaleidoscope::hostos::LINUX) {
    // do something linux-y
  }
  if (HostOS.os() == kaleidoscope::hostos::MACOS) {
    // do something macOS-y
  }
}

KALEIDOSCOPE_INIT_PLUGINS(HostOS)

void setup() {
  Kaleidoscope.setup ();
}
```

Extension methods

The extension provides the following methods on the `HostOS` singleton:

`.os()`

Returns the stored type of the Host OS.

`.os(type)`

Sets the type of the host OS, overriding any previous value. The type is then stored in EEPROM for persistence.

Host OS Values

The OS type (i.e. the return type of `.os()` and the arguments to `.os(type)`) will be one of the following:

- `kaleidoscope::hostos::LINUX`
- `kaleidoscope::hostos::MACOS`
- `kaleidoscope::hostos::WINDOWS`
- `kaleidoscope::hostos::OTHER`

For compability reasons, `kaleidoscope::hostos::OSX` is an alias to `kaleidoscope::hostos::MACOS`.

Focus commands

The plugin provides the `FocusHostOSCommand` object, which, when enabled, provides the `hostos.type` Focus command.

`hostos.type [type]`

Without argument, returns the current OS type set (a numeric value).

With an argument, it sets the OS type.

This command can be used from the host to reliably set the OS type within the firmware.

Dependencies

- Kaleidoscope-EEPROM-Settings

Further reading

Starting from the *example* is the recommended way of getting started with the extension.

10.8.23 HostPowerManagement

Support performing custom actions whenever the host suspends, resumes, or is sleeping.

Using the plugin

To use the plugin, one needs to include the header, and activate it. No further configuration is necessary, unless one wants to perform custom actions.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-HostPowerManagement.h>

KALEIDOSCOPE_INIT_PLUGINS(HostPowerManagement);

void setup () {
  Kaleidoscope.setup ();
}
```

Plugin methods

The plugin provides the `HostPowerManagement` object, with no public methods.

Overridable methods

`hostPowerManagementEventHandler(event)`

The `hostPowerManagementEventHandler` method is the brain of the plugin: this function tells it what action to perform in response to the various events.

Currently supported events are: `kaleidoscope::plugin::HostPowerManagement::Suspend` is fired once when the host suspends; `kaleidoscope::plugin::HostPowerManagement::Sleep` is fired every cycle while the host is suspended; `kaleidoscope::plugin::HostPowerManagement::Resume` is fired once when the host wakes up.

The default implementation is empty.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

Caveats

On some systems, there can be a long delay between suspending/sleeping the host and the firmware responding to it and calling `hostPowerManagementEventHandler()`. In particular, on macOS, it can take 30 seconds or more after invoking "sleep" mode on the host before the keyboard responds. One user reports that it can take more than a minute, so if this plugin doesn't appear to be working, please wait a few minutes and check again.

10.8.24 IdleLEDs

Having LED effects on the keyboard can be exceptionally helpful. However, having the effects - or lights, in general - on all the time, even when the keyboard is otherwise idle, is perhaps not the best. When one leaves the keyboard, locks the computer, what use are the LED effects then?

One could turn them off manually, but… that's too easy to forget, and why do something the firmware could do for us anyway? What if the LEDs turned themselves off after some configurable idle time? Say, if one did not press any keys for the past ten minutes, just shut 'em off.

This is exactly what the `IdleLEDs` plugin does.

Using the plugin

The plugin comes with reasonable defaults (see below), and can be used out of the box, without any further configuration:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>
#include <Kaleidoscope-IdleLEDs.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, IdleLEDs, LEDEffectRainbowWave);

void setup () {
  Kaleidoscope.setup ();
}
```

Because the plugin needs to know about key events, it is best to make it one of the first plugins, so it can catch all of them, before any other plugin would have a chance to consume key events.

It is also possible to enable run-time configuration via he `Focus` plugin, and persistent storage of such settings. To do that, one has to use the `PersistentIdleLEDs` object instead, provided by the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-IdleLEDs.h>

KALEIDOSCOPE_INIT_PLUGINS(
  EEPROMSettings,
  Focus,
  LEDControl,
  PersistentIdleLEDs,
  LEDEffectRainbowWave
);

void setup () {
  Kaleidoscope.setup ();
}
```

Plugin Properties

The plugin provides two objects, `IdleLEDs`, and `PersistentIdleLEDs`, both with the following properties and methods.

`.idle_time_limit`

Property storing the amount of time that can pass without a single key being pressed before the plugin considers the keyboard idle and turns off the LEDs. Value is expressed in milliseconds.

Defaults to 600000 milliseconds (10 minutes).

Provided for compatibility reasons. It is recommended to use one of the methods below instead of setting this property directly. If using `PersistentIdleLEDs`, setting this property will not persist the value to storage. Use `.setIdleTimeoutSeconds()` if persistence is desired.

`.idleTimeoutSeconds()`

Returns the amount of time (in seconds) that can pass without a single key being pressed before the plugin considers the keyboard idle and turns off the LEDs.

`.setIdleTimeoutSeconds(uint32_t new_limit)`

Sets the amount of time (in seconds) that can pass without a single key being pressed before the plugin considers the keyboard idle and turns off the LEDs.

Setting the timeout to 0 will disable the plugin until it is set to a higher value.

Focus commands

The plugin provides a single Focus command, but only when using the `PersistentIdleLEDs` variant:

`idleleds.time_limit [seconds]`

Sets the idle time limit to `seconds`, when called with an argument. Returns the current limit (in seconds) when called without any.

Setting the timeout to 0 will disable the plugin until it is set to a higher value.

Dependencies

- Kaleidoscope-LEDControl

Optional dependencies

- Kaleidoscope-EEPROM-Settings
- FocusSerial

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.25 LED-ActiveLayerColor

A simple way to light up the keyboard in uniform colors, depending on what layer one's on. Unlike Colormap, all keys will be the same color. But this plugin uses considerably less resources, and is easier to set up as well. A perfect solution when one wants to quickly see what layer they're on, with minimal resources and time investment.

Using the plugin

To use the plugin, one needs to include the header, and activate the effect. Then, one needs to configure a color map:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-ActiveLayerColor.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDActiveLayerColorEffect);

```

(continues on next page)

(continued from previous page)

```
void setup () {
  static const cRGB layerColormap[] PROGMEM = {
    CRGB(128, 0, 0),
    CRGB(0, 128, 0)
  };

  Kaleidoscope.setup();
  LEDActiveLayerColorEffect.setColormap(layerColormap);
}
```

Plugin properties

The plugin provides the `LEDActiveLayerColorEffect` object, which has the following method:

`.setColormap(colormap)`

Sets the colormap to the supplied map. Each element of the map should be a `cRGB` color, and the array must have the same amount of items as there are layers. The map should reside in PROGMEM.

Dependencies

- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.26 LED-ActiveLayerKeys

A simple way to light up all keys in the top layer in uniform colors. Unlike Colormap, all keys will be the same color. But this plugin uses considerably less resources, and is easier to set up as well. A perfect solution when one wants to quickly see what layer they're on and which keys are on that layer, with minimal resources and time investment.

Using the plugin

To use the plugin, one needs to include the header, and activate the effect. Then, one needs to configure a color map:

```
#include "Kaleidoscope.h"
#include "Kaleidoscope-LEDControl.h"
#include "Kaleidoscope-LED-ActiveLayerKeys.h"

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDActiveLayerKeysEffect);

void setup () {
  static const cRGB layerColormap[] PROGMEM = {
    CRGB(0xff, 0x00, 0x00),  // red, for the first layer
```

(continues on next page)

(continued from previous page)

```
    CRGB(0x00, 0xff, 0x00),  // green, for the second layer
    CRGB(0x00, 0x00, 0xff),  // blue, for the third layer
  };

  Kaleidoscope.setup();
  // By default, only LEDs for keys on the topmost layer are lit.
  //LEDActiveLayerKeysEffect.lightLowerLayers(false);
  LEDActiveLayerKeysEffect.setColormap(layerColormap);
}
```

Plugin properties

The plugin provides the `LEDActiveLayerKeysEffect` object, which has the following methods:

`.setColormap(colormap)`

Sets the colormap to the supplied map. Each element of the map should be a `cRGB` color, and the map should reside in PROGMEM. The array should have the same amount of items as there are layers. Any layer that doesn't have a matching entry in the array, will have leds turned off.

`.lightLowerLayers(boolean)`

By default, this plugin only lights up LEDs keys on the topmost layer. This method allows overriding this default, to have the plugin change the leds of all non-blocked keys to the color of their respective layers.

Dependencies

- Kaleidoscope-LEDControl

10.8.27 LED-ActiveModColor

With this plugin, any active modifier on the keyboard will have the LED under it highlighted. No matter how the modifier got activated (a key press, a macro, anything else), the coloring will apply. Layer shift keys and OneShot layer keys count as modifiers as far as the plugin is concerned.

Using the plugin

To use the plugin, one needs to include the header, and activate the effect. It is also possible to use a custom color instead of the white default.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-ActiveModColor.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          ActiveModColorEffect);
```

(continues on next page)

(continued from previous page)

```
void setup () {
  Kaleidoscope.setup ();

  ActiveModColorEffect.highlight_color = CRGB(0x00, 0xff, 0xff);
}
```

It is recommended to place the activation (the `KALEIDOSCOPE_INIT_PLUGINS` parameter) of the plugin last, so that it can reliably override any other plugins that may work with the LEDs, and apply the highlight over those.

Plugin properties

The plugin provides the `ActiveModColorEffect` object, which has the following configuration methods. These methods all take a `cRGB` object, which can be written as `CRGB(r, g, b)`, where `r`, `g`, and `b` are all 8-bit integers (0-255). For example, `CRGB(50, 0, 50)` would be a purple-ish color.

`.setHighlightColor(color)`

Sets the color (a `cRGB` object) to use for highlighting normal modifier keys and layer-shift keys. Defaults to a white color.

`.setOneShotColor(color)`

Sets the color (a `cRGB` object) to use for highlighting active one-shot keys. These are the keys that will time out or deactivate when a subsequent key is pressed. Defaults to a yellow color.

`.setStickyColor(color)`

Sets the color (a `cRGB` object) to use for highlighting "sticky" one-shot keys. These keys will remain active until they are pressed again. Defaults to a red color.

Plugin methods

The `ActiveModColorEffect` object provides the following methods:

`.highlightNormalModifiers(bool)`

Can be used to enable or disable the highlighting of normal modifiers. Defaults to true.

Dependencies

- Kaleidoscope-LEDControl
- Kaleidoscope-OneShot
- Kaleidoscope-OneShotMetaKeys

The `ActiveModColorEffect` plugin doesn't require that either OneShot or OneShotMetaKeys plugins are registered with `KALEIDOSCOPE_INIT_PLUGINS()` in order to work, but it does depend on their header files.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.28 LED-AlphaSquare

An alphabet for your per-key LEDs, `AlphaSquare` provides a way to display 4x4 "pixel" symbols on your keyboard. With this building block, one can build some sweet animations, or just show off - the possibilities are almost endless!

Using the plugin

To use the plugin, one needs to include the header in their Sketch, tell the firmware to `use` the plugin, and one way or another, call the `display` method. This can be done from a macro, or via the `AlphaSquareEffect` LED mode.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-AlphaSquare.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          AlphaSquare,
                          AlphaSquareEffect);

void setup() {
  Kaleidoscope.setup();

  AlphaSquare.display (Key_A);
}
```

Plugin methods

The plugin provides the `AlphaSquare` object, which has its methods and properties listed below, and an `AlphaSquareEffect` LED mode, which has no methods or properties other than those provided by all LED modes.

`.display(key)`

`.display(key, col)`

`.display(key, key_addr)`

`.display(key, key_addr, color)`

Display the symbol for `key` at the given led address, with pixels set to the specified `color`. If only `col` is provided, the first row - `0` is assumed. If the column is omitted, then the third column - `2` - is used. If the `color` is omitted, the plugin will use the global `.color` property.

The plugin can display the English alphabet, and the numbers from 0 to 9. The symbol will be drawn with the top-left corner at the given position.

Please consult the appropriate hardware library of your keyboard to see how keys are laid out in rows and columns.

`.display(symbol)`

`.display(symbol, col)`

`.display(symbol, key_addr)`

`.display(symbol, key_addr, color)`

As the previous function, but instead of a key, it expects a 4x4 bitmap in the form of a 16-bit unsigned integer, where the low bit is the top-right corner, the second-lowest bit is to the right of that, and so on.

The `SYM4x4` macro can be used to simplify creating these bitmaps.

`.clear(key)`, `.clear(symbol)`

`.clear(key, col)`, `.clear(symbol, col)`

`.clear(key, key_addr)`, `.clear(symbol, key_addr)`

Just like the `.display()` counterparts, except these clear the symbol, by turning the LED pixels it is made up from off.

`.color`

The color to use to draw the pixels.

Defaults to `{ 0x80, 0x80, 0x80 }` (light gray).

Plugin helpers

`SYM4x4(...)`

A helper macro, which can be used to set up custom bitmaps. It expects 16 values, a 4x4 square of zeroes and ones. Zeroes are transparent pixels, ones will be colored.

Extra symbols

There is a growing number of pre-defined symbols available in the `kaleidoscope::plugin::alpha_square::symbols` namespace. Ok, growing may have been an exaggeration, there is only one as of this writing:

`Lambda`

A lambda () symbol.

Dependencies

- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.29 LED-Palette-Theme

A common base for plugins that want to provide themes, or theme-related capabilities, using a 16 color palette. In other words, this is for plugin authors primarily. The primary aim of the plugin is to provide not only a common palette, but tools that make it easier to use it too.

Using the plugin

To use the plugin, one needs to do a bit more than include the header, and tell the firmware to use it. Itself being a mere building block, to use it to its full extent, we need to create our own plugin on top of it.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-LED-Palette-Theme.h>
#include <Kaleidoscope-FocusSerial.h>

namespace example {

```

(continues on next page)

(continued from previous page)

```
class TestLEDMode : public LEDMode {
 protected:
  void setup() final;
  void update() final;

  kaleidoscope::EventHandlerResult onFocusEvent(const char *input);

 private:
  static uint16_t map_base_;
};

uint16_t TestLEDMode::map_base_;

void TestLEDMode::setup() {
  map_base_ = LEDPaletteTheme.reserveThemes(1);
}

void TestLEDMode::update() {
  LEDPaletteTheme.updateHandler(map_base_, 0);
}

kaleidoscope::EventHandlerResult
TestLEDMode::onFocusEvent(const char *input) {
  return LEDPaletteTheme.themeFocusEvent(input, PSTR("testLedMode.map"), map_base_, 1);
}

}

example::TestLEDMode TestLEDMode;

KALEIDOSCOPE_INIT_PLUGINS(
  Focus,
  LEDPaletteTheme,
  TestLEDMode,
  EEPROMSettings
);

void setup() {
  Kaleidoscope.setup();

  TestLEDMode.activate();
}
```

This is a simple extension, where it provides a `testLEDMode.map` Focus command, with which one can set the theme which will be saved to EEPROM.

Plugin methods

The plugin provides the `LEDPaletteTheme` object, which has the following methods and properties:

`.reserveThemes(max_themes)`

Reserve space in EEPROM for `max_themes`. Each key on a theme uses half a byte of space. The function returns the `theme_base` to be used with the rest of the methods.

The `theme_base` is a pointer into the EEPROM where the theme storage starts.

`.updateHandler(theme_base, theme)`

A helper we can call in our plugin's `.update()` method: given an EEPROM location (`theme_base`), and a `theme` index, it will update the keyboard with the colors of the specified theme.

The `theme` argument can be any index between zero and `max_themes`. How the plugin decides which theme to display depends entirely on the plugin.

`.themeFocusEvent(command, expected_command, theme_base, max_themes)`

To be used in a custom `Focus` handler: handles the `expected_command` Focus command, and provides a way to query and update the themes supported by the plugin.

When queried, it will list the color indexes. When used as a setter, it expects one index per key.

The palette can be set via the `palette` focus command, provided by the `LEDPaletteTheme` plugin.

Focus commands

`palette`

Without arguments, prints the palette: RGB values for all 16 colors.

With arguments, updates the palette with new colors. One does not need to give the full palette, the plugin will process as many arguments as available, and ignore anything past the last index. It expects colors to have all three components specified, or none at all. Thus, partial palette updates are possible, but only on the color level, not at component level.

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial
- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.30 LED-Stalker

The `StalkerEffect` plugin provides an interesting new typing experience: the LEDs light up as you tap keys and play one of the selected effects: a haunting trail of ghostly white lights, or a blazing trail of fire.

Using the plugin

To use the plugin, one needs to include the header and select the effect.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Stalker.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, StalkerEffect);

void setup (){
  Kaleidoscope.setup();

  StalkerEffect.variant = STALKER(Haunt, (CRGB(0, 128, 0)));
  StalkerEffect.activate();
}
```

It is recommended to place the activation of the plugin (the `Kaleidoscope.use` call) as early as possible, so the plugin can catch all relevant key presses. The configuration can happen at any time and should use the `STALKER` macro to do so.

Plugin methods

The plugin provides the `StalkerEffect` object, which has the following properties:

`.variant`

Set the effect to use with the plugin. See below for a list.

It is recommended to use the `STALKER` macro to declare the effect itself.

`.step_length`

The length - in milliseconds - of each step of the animation. An animation lasts 256 steps.

Defaults to 50.

`.inactive-color`

The color to use when a key hasn't been pressed recently.

Defaults to `(cRGB) { 0, 0, 0 }`

Plugin helpers

`STALKER(effect, params)`

Returns an effect, to be used to assign a value the `.variant` property of the `StalkerEffect` object. Any arguments given to the macro are passed on to the effect. If the effect takes no arguments, use an empty `params` list.

Plugin effects

The plugin provides the following effects:

`Haunt([color])`

A ghostly haunt effect, that trails the key taps with a ghostly white color (or any other color, if specified). Use the `CRGB(r,g,b)` macro to specify the color, if you want something else than the ghostly white.

`BlazingTrail()`

A blazing trail of fire will follow our fingers!

`Rainbow()`

Leave a rainbow behind, where your fingers has been!

Dependencies

- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.31 LED-Wavepool

The `WavepoolEffect` plugin makes waves of light splash out from each keypress. When idle, it will also simulate gentle rainfall on the keyboard.

Using the plugin

To use the plugin, one needs to include the header and select the effect.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Wavepool.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, WavepoolEffect);

void setup (){
  Kaleidoscope.setup();

  WavepoolEffect.idle_timeout = 5000;  // 5 seconds
  WavepoolEffect.activate();
}
```

It is recommended to place the activation of the plugin as early as possible, so the plugin can catch all relevant key presses.

Plugin properties

The plugin provides the `WavepoolEffect` object, which has the following properties:

`.idle_timeout`

When to keys are being pressed, light will periodically splash across the keyboard. This value sets the delay in ms before that starts.

To disable the idle animation entirely, set this to 0.

Default is 5000 (5 seconds).

`.ripple_hue`

The Hue of the ripple animation. If set, the light splashing across the keyboard will use this value instead of all colors of the rainbow.

Setting it to the special value of `WavepoolEffect.rainbow_hue` will cause the plugin to use all colors again.

Defaults to `WavepoolEffect.rainbow_hue`.

Dependencies

- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.32 LEDBrightnessConfig

The `LEDBrightnessConfig` plugin provides a way to set the brightness of all LEDs on a keyboard, and persist this setting to EEPROM too.

Using the plugin

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDBrightnessConfig.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>
#include <Kaleidoscope-FocusSerial.h>

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          LEDControl,
                          LEDOff,
                          LEDRainbowEffect,
                          LEDRainbowWaveEffect,
                          Focus,
                          LEDBrightnessConfig);

void setup() {
  Kaleidoscope.setup();
}
```

Focus commands

`led.brightness`

Without arguments, prints the current brightness.

If an argument is given, it sets the brightness to the desired value, and stores it in EEPROM.

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

10.8.33 Kaleidoscope-LEDControl

This is a plugin for Kaleidoscope, for controlling the LEDs, and LED effects. It is also a building block for plugins that control LEDs.

Using the extension

Plugin methods

`.next_mode(void)`

Activates the next LED mode. Cycles to the first LED mode if the current LED mode is the last one.

`.prev_mode(void)`

Activates the previous LED mode. Cycles to the last LED mode if the current LED mode is the first one.

`.get_mode()`

Returns the current LED mode.

`.get_mode<typename>()`

`.set_mode(uint8_t mode_id)`

Activates a LED mode by its index in the firmware. If the index exceeds the numer of led modes, the method returns early.

`.get_mode_index()`

Returns the index of the currently active LED mode.

`.refreshAll()`

If the hardware has LEDs and LEDs are enabled, turn all LEDs off and then trigger the current LED mode to refresh.

`.setCrgbAt(uint8_t led_index, cRGB crgb)`

Sets the specified LED to the provided color.

`.setCrgbAt(KeyAddr key_addr, cRGB color)`

Sets the LED for the specified key to the provided color.

`.getCrgbAt(uint8_t led_index)`

Get the LED color of the specified LED.

`.getCrgbAt(KeyAddr key_addr)`

Get the LED color of the LED for the specified key.

`.syncLeds(void)`

Force an update of all LEDs.

`.set_all_leds_to(uint8_t r, uint8_t g, uint8_t b)`

Set all LEDs using the provided rgb values.

`.set_all_leds_to(cRGB color)`

Set all LEDs to the specified color.

`.setSyncInterval(uint8_t interval)`

Set the interval at which the LEDs should sync, in milliseconds.

Note: LED updates are considered on each cycle of the runtime. Because of that, the interval effectively means that *at least* `interval` milliseconds has passed before LEDs are synced.

`.setBrightness(uint8_t brightness)`

Set the brightness for all LEDs.

`.getBrightness()`

Returns the current brightness of the LEDs as a uint8_t.

`.onSetup()`

See [[event-handler-hooks]]

`.setup(void)`

`.onKeyEvent(KeyEvent &event)`

See [[event-handler-hooks]]

`.afterEachCycle()`

See [[event-handler-hooks]]

`.update(void)`

Triggers the currently active LED mode to update. It is up to the LED mode to handle this correctly.

`.refreshAt(KeyAddr key_addr)`

Triggers the currently active LED mode to refresh the LED at the specified key address.

`.activate(LEDModeInterface *plugin)`

`.disable()`

Turn off all LEDs and disables updating LEDs

`.enable()`

Enables updating LEDs and calls `refreshAll()`

`.isEnabled()`

Returns a bool value reflecting whether LEDs are currently enabled.

10.8.34 LEDEffect-BootAnimation

With this plugin enabled, the keyboard will play a little boot animation when starting up (this animation does not inhibit typing, you can still use the keyboard while the animation plays).

Using the plugin

To use the plugin, include the header, and tell `Kaleidoscope` to use the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-BootAnimation.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          BootAnimationEffect
                          LEDOff);

void setup() {
  Kaleidoscope.setup();
}
```

Plugin properties

The plugin provides the `BootAnimationEffect` object, with the following properties:

`.timeout`

This property specifies the timeout (in milliseconds) each step of the animation is displayed.

Defaults to `1000` ms, or one second.

`.color`

This property sets the color the animation is played with.

The default is a red color.

Dependencies

- Kaleidoscope-LEDControl

10.8.35 LEDEffect-BootGreeting

If you want to have your keyboard signal when it turns on, but you don't want to use any more complicated LED modes, this plugin is for you. It will make the `LEDEffectNext` key on your keymap slowly breathe for about ten seconds after plugging the keyboard in (without blocking the normal functionality of the keyboard, of course).

Using the plugin

To use the plugin, include the header, and tell `Kaleidoscope` to use the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-BootGreeting.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          BootGreetingEffect
                          LEDOff);

void setup() {
  Kaleidoscope.setup();
}
```

You may also set optional parameters.

Specify by search key

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-BootGreeting.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          BootGreetingEffect
                          LEDOff);

void setup() {
  Kaleidoscope.setup();

  BootGreetingEffect.search_key = Key_M;
}
```

Specify by position

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-BootGreeting.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          BootGreetingEffect
                          LEDOff);

void setup() {
  Kaleidoscope.setup();

  //Butterfly key
  BootGreetingEffect.key_col = 7;
  BootGreetingEffect.key_row = 3;
}
```

Specify longer timeout

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-BootGreeting.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          BootGreetingEffect
                          LEDOff);

void setup() {
  Kaleidoscope.setup();

  //Butterfly key
  BootGreetingEffect.timeout = 15000;
}
```

Specify different color

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-BootGreeting.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          BootGreetingEffect
                          LEDOff);

void setup() {
  Kaleidoscope.setup();

  //Butterfly key
```

(continues on next page)

(continued from previous page)

```
  BootGreetingEffect.hue = 90;

  Kaleidoscope.setup();
}
```

Plugin methods

The plugin provides the `BootGreetingEffect` object, with the following methods and properties:

`.search_key`

Set the key in the current keymap that should be activated with the pulsing LED on startup. The plugin will search from the top left to the bottom right of the keyboard, row by row, to find this key. The first matching key will be selected.

Defaults to `Key_LEDEffectNext`

`.key_row`

This is an optional override to explicitly set the selected key by exact row and column. This number is 0-indexed, so the top row is 0, the second row is 1, etc. Must set `.key_col` property for this feature to be enabled.

`.key_col`

This is an optional override to explicitly set the selected key by exact row and column. This number is 0-indexed, so the left-most column is 0, the second column is 1, etc. Must set `.key_row` property for this feature to be enabled.

`.timeout`

This property specifies the timeout (in milliseconds) for the effect to last. When the keyboard is first connected, the pulsing LED effect will last for this duration before turning off.

Defaults to `9200` ms.

`.hue`

This property sets the color hue that the LED pulsing effect.

The default is `170`, which is a blue color.

Dependencies

- Kaleidoscope-LEDControl

10.8.36 LEDEffect-Breathe

Provides a breathing effect for the keyboard. Breathe in, breathe out.

Using the extension

To use the plugin, include the header, and tell the firmware to use it:

```
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Breathe.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDBreatheEffect);

void setup() {
  Kaleidoscope.setup();
}
```

Plugin properties

The plugin provides the `LEDBreatheEffect` object, which has a single property:

`.hue`

The hue of the breathe effect.

Defaults to 170, a blue hue.

`.saturation`

The color saturation of the breathe effect.

Defaults to 255, the maximum.

Dependencies

- Kaleidoscope-LEDControl

10.8.37 LEDEffect-Chase

A simple LED effect where one color chases another across the keyboard and back, over and over again. Playful colors they are.

Using the extension

To use the plugin, include the header, and tell the firmware to use it:

```
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Chase.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDEffect-Chase);

void setup() {
  Kaleidoscope.setup();
}
```

Plugin methods

The plugin provides the `LEDChaseEffect` object, which has the following methods outside of those provided by all LED modes:

`.update_delay([delay])`

Accessor for the update delay, the time between each step of the animation. When called without an argument, returns the current setting. When called with one, sets it.

Defaults to 150 (milliseconds).

`.distance([pixels])`

Accessor for the distance between the two pixels. When called without an argument, returns the current setting. When called with one, sets it.

Defaults to 5.

Dependencies

- Kaleidoscope-LEDControl

10.8.38 Kaleidoscope-LEDEffect-DigitalRain

An LED effect similar to the “digital rain” seen in the “Matrix” films with green lights flowing downwards on the keyboard.

Using the extension

To use the plugin, include the header, and tell the firmware to use it:

```
#include <Kaleidoscope-LEDEffect-DigitalRain.h>

KALEIDOSCOPE_INIT_PLUGINS(
  LEDDigitalRainEffect
);

void setup() {
  Kaleidoscope.setup();

  // Optionally adjust the configuration
  LEDDigitalRainEffect.setDropMs(260); // Make the rain fall more slowly
  LEDDigitalRainEffect.setColorChannel(LEDDigitalRainEffect.ColorChannel::BLUE);

  // Optionally switch this LED mode on at startup
  LEDDigitalRainEffect.activate();
}
```

Plugin methods

The plugin provides the `LEDDigitalRainEffect` object, which has various public getters and setters for configuration, as well as the methods associated with all LED mode plugins.

`.getDecayMs()`

`.setDecayMs(decayMs)`

Gets or sets the number of milliseconds it takes for a raindrop to decay from full intensity.

Defaults to 2000 milliseconds.

`.getDropMs()`

`.setDropMs(dropMs)`

Gets or sets the number of milliseconds before digital raindrops fall one row.

Defaults to 180 milliseconds.

`.getNewDropProbability()`

`.setNewDropProbability(probability)`

Get or set the *inverse* probability of a new raindrop appearing at the top of each column each time drops fall. Must be a value between 0 and 255.

Defaults to 18.

`.getTintShadeRatio()`

`.setTintShadeRatio(ratio)`

Get or set the intensity level (between 0 and 255) at which pure green (or red, or blue) should be output. This allows the timeshare ratio of tints vs shades of green to be controlled. A tint-shade ratio of 0 means all tints, while 255 means all shades.

Defaults to 208.

`.getMaximumTint()`

`.setMaximumTint(max)`

Gets or sets the maximum tint of a pixel. A value of 0 means nothing brighter than pure green (or red, or blue), while a value of 255 would mean tinting all the way to pure white.

Defaults to 192.

`.getColorChannel()`

`.setColorChannel(channel)`

Gets or sets the color channel to use. Values can be either of the following:

- `LEDDigitalRainEffect.ColorChannel::RED`
- `LEDDigitalRainEffect.ColorChannel::GREEN`
- `LEDDigitalRainEffect.ColorChannel::BLUE`

Defaults to green.

Dependencies

- Kaleidoscope-LEDControl

10.8.39 LEDEffect-Rainbow

Two colorful rainbow effects are implemented by this plugin: one where the rainbow waves through the keys, and another where the LEDs breathe though the colors of a rainbow. The difference is that in the first case, we have all the rainbow colors on display, and it waves through the keyboard. In the second case, we have only one color at a time, for the whole board, and the color cycles through the rainbow's palette.

Using the extension

To use the plugin, include the header, and tell the firmware to use either (or both!) of the effects:

```
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDRainbowEffect, LEDRainbowWaveEffect);

void setup() {
  Kaleidoscope.setup();

  LEDRainbowEffect.brightness(150);
  LEDRainbowWaveEffect.brightness(150);
  LEDRainbowWaveEffect.update_delay(50);
}
```

Plugin methods

The plugin provides two objects: `LEDRainbowEffect`, and `LEDRainbowWaveEffect`, both of which provide the following methods:

`.brightness([brightness])`

Sets (or gets, if called without an argument) the LED brightness for the effect.

Defaults to 50.

`.update_delay([delay])`

Sets (or gets, if called without an argument) the number of milliseconds between effect updates. Smaller number results in faster rainbows.

Defaults to 40.

Dependencies

- Kaleidoscope-LEDControl

10.8.40 LEDEffect-SolidColor

This plugin provides tools to build LED effects that set the entire keyboard to a single color. For show, and for back-lighting purposes.

Using the extension

To use the plugin, include the header, declare an effect using the `kaleidoscope::plugin::LEDSolidColor` class, and tell the firmware to use the new effect:

```
#include <Kaleidoscope-LEDEffect-SolidColor.h>

static kaleidoscope::plugin::LEDSolidColor solidRed(160, 0, 0);

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, solidRed);

void setup() {
  Kaleidoscope.setup();
}
```

Dependencies

- Kaleidoscope-LEDControl

10.8.41 LEDEffects

The `LEDEffects` plugin provides a selection of LED effects, each of them fairly simple, simple enough to not need a plugin of their own.

Using the plugin

There are a number of different effects included in the package, all of them are available once including the header, and one's free to choose any number of them.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDEffects.h>

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, JukeBoxEffect);

void setup() {
  Kaleidoscope.setup();
}
```

Included effects

All of these effects will scan the active layers, and apply effects based on what keys are active on each position, thus, it needs no hints or configuration to figure out our layout!

`MiamiEffect`

Applies a color effect to the keyboard, inspired by the popular Miami keyset:

plugins/extras/MiamiEffect.png

Alphas, punctuation, numbers, the space bar, the numbers and the dot on the keypad, and half the function keys will be in a cyan-ish color, the rest in magenta.

`JukeboxEffect`

Applies a color effect to the keyboard, inspired by the JukeBox keyset:

plugins/extras/JukeboxEffect.png

Alphas, punctuation, numbers, the space bar, the numbers and the dot on the keypad, and half the function keys will be in a beige-ish color, the rest in light green, except for the `Esc` key, which will be in red.

An alternative color scheme exists under the `JukeboxAlternateEffect` name, where the light green and red colors are swapped.

TriColor

TriColor is a class that can be used to create LED effects that all follow a similar pattern: alphas and similar in one color; modifiers, special keys, and half the function keys in another, and `Esc` in a third (this latter being optional). If we have a color scheme that follows this pattern, the `TriColor` extension can make it a lot easier to implement it.

Using the extension

Because the extension is part of the `LEDEffects` library, we need to include that header:

```
#include <Kaleidoscope-LEDEffects.h>
```

Then, we simply create a new instance of the `TriColor` class, with appropriate colors set for the constructor:

```
kaleidoscope::plugin::TriColor BlackAndWhiteEffect (CRGB(0x00, 0x00, 0x00),
                                                     CRGB(0xff, 0xff, 0xff),
                                                     CRGB(0x80, 0x80, 0x80));
```

The first argument is the base color, the second is for modifiers and special keys, the last one is for the `Esc` key. If the last one is omitted, the extension will use the modifier color for it.

Plugin methods

The plugin provides a single method on each of the included effect objects:

`.activate()`

When called, immediately activates the effect. Mostly useful in the `setup()` method of the Sketch, or in macros that are meant to switch to the selected effect, no matter where we are in the list.

Dependencies

- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.42 LayerFocus

The `LayerFocus` plugin exposes a number of layer-related commands via Focus, to allow controlling layers from the host side.

Using the plugin

To use the plugin, we need to include the header, and let the firmware know we want to use it:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-LayerFocus.h>

KALEIDOSCOPE_INIT_PLUGINS(
  Focus,
  LayerFocus
);
```

Focus commands

The plugin provides the following Focus commands:

`layer.activate N` / `layer.deactivate N` / `layer.isActive N`

Activates, deactivates, or queries the state of layer `N`.

`layer.moveTo N`

Moves to layer `N`, deactivating all other layers in the process.

`layer.state [STATE...]`

Without arguments, display the state of all layers, from lower to higher. Each active layer will be represented by `1`, while inactive layers will be represented by `0`.

With arguments, override the state of layers with the `STATE` given.

Dependencies

- Kaleidoscope-FocusSerial

10.8.43 LayerNames

This plugin provides a [Focus][plugin:focus]-based interface for storing custom layer names, to be used by software such as Chrysalis. The firmware itself does nothing with the layer names, it is purely for use by applications on the host side.

Using the plugin

To use the plugin, we need to include the header, initialize the plugin with `KALEIDOSCOPE_INIT_PLUGINS()`, and reserve storage space for the names. This is best illustrated with an example:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROMSettings.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-LayerNames.h>

KALEIDOSCOPE_INIT_PLUGINS(
  EEPROMSettings,
  Focus,
  LayerNames
);

void setup() {
  Kaleidoscope.setup();
```

(continues on next page)

(continued from previous page)

```
  LayerNames.reserve_storage(128);
}
```

Plugin methods

The plugin provides a `LayerNames` object, with the following method available:

`.reserve_storage(size)`

Reserves `size` bytes of storage for layer names. This must be called from the `setup()` method of your sketch.

Focus commands

The plugin provides a single Focus command: `keymap.layerNames`.

`keymap.layerNames [name_length name]...`

Without arguments, displays all the stored layer names. Each layer is printed on its own line, preceded by its length. At the end, the plugin will also print an extra line with a name length of zero, followed by the string "size=", and then the total size of the storage reserved for layer names.

To set custom names, a list of length & name pairs must be given. The plugin stops processing arguments when it encounters a name length of 0.

Example

```
> keymap.layerNames
< 6 Qwerty
< 6 Numpad
< 8 Function
< 0 size=128
< .

> keymap.layerNames 6 Dvorak 6 Numpad 8 Function 0
< .
```

Dependencies

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

10.8.44 Leader

Leader keys are a kind of key where when they are tapped, all following keys are swallowed, until the plugin finds a matching sequence in the dictionary, it times out, or fails to find any possibilities. When a sequence is found, the corresponding action is executed, but the processing still continues. If any key is pressed that is not the continuation of the existing sequence, processing aborts, and the key is handled normally.

This behaviour is best described with an example. Suppose we want a behaviour where `LEAD u` starts unicode input mode, and `LEAD u h e a r t` should result in a heart symbol being input, and we want `LEAD u 0 0 e 9 SPC` to input `é`, and any other hex code that follows `LEAD u`, should be handled as-is, and passed to the host. Obviously, we can't have all of this in a dictionary.

So we put `LEAD u` and `LEAD u h e a r t` in the dictionary only. The first will start unicode input mode, the second will type in the magic sequence that results in the symbol, and then aborts the leader sequence processing. With this setup, if we type `LEAD u 0`, then `LEAD u` will be handled first, and start unicode input mode. Then, at the `0`, the plugin notices it is not part of any sequence, so aborts leader processing, and passes the key on as-is, and it ends up being sent to the host. Thus, we covered all the cases of our scenario!

Using the plugin

To use the plugin, one needs to include the header, implement some actions, create a dictionary, and configure the provided `Leader` object to use the dictionary:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Leader.h>

static void leaderA(uint8_t seq_index) {
  Kaleidoscope.serialPort().println("leaderA");
}

static void leaderTX(uint8_t seq_index) {
  Kaleidoscope.serialPort().println("leaderTX");
}

static const kaleidoscope::plugin::Leader::dictionary_t leader_dictionary[] PROGMEM =
  LEADER_DICT({LEADER_SEQ(LEAD(0), Key_A), leaderA},
              {LEADER_SEQ(LEAD(0), Key_T, Key_X), leaderTX});

KALEIDOSCOPE_INIT_PLUGINS(Leader);

void setup() {
  Kaleidoscope.serialPort().begin(9600);

  Kaleidoscope.setup();

  Leader.dictionary = leader_dictionary;
}
```

The dictionary is made up of a list of keys, and an action callback. Using the `LEADER_DICT` and `LEADER_SEQ` helpers is recommended. The dictionary *must* be marked `PROGMEM`!

Plugin methods

The plugin provides the `Leader` object, with the following methods and properties:

`.dictionary`

Set this property to the dictionary `Leader` should use. The dictionary is an array of `kaleidoscope::plugin::Leader::dictionary_t` elements. Each element is made up of two elements, the first being a list of keys, the second an action to perform when the sequence is found.i

The dictionary *MUST* reside in `PROGMEM`.

`.reset()`

Finishes the leader sequence processing. This is best called from actions that are final actions, where one does not wish to continue the leader sequence further in the hopes of finding a longer match.

`.setTimeout(ms)`

The number of milliseconds to wait before a sequence times out. Once the sequence timed out, if there is a partial match with an action, that will be performed, otherwise the Leader sequence will simply reset.

Defaults to 1000.

Dependencies

- Kaleidoscope-Ranges

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.45 MacroSupport

This plugin provides the supplemental key array used by the Macros and DynamicMacros plugins, and is necessary for the proper functioning of those plugins.

Using the plugin

Any firmware sketch that uses either Macros or DynamicMacros automatically includes this plugin, so there's no need to add it explicitly. If your sketch doesn't require either type of Macros key, however, you can still make use of the MacroSupport plugin's helper methods (`tap()`, `press()`, *et al*). In that case, you should include the MacroSupport header file, and include it in `KALEIDOSCOPE_INIT_PLUGINS()`:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-MacroSupport.h>

// Other plugin code that calls `MacroSupport.press()` (for example)
```

(continues on next page)

(continued from previous page)

```
KALEIDOSCOPE_INIT_PLUGINS(
  MacroSupport,
  // Other plugin(s) that make use of MacroSupport
);

void setup() {
  Kaleidoscope.setup ();
}
```

Plugin methods

The plugin provides a `MacroSupport` object, which contains a supplemental array of virtual keys that it adds to USB Keyboard reports. Other plugins and user code can interact with it via the following methods:

`.press(key)`

Sends a key press event for `key`, and will keep that virtual key active in the supplemental virtual keys array.

`.release(key)`

Sends a key release event for `key`, and removes it from the supplemental virtual keys array.

`.clear()`

Releases all active virtual keys held by MacroSupport. This both empties the supplemental keys array (see above) and sends a release event for each key stored there.

`.tap(key)`

Sends an immediate press and release event for `key` with no delay, using an invalid key address. This method doesn't actually use the supplemental keys array, but is provided here for convenience and simplicity.

It is not necessary to use either the Macros (or DynamicMacros) to make use of MacroSupport. When using it with custom code, however, please remember that the supplemental active keys array it provides will be shared by all clients (e.g. Macros, user-defined Leader or TapDance functions), so if you want more than one of those clients to be active simultaneously, be aware that calles to `MacroSupport.clear()` will affect all of them, not just the caller.

10.8.46 Macros

Macros are a standard feature on many keyboards and Kaleidoscope-powered ones are no exceptions. Macros are a way to have a single key-press do a whole lot of things under the hood: conventionally, macros play back a key sequence, but with Kaleidoscope, there is much more we can do. Nevertheless, playing back a sequence of events is still the primary use of macros.

Playing back a sequence means that when we press a macro key, we can have it play pretty much any sequence. It can type some text for us, or invoke a complicated shortcut - the possibilities are endless!

In Kaleidoscope, macros are implemented via this plugin. You can define upto 256 macros.

Using the plugin

To use the plugin, we need to include the header, initialize the plugins with `KALEIDOSCOPE_INIT_PLUGINS()`, place macros on the keymap, and create a special handler function (`macroAction()`) that will determine what happens when macro keys are pressed. It is best illustrated with an example:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Macros.h>

// Give a name to the macros!
enum {
  MACRO_MODEL01,
  MACRO_HELLO,
  MACRO_SPECIAL,
};

// Somewhere in the keymap:
M(MACRO_MODEL01), M(MACRO_HELLO), M(MACRO_SPECIAL)

// later in the Sketch:
const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {
  case MACRO_MODEL01:
    if (keyToggledOn(event.state)) {
      return MACRO(I(25),
                   D(LeftShift), T(M), U(LeftShift), T(O), T(D), T(E), T(L),
                   T(Spacebar),
                   W(100),
                   T(0), T(1) );
    }
    break;
  case MACRO_HELLO:
    if (keyToggledOn(event.state)) {
      return Macros.type(PSTR("Hello "), PSTR("world!"));
    }
    break;
  case MACRO_SPECIAL:
    if (keyToggledOn(event.state)) {
      // Do something special
    }
    break;
  }
```

(continues on next page)

(continued from previous page)

```
  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(Macros);

void setup() {
  Kaleidoscope.setup ();
}
```

Keymap markup

`M(id)`

Places a macro key on the keymap, with the `id` number (0 to 255) as identifier. Whenever this key has to be handled, the `macroAction` overrideable function will be called, with the identifier and key state as arguments.

It is recommended to give a *name* to macro ids, by using an `enum`.

Plugin methods

The plugin provides a `Macros` object, with the following methods and properties available:

`.play(macro)`

Plays back a macro, where a macro is a sequence created with the `MACRO()` helper discussed below. This method will be used by the plugin to play back the result of the `macroAction()` method, but is used rarely otherwise.

The `macro` argument must be a sequence created with the `MACRO()` helper! For example:

Macros.play(MACRO(D(LeftControl), D(LeftAlt), D(Spacebar), U(LeftControl), U(LeftAlt), U(Spacebar)));

`.type(strings...)`

In cases where we only want to type some strings, it is far more convenient to use this method: we do not have to use the `MACRO()` helper, but just give this one a set of strings, and it will type them for us on the keyboard. We can use as many strings as we want, and all of them will be typed in order.

Each string is limited to a sequence of printable ASCII characters. No international symbols, or unicode, or anything like it: just plain ASCII.

Each of `strings` arguments must also reside in program memory, and the easiest way to do that is to wrap the string in a `PSTR()` helper. See the program code at the beginning of this documentation for an example!

`.press(key)/.release(key)`

Used in `Macros.play()`, these methods press virtual keys in a small supplemental `Key` array for the purpose of keeping keys active for complex macro sequences where it's important to have overlapping key presses.

`Macros.press(key)` sends a key press event, and will keep that virtual key active until either `Macros.release(key)` is called, or a Macros key is released. If you use `Macros.press(key)` in a macro, but also change the value of `event.key`, you will need to make sure to also call `Macros.release(key)` at some point to prevent that key from getting "stuck" on.

`.clear()`

Releases all virtual keys held by macros. This both empties the supplemental `Key` array (see above) and sends a release event for each key stored there.

`.tap(key)`

Sends an immediate press and release event for `key` with no delay, using an invalid key address.

Macro helpers

Macros need to be able to simulate key down and key up events for any key - even keys that may not be on the keymap otherwise. For this reason and others, we need to define them in a special way, using the `MACRO` helper.

`MACRO(steps...)`

Defines a macro, that is built up from `steps` (explained below). The plugin will iterate through the sequence, and re-play the steps in order.

Note: In older versions of the Macros plugin, the sequence of steps had to end with a special step called END. This is no longer required. Existing macros that end with END will still work correctly, but new code should not use END; usage of END is deprecated.

`MACRO` steps

Macro steps can be divided into the following groups:

Delays

- `I(millis)`: Sets the interval between steps to `millis`. By default, there is no delay between steps, and they are played back as fast as possible. Useful when we want to see the macro being typed, or need to slow it down, to allow the host to process it.
- `W(millis)`: Waits for `millis` milliseconds. For dramatic effects.

Key events

Key event steps have three variants: one that prefixes its argument with `Key_`, one that does not, and a third that allows for a more compact - but also more limited - representation. The first are the `D`, `U`, and `T` variants, the second are `Dr`, `Ur`, and `Tr`, and the last variant are `Dc`, `Uc`, and `Tc`. In most cases, one is likely use normal keys for the steps, so the `D`, `U`, and `T` steps apply the `Key_` prefix. This allows us to write `MACRO(T(X))` instead of `MACRO(Tr(Key_X))` - making the macro definition shorter, and more readable.

The "raw" variants (`Dr`/`Ur`/`Tr`) use the full name of the `Key` object, without adding the `Key_` prefix to the argument given. `Tr(Key_X)` is the same as `T(X)`.

The "compact" variants (`Dc`/`Uc`/`Tc`) prefix the argument with `Key_` too, but unlike `D`, `U`, and `T`, they ignore the `flags` component of the key, and as such, are limited to ordinary keys. Mouse keys, consumer- or system keys are not supported by this compact representation.

- `D(key)`, `Dr(key)`, `Dc(key)`: Simulates a key being pressed (pushed down).
- `U(key)`, `Ur(key)`, `Uc(key)`: Simulates a key being released (going up).
- `T(key)`, `Tr(key)`, `Tc(key)`: Simulates a key being tapped (pressed first, then released).

Key sequences

One often used case for macros is to type longer sequences of text. In these cases, assembling the macro step by step using the events described above is verbose both in source code, and compiled. For this reason, the plugin provides two other actions, both of which take a sequence of keys, and will tap all of them in order.

- `SEQ(K(key1), K(key2), ...)`: Simulates all the given keys being tapped in order, with the currently active interval waited between them. The keys need to be specified by their full name.
- `SEQc(Kc(key1), Kc(key2), ...)`: Same as `SEQ()`, but the keys are prefixed with `Key_`, and they ignore the `flags` component of a key, and as such, are limited to ordinary keys.

Overrideable functions

`macroAction(uint8_t macro_id, KeyEvent &event)`

The `macroAction` method is the brain of the macro support in Kaleidoscope: this function tells the plugin what sequence to play when given a macro index and a key state.

It should return a macro sequence, or `MACRO_NONE` if nothing is to be played back.

Limitations

Due to technical and practical reasons, `Macros.type()` assumes a QWERTY layout on the host side, and so do all other parts that work with keycodes. If your operating system is set to a different layout, the strings and keycodes will need to be adjusted accordingly.

Dependencies

- Kaleidoscope-MacroSupport

10.8.47 MagicCombo

The `MagicCombo` extension provides a way to perform custom actions when a particular set of keys are held down together. The functionality assigned to these keys are not changed, and the custom action triggers as long as all keys within the set are pressed. The order in which they were pressed do not matter.

This can be used to tie complex actions to key chords.

Using the extension

To use the extension, we must include the header, create actions for the magic combos we want to trigger, and set up a mapping:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Macros.h>
#include <Kaleidoscope-MagicCombo.h>

enum { KIND_OF_MAGIC };

void kindOfMagic(uint8_t combo_index) {
  Macros.type(PSTR("It's a kind of magic!"));
}

USE_MAGIC_COMBOS(
[KIND_OF_MAGIC] = {
  .action = kindOfMagic,
  .keys = {R3C6, R3C9} // Left Fn + Right Fn
});

KALEIDOSCOPE_INIT_PLUGINS(MagicCombo, Macros);

void setup() {
  Kaleidoscope.setup();
}
```

It is recommended to use the `RxCy` macros of the core firmware to set the keys that are part of a combination.

Plugin properties

The extension provides a `MagicCombo` singleton object, with the following method:

`.setMinInterval(min_interval)`

Restrict the magic action to fire at most once every `min_interval` milliseconds.

Defaults to 500.

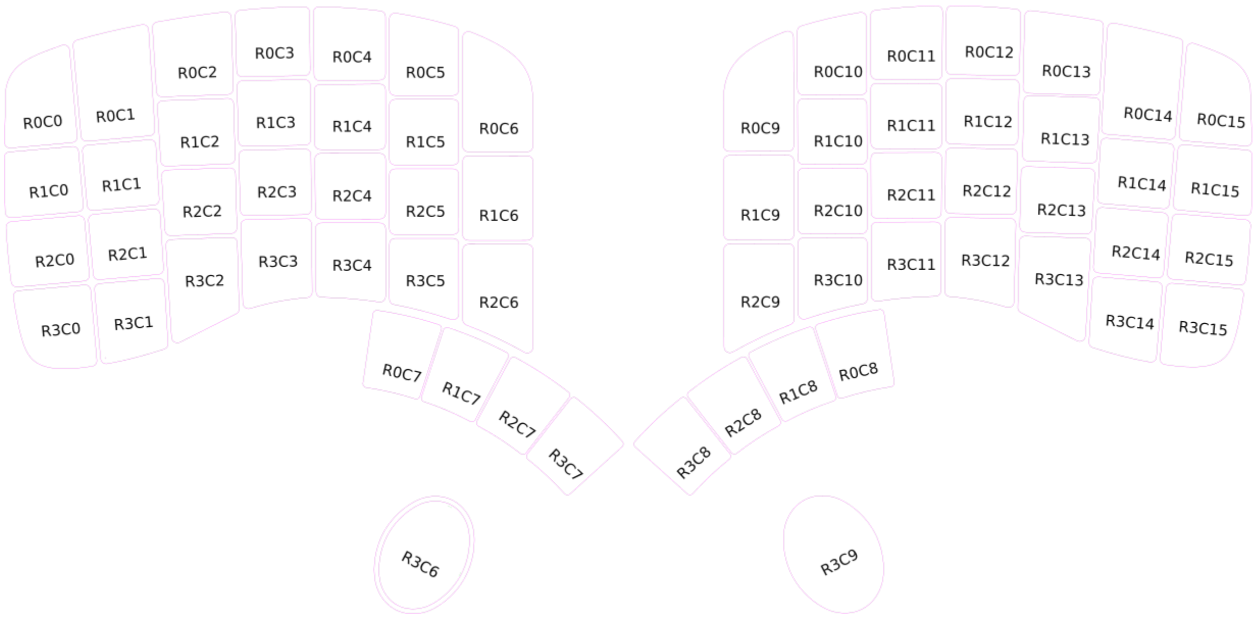
Plugin callbacks

Whenever a combination is found to be held, the plugin will trigger the specified action, which is just a regular method with a single `uint8_t` argument: the index of the magic combo. This function will be called repeatedly (every `min_interval` milliseconds) while the combination is held.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

`RxCy` coordinates for a Model01:



10.8.48 MouseKeys

Have you ever wanted to control the mouse cursor from the comfort of your keyboard? With this plugin, you can. While it may not replace the mouse in all situations, there are plenty of cases where one will not have to lift their hands off the keyboard just to nudge the mouse cursor away a little.

Of course, there are a lot more one can do with the plugin than to nudge the cursor! Mouse keys are provided for all four *and* diagonal movement; mouse buttons; and a unique warping mechanism too. And not only these: the speed of the cursor, the mouse wheel, and that of acceleration can all be configured to match one's desired behaviour.

Using the plugin

To use the plugin, simply include the header in your Sketch, tell the firmware to use the `MouseKeys` object, and place mouse keys on your keymap. It is best illustrated with an example:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-MouseKeys.h>

// Somewhere in the keymap:
Key_mouseUp, Key_mouseDn, Key_mouseL, Key_mouseR,
Key_mouseBtnL, Key_mouseBtnR

KALEIDOSCOPE_INIT_PLUGINS(
  MouseKeys,
  MouseKeysConfig // optionally add this to make the configuration runtime editable via␣
↪focus commands.
);

void setup() {
  Kaleidoscope.setup();
}
```

Keys provided by the plugin

The plugin provides a number of keys one can put on the keymap, that allow control of the mouse. They can be divided into a few groups:

Mouse buttons

Mouse button keys are straightforward; pressing one is the same as pressing the corresponding button on a physical mouse. You can hold a mouse button key to perform drag gestures, as you might expect. MouseKeys supports five mouse buttons: left, right, middle, previous, and next.

- `Key_mouseBtnL`, `Key_mouseBtnM`, `Key_mouseBtnR`, `Key_mouseBtnP`, `Key_mouseBtnN`: The left, middle, right, previous, and next mouse buttons, respectively.

Cursor movement

When a cursor movement key is pressed, the mouse cursor will begin to move slowly, then accelerate to full speed. Both the full speed and the time it takes to reach full speed are configurable.

The cursor movement keys are as follows:

- `Key_mouseUp`, `Key_mouseDn`, `Key_mouseL`, `Key_mouseR`: Move the cursor up, down, left, or right, respectively.
- `Key_mouseUpL`, `Key_mouseUpR`, `Key_mouseDnL`, `Key_mouseDnR`: Move the cursor up-left, up-right, down-left, down-right, respectively.

Scroll wheels

Controlling the scroll wheel is similarly simple. It does not have acceleration, but one can control the speed with the `MouseKeys.setScrollInterval()` function, which controls the length of time between scroll events.

- `Key_mouseScrollUp`, `Key_mouseScrollDn`: Scroll the mouse wheel up or down, respectively.
- `Key_mouseScrollL`, `Key_mouseScrollR`: Scroll the mouse wheel left or right, respectively.

Warping

Warping is one of the most interesting features of the plugin, and is a feature unique to Kaleidoscope, as far as we can tell. The warping keys position the mouse cursor within a sector of the screen on first press, and any subsequent taps will warp within the previously selected sector. For example, pressing the north-west warp key twice will first jump to the middle of the north-west sector of your screen, then select the north-west sector of that, and jump to the middle of it.

To stop warping, use any other mouse key, or hit the "warp end" key.

Warp grid size

The warp grid size determines how MouseKeys partitions the screen to select the next position to jump to when pressing a warp key. The plugin provides two grid sizes to choose from: a *2x2* grid that splits the screen into quadrants, and a *3x3* grid with nine cells similar to a navigation feature included with some speech recognition software. By default, the plugin splits the screen into the 2x2 grid.

To change the warp grid size, call the plugin's `setWarpGridSize()` method:

```
MouseKeys.setWarpGridSize(MOUSE_WARP_GRID_3X3);
```

2x2 grid

As described above, MouseKeys warps the pointer using a grid model that reflects locations on the screen. By default, the plugin uses a 2x2 grid. To understand how warping works, examine this diagram of a screen split into that 2x2 grid:

```
+-----------------------|-----------------------+
|           |           |                       |
|     G     |    tab    |                       |
|           |           |                       |
|-----------|-----------|          tab          |
|           |           |                       |
|     B     |    esc    |                       |
|           |           |                       |
+-----------------------|-----------------------+
|                       |                       |
|                       |                       |
|                       |                       |
|           B           |          esc          |
|                       |                       |
|                       |                       |
|                       |                       |
+-----------------------|-----------------------+
```

Each quadrant is labed with a key that, when pressed, moves the mouse pointer to the center of that quadrant. With this layout, pressing G warps the pointer to the top-left quadant. Then, the plugin "zooms" into that sector with a smaller grid so that the next warp key pressed jumps the pointer more precisely within the sector. In this case, if we press esc next, the pointer warps to the bottom-right corner within that quadrant.

The warping keys for the 2x2 grid are the following:

- `Key_mouseWarpNW`, `Key_mouseWarpNE`, `Key_mouseWarpSW`, `Key_mouseWarpSE`: Warp towards the north-west, north-east, south-west, or south-east quadrants, respectively.
- `Key_mouseWarpEnd`: End the warping sequence, resetting it to the default state. Using any of the warping keys after this will start from the whole screen again.

3x3 grid

A 3x3 warp grid assigns a key to each of nine sectors of the screen. The next diagram shows a screen with a key label that warps to each sector. As we can see, pressing W warps the pointer into the top-left sector, and pressing V warps to the bottom-right corner within that sector:

```
+-----------------|-----------------|-----------------+
|  W  |  E  |  R  |                 |                 |
|-----|-----|-----|                 |                 |
|  S  |  D  |  F  |        E        |        R        |
|-----|-----|-----|                 |                 |
|  X  |  C  |  V  |                 |                 |
+-----------------|-----------------|-----------------+
|                 |                 |                 |
|                 |                 |                 |
|        S        |        D        |        F        |
|                 |                 |                 |
|                 |                 |                 |
+-----------------|-----------------|-----------------+
|                 |                 |                 |
|                 |                 |                 |
|        X        |        C        |        V        |
|                 |                 |                 |
|                 |                 |                 |
+-----------------|-----------------|-----------------+
```

To use a 3x3 warp grid, we may need to remap some keys. A suggested warp key mapping is shown below on the left side of a keyboard with a QWERTY layout:

```
   W | E | R  T      A - End Warping      (Key_mouseWarpEnd)
  ---|---|---        W - Warp NW Sector   (Key_mouseWarpNW)
A  S | D | F  G      E - Warp N Sector    (Key_mouseWarpN)
  ---|---|---        R - Warp NE Sector   (Key_mouseWarpNE)
   X | C | V  B      S - Warp W Sector    (Key_mouseWarpW)
                     D - Warp/Zoom Center (Key_mouseWarpIn)
                     F - Warp E Sector    (Key_mouseWarpE)
                     X - Warp SW Sector   (Key_mouseWarpSW)
                     C - Warp S Sector    (Key_mouseWarpS)
                     V - Warp SE Sector   (Key_mouseWarpSE)
                     T - Right Click      (Key_mouseBtnR)
```

(continues on next page)

(continued from previous page)

```
                    G - Left Click       (Key_mouseBtnL)
                    B - Middle Click     (Key_mouseBtnM)
```

This example layout replaces the default directional mouse keys and sets the warp keys in a comfortable position for a warp-only configuration. Of course, a Kaleidoscope user may retain the directional keys and map the warp keys elsewhere according to his or her liking.

A 3x3 warp grid layout contains all of the keys from the 2x2 grid layout with the following additions:

- `Key_mouseWarpN`, `Key_mouseWarpE`, `Key_mouseWarpS`, `Key_mouseWarpW`: Warp towards the north, east, south, and west sectors, respectively.
- `Key_mouseWarpIn`: Warp to the center sector of the grid. The plugin will continue to "zoom" into center of the current cell with each consecutive press of this key.

Plugin methods

The plugin provides a `MouseKeys` object, with the following methods and properties available:

`.setCursorInitSpeed(speed)/.getCursorInitSpeed()`

Controls (or returns) the current starting speed value for mouse cursor movement. When a mouse movement key is pressed, the cursor starts moving at this speed, then accelerates. The number is abstract, but linear, with higher numbers representing faster speeds. Default starting speed is `1`.

`.setCursorBaseSpeed(speed)/.getCursorBaseSpeed()`

Controls (or returns) the current top speed value for mouse cursor movement. When a mouse movement key is pressed, the cursor accelerates until it reaches this speed. The number is abstract, but linear, with higher numbers representing faster speeds. Default full-speed value is `50`.

`.setCursorAccelDuration(duration)/.getCursorAccelDuration()`

Controls (or returns) the current time it takes for the mouse cursor to reach full speed (in milliseconds), starting from when the first movement key is pressed. Default value is `800` ms.

`.setScrollInterval(interval)/.getScrollInterval()`

Controls (or returns) the current scrolling speed, by setting the time between mouse scroll reports (in milliseconds). Default value is `50` ms.

`.setWarpGridSize(size)`

This method changes the size of the grid used for *warping*. The following are valid sizes: `MOUSE_WARP_GRID_2X2`, `MOUSE_WARP_GRID_3X3`

Further reading

There is an *example* that demonstrates how to use this plugin.

10.8.49 NumPad

This is a plugin for Kaleidoscope, that adds a NumPad-specific LED effect and applies it when the numpad layer is active.

Using the extension

To use the plugin, include the header, and tell the firmware to use it:

```
#include "Kaleidoscope-NumPad.h"

KALEIDOSCOPE_INIT_PLUGINS(NumPad);

void setup() {
  Kaleidoscope.setup();

  NumPad.color = CRGB(0, 0, 160); // a blue color
  NumPad.lock_hue = 85; // green
}
```

Plugin methods

The plugin provides the `NumPad` object, with the following properties:

`.color`

This property sets the color that the NumPad keys are highlighted in.

The default is `CRGB(160, 0, 0)`, a red color.

`.lock_hue`

This property sets the color hue that the NumLock LED breathes in.

The default is `170`, a blue hue.

10.8.50 OneShot

One-shots are a new kind of behaviour for your standard modifier and momentary layer keys: instead of having to hold them while pressing other keys, they can be tapped and released, and will remain active until any other key is pressed subject to a time-out.

In short, they turn `Shift, A` into `Shift+A`, and `Fn, 1` to `Fn+1`. The main advantage is that this allows us to place the modifiers and layer keys to positions that would otherwise be awkward when chording. Nevertheless, they still act as normal when held, that behaviour is not lost.

Furthermore, if a one-shot key is double-tapped ie tapped two times in quick succession, it becomes sticky, and remains active until disabled with a third tap. This can be useful when one needs to input a number of keys with the modifier or layer active, and does not wish to hold the key down. If this "stickability" feature is undesirable, it can be unset (and later again set) for individual modifiers/layers. If stickability is unset, double-tapping a one-shot modifier will just restart the timer.

To make multi-modifier, or multi-layer shortcuts possible, one-shot keys remain active if another one-shot of the same type is tapped, so `Ctrl, Alt, b` becomes `Ctrl+Alt+b`, and `L1, L2, c` is turned into `L1+L2+c`. Furthermore, modifiers and other layer keys do not cancel the one-shot effect, either.

Using One-Shot keys

To enter one-shot mode, tap *quickly* on a one-shot key. The next normal (non-one-shot) key you press will have the modifier applied, and then the modifier will automatically turn off. If the Shift key is a one-shot modifier, then hitting `Shift, a, b` will give you `Ab`, *if you hit shift quickly.*

Longish keypresses do not activate one-shot mode. If you press `Shift, a, b`, as above, but hold the Shift key a bit longer, you'll get `ab`.

To enter sticky mode, *tap twice quickly* on a one-shot key. The modifier will now stay on until you press it again. Continuing the `Shift` example, tapping `Shift, Shift` *quickly* and then `a, b, c, Shift, d, e, f` will give you `ABCdef`.

This can be a bit tricky; combining this plugin with LED-ActiveModColor will help you understand what state your one-shot is in; when a one-shot key is active, it will have a yellow LED highlight; when sticky, a red highlight. When it is in a "held" state, but will be deactivated when released like any non-one-shot key, it will have a white highlight. (These colors are configurable.)

Using the plugin

After adding one-shot keys to the keymap, all one needs to do, is enable the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-OneShot.h>

// somewhere in the keymap...
OSM(LeftControl), OSL(_FN)

KALEIDOSCOPE_INIT_PLUGINS(OneShot);

void setup() {
  Kaleidoscope.setup();
}
```

To enable configuring the plugin via Focus (including via Chrysalis), one will also need the `OneShotConfig` plugin enabled in addition.

Keymap markup

There are two macros the plugin provides:

`OSM(mod)`

A macro that takes a single argument, the name of the modifier: `LeftControl`, `LeftShift`, `LeftAlt`, `LeftGui` or their right-side variant. When marked up with this macro, the modifier will act as a one-shot modifier.

`OSL(layer)`

Takes a layer number as argument, and sets up the key to act as a one-shot layer key.

Please note that while `Kaleidoscope` supports more, one-shot layers are limited to 8 layers only.

In addition, there is a special key:

`Key_MetaSticky`

A key that behaves like a one-shot key, but while active, it makes other keys that are pressed become sticky, just like double-tapped one-shot keys.

Plugin methods

The plugin provides one object, `OneShot`, which implements both one-shot modifiers and one-shot layer keys. It has the following methods:

Configuration methods: Timeouts

`.setTimeout(timeout)`

OneShot keys will remain active after they're pressed for `timeout` milliseconds (or until a subsequent non-oneshot key is pressed). The default value is 2500 (2.5 seconds).

`.setHoldTimeout(timeout)`

If a one-shot key is held for longer than `timeout` milliseconds, it will behave like a normal key, and won't remain active after it is released. The default value is 250 (1/4 seconds).

`.setDoubleTapTimeout(timeout)`

If a one-shot key is double-tapped (pressed twice in a row) in less than `timeout` milliseconds, it wil become sticky, and will remain active until it is pressed a third time. The default value is -1, which indicates that it should use the same timeout as `.setTimeout()`.

Configuration methods: Stickability

`.enableStickability(key...)`

`.disableStickability(key...)`

Enables/disables stickability for all keys listed. The keys should all be OneShot keys, modifier keys, or layer-shift keys, as specified on the keymap. For example: `OneShot.enableStickability(OSM(LeftShift), OSL(1), Key_RightGUI)`. `OneShot.disableStickability(OSM(RighttAlt), OSL(2), ShiftToLayer(4))`.

By default, all OneShot keys are stickable.

`.enableStickabilityForModifiers()`

`.enableStickabilityForLayers()`

`.disableStickabilityForModifiers()`

`.disableStickabilityForLayers()`

Enables/disables stickability for all modifiers and layers, respectively. These are convenience methods for cases where one wants to enable stickability for a group of one-shot keys.

Configuration methods: Automatic one-shot keys

`.enableAutoModifiers()`

`.disableAutoModifiers()`

`.toggleAutoModifiers()`

Enables/disables/toggles auto-oneshot functionality for modifier keys. When enabled, all normal modifier keys, including those with other modifier flags added to them (e.g. `LSHIFT(Key_LeftAlt)`, `Key_Meh`) will be automatically treated as one-shot keys, in addition to dedicated ones like `OSM(LeftGui)`.

`.enableAutoLayers()`

`.disableAutoLayers()`

`.toggleAutoLayers()`

Enables/disables/toggles auto-oneshot functionality for layer shift keys (see above).

`.enableAutoOneShot()`

`.disableAutoOneShot()`

`.toggleAutoOneShot()`

Enables/disables/toggles auto-oneshot functionality for all modifiers and layer shift keys.

Test methods

`.isActive(key_addr)`

Returns `true` if the key at `key_addr` is in an active one-shot state. Note that if a key is still being held, but will be not remain active after it is released, it is not considered to be in a one-shot state, even if it had been earlier.

`.isTemporary(key_addr)`

Returns `true` if the key at `key_addr` is in a temporary one-shot state. Such a key will eventually time out or get deactivated by a subsequent key press.

`.isSticky(key_addr)`

Returns `true` if the key at `key_addr` is in a permanent one-shot state. Such a key will not be deactivated by subsequent keypresses, nor will it time out. It will only be deactivated by pressing it one more time, or by being cancelled by the `cancel()` method (see below).

`.isActive()`

Returns `true` if there are any active one-shot keys. Note: it returns `false` if there are no currently active one-shot keys, but there are keys that were at one time in a one-shot state, but are still being held after that state has been cancelled.

`.isSticky()`

Returns `true` if there are any sticky one-shot keys.

`.isStickable(key)`

Returns `true` if a key of the specified value can be made sticky by double-tapping.

`.isModifier(key)`

Returns `true` if the specified key is a modifier key. This does not include OneShot modifiers (e.g. `OSM(LeftShift)`), but it does include modifiers with additional modifier flags (e.g. `Key_Meh`, `LCTRL(Key_RightGui)`).

`.isLayerShift(key)`

Returns `true` if the specified key is a layer-shift key (e.g. `ShiftToLayer(2)`). OneShot layer keys (e.g. `OSL(5)` are not included).

`.isOneShotKey(key)`

Returns `true` if the specified key is a OneShot modifier or layer-shift key (e.g. `OSM(LeftAlt)`, `OSL(3)`).

Other methods

`.cancel([with_stickies])`

Immediately deactivates the one-shot status of any *temporary* one-shot keys. Any keys still being physically held will continue to function as normal modifier/layer-shift keys.

If `with_stickies` is `true` (the default is `false`), *sticky* one-shot keys will also be deactivated, in the same way.

Deprecated methods

The following methods have been deprecated, and should no longer be used, if possible. These functions made more sense when OneShot was based on `Key` values; it has since be rewritten to be based on `KeyAddr` values.

`.inject(key, key_state)`

Finds an idle key on the keyboard, and turns it into a one-shot key. When OneShot was based on `Key` values, this made more sense, as it didn't need a valid `KeyAddr` to work. Since the main purpose of this method was to enable the triggering of multiple one-shot modifiers with a single key, it is much better to use automatic one-shot modifiers, if possible, because then it's not necessary to use a Macro to configure.

`.isModifierActive(key)`

Returns `true` if a keymap cache entry with the current value of `key` is active (one-shot, sticky, or held). This should be a function that is not specific to OneShot.

`.isActive(key)`

Returns `true` if a keymap cache entry with the current value of `key` is in an active one-shot state. Please use `.isActive(key_addr)` instead.

`.isSticky(key)`

Returns `true` if a keymap cache entry with the current value of `key` is in a sticky one-shot state. Please use `.isSticky(key_addr)` instead.

`.isPressed()`

Returns `false`. OneShot doesn't need to keep track of whether or not a one-shot key is still pressed any more. This function was mainly used by LED-ActiveModColor, which no longer needs it.

Focus commands

When the `OneShotConfig` plugin is enabled, the following Focus commands become available:

`.timeout`

`.hold_timeout`

`.double_tap_timeout`

These correspond to the `.setTimeout()`, `.setHoldTimeout()`, and `.setDoubleTapTimeout()` methods, and can be used to query or set the respective timeout value. When used without an argument, the command will print the current timeout value. When used with one, it will update it.

`.auto_modifiers`

`.auto_layers`

Corresponds to the `.enableAutoModifiers()` and `.enableAutoLayers()` methods. Used without an argument, the command will print the current status of the setting, otherwise it will update it.

A value of `1` means the setting is enabled, a value of `0` means it is disabled.

`.stickable_keys`

Can be used to query or set the bitmap used for controlling the stickability of the oneshot modifier and layer keys. Constructing the bitmap is complicated, and is best done through Chrysalis.

Dependencies

- Kaleidoscope-Ranges

If the `OneShotConfig` plugin is enabled, additional dependencies are:

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.51 OneShot Meta Keys

This plugin provides support for two special OneShot keys: `OneShot_MetaStickyKey` & `OneShot_ActiveStickyKey`, each of which can be used to make any key on the keyboard (not just modifiers and layer shift keys) "sticky", so that they remain active even after the key has been released. These are both `Key` values that can be used as entries in your sketch's keymap.

Any keys made sticky in this way can be released just like OneShot modifier keys, by tapping them again to cancel the effect.

The `OneShot_MetaStickyKey`

This special OneShot key behaves like other OneShot keys, but its affect is to make the next key pressed sticky. Tap `OneShot_MetaStickyKey`, then tap `X`, and `X` will become sticky. Tap `X` again to deactivate it.

Double-tapping `OneShot_MetaStickyKey` will make it sticky, just like any other OneShot key. A third tap will release the key.

The `OneShot_ActiveStickyKey`

This special key doesn't act like a OneShot key, but instead makes any key(s) currently held (or otherwise active) sticky. Press (and hold) `X`, tap `OneShot_ActiveStickyKey`, then release `X`, and `X` will stay active until it is tapped again to deactivate it.

Using the plugin

To use the plugin, just include one of the two special OneShot keys somewhere in your keymap, and add both OneShot and OneShotMetaKeys to your sketch:

```
#include <Kaleidoscope-OneShot.h>
#include <Kaleidoscope-OneShotMetaKeys.h>

// somewhere in the keymap...
OneShot_MetaStickyKey, OneShot_ActiveStickyKey

KALEIDOSCOPE_INIT_PLUGINS(OneShot, OneShotMetaKeys);
```

Important note: OneShotMetaKeys *must* be registered after OneShot in `KALEIDOSCOPE_INIT_PLUGINS()` in order to function properly.

Dependencies

- Kaleidoscope-OneShot
- Kaleidoscope-Ranges

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.52 PrefixLayer

The `PrefixLayer` plugin allows you to easily create a keyboard layer designed for use with programs that use a prefix key, such as tmux or screen. When a key in a prefix layer is pressed, the prefix is injected first, then the key in that layer is pressed.

Using the plugin

You will need to define a keymap layer and configure the plugin to use that layer with a prefix key. You can then include the plugin's header and set the `.prefix_layers` property.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-PrefixLayer.h>

enum {
  PRIMARY,
  TMUX,
};  // layers
```

(continues on next page)

(continued from previous page)

```
static const kaleidoscope::plugin::PrefixLayer::Entry prefix_layers[] PROGMEM = {
  kaleidoscope::plugin::PrefixLayer::Entry(TMUX, LCTRL(Key_B)),
};

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, PrefixLayer);

void setup() {
  Kaleidoscope.setup();
  PrefixLayer.prefix_layers = prefix_layers;
  PrefixLayer.prefix_layers_length = 1;
}
```

Plugin methods

The plugin provides a `PrefixLayer` object, which has the following methods and properties:

`.prefix_layers`

A `kaleidoscope::plugin::PrefixLayer::Entry` array that maps layers to prefix keys. The `Entry` constructor accepts `Entry(layer_number, prefix_key)`. This array must be stored in `PROGMEM` as shown above.

Defaults to an empty array.

`.prefix_layers_length`

Length of the `prefix_layers` array.

Defaults to *0*

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.53 Qukeys

Concept

This Kaleidoscope plugin allows you to overload keys on your keyboard so that they produce one keycode (i.e. symbol) when tapped, and a different keycode – most likely a modifier (e.g. `shift` or `alt`) – when held. The name is a play on the term *qubit*; a qukey is a "quantum key". When it is first pressed it is in a superposition of states until some event determines which state it ends up in. While a qukey is in this indeterminate state, its key press event and any subsequent key presses are delayed until something determines the qukey's ultimate state.

Most likely, what determines the qukey's state (*primary* or *alternate*) is the release of a key; if the qukey is released before a subsequent key, it will take on its primary value (most likely a printable character), but if the subsequent key is released first, it will take on its alternate value (usually a modifier).

Qukeys is designed to make it practical to use these overloaded keys on the home row, where similar designs have historically been problematic. For some typists (particularly those who are accustomed to rolling over from modifiers to modified keys, rather than deliberately holding the modifier until the subsequent key has been released), this may still not work perfectly with Qukeys, but some people have reported good results with home-row qukeys.

Setup

- Include the header file:

```
#include <Kaleidoscope-Qukeys.h>
```

- Use the plugin in the `KALEIDOSCOPE_INIT_PLUGINS` macro:

```
KALEIDOSCOPE_INIT_PLUGINS(Qukeys);
```

- Define some `Qukeys` of the format `Qukey(layer, key_addr, alternate_key)`. Layers and key addresses are all zero-indexed, in key addresses rows are top to bottom and columns are left to right:
- For the Keyboardio Model 01, key coordinates refer to this header file.
- For the Keyboardio Model 100, key coordinates refer to this header file.

```
QUKEYS(
    // left-side modifiers
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 1), Key_LeftGui),      // A
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 2), Key_LeftAlt),      // S
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 3), Key_LeftControl),  // D
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 4), Key_LeftShift),    // F
    // left-side layer shifts
    kaleidoscope::plugin::Qukey(0, KeyAddr(3, 3), ShiftToLayer(NUMPAD)),    // C
    kaleidoscope::plugin::Qukey(0, KeyAddr(3, 4), ShiftToLayer(FUNCTION)),  // V
```

`Qukeys` will work best if it's the first plugin in the `INIT()` list, because when typing overlap occurs, it will (temporarily) mask keys and block them from being processed by other plugins. If those other plugins handle the keypress events first, it may not work as expected. It doesn't *need* to be first, but if it's `INIT()`'d after another plugin that handles typing events, especially one that sends extra keyboard HID reports, it is more likely to generate errors and out-of-order events.

Configuration

`.setHoldTimeout(timeout)`

Sets the time (in milliseconds) after which a qukey held on its own will take on its alternate state. Note: this is not the primary determining factor for a qukey's state. It is not necessary to wait this long before pressing a key that should be modified by the qukey's alternate value. The primary function of this timeout is so that a qukey can be used as a modifier for an separate pointing device (i.e. `shift` + `click`).

Defaults to `250`.

`.setMaxIntervalForTapRepeat(timeout)`

Sets the time (in milliseconds) that limits the tap-repeat window. If the same qukey is pressed, released, and pressed again within this timeframe, then held, Qukeys will turn it into a single press and hold event, using the primary key value (which cannot otherwise be held). If the second press is also a tap, and the two *release* events occur within the same timeframe, it will instead be treated as a double tap (of the primary key value).

To effectively shut off the tap-repeat feature, set this value to `0`. The maximum value is `255`; anything higher than `250` could result in key repeat being triggered on the host before Qukeys determines whether it's a tap-repeat or a double-tap sequence, because most systems delay the key repeat by 500 ms.

Defaults to `200`.

`.setOverlapThreshold(percentage)`

This sets a variable that allows the user to roll over from a qukey to a subsequent key (i.e. the qukey is released first), and still get the qukey's alternate (modifier) state.

The `percentage` parameter should be between `1` and `100` (`75` means 75%), and represents the fraction of the *subsequent* key press's duration that overlaps with the qukey's press. If the subsequent key is released soon enough after the qukey is released, the percentage overlap will be high, and the qukey will take on its alternate (modifier) value. If, on the other hand, the subsequent key is held longer after the qukey is released, the qukey will take on its primary (non-modifier) value.

Setting `percentage` to 100% turns off the grace period, so you can't reliably get either output if you release the two keys simultaneously. That means the subsequent key must be released before the qukey for the release-order condition to trigger making the qukey take on its alternate state.

Setting `percentage` to a low value (e.g. `30`) will result in a longer grace period. If you're getting primary values when you intended modifiers, try decreasing this setting. If, on the other hand, you start getting modifiers when you intend primary values, try increasing this setting. If you're getting both, the only solution is to change your typing habits, unfortunately.

Defaults to `80`.

`.setMinimumHoldTime(min_hold_time)`

Sets the minimum amount of time (in milliseconds) a qukey must be held before it is allowed to resolve to its alternate `Key` value. Use this if you find that you're getting unintended alternate values (i.e. modifiers) while typing on home-row qukeys, despite setting the overlap threshold (see above) to 100%. It may mean that you'll need to slow down when using Qukeys to get modifiers, however.

Defaults to `50` (milliseconds).

`.setMinimumPriorInterval(min_interval)`

Sets the minimum amount of time (in milliseconds) that must pass between the press event of a prior (non-modifier) key and the press of a qukey required to make that qukey eligible to take on it's alternate state. This is another measure that can be taken to prevent unintended modifiers while typing fast.

Defaults to `75` (milliseconds).

`.activate()`

`.deactivate()`

`.toggle()`

Activate/deactivate `Qukeys` plugin.

DualUse key definitions

In addition to normal `Qukeys` described above, Kaleidoscope-Qukeys also treats DualUse keys in the keymap as `Qukeys`. This makes `Qukeys` a drop-in replacement for the `DualUse` plugin, without the need to edit the keymap.

The plugin provides a number of macros one can use in keymap definitions:

`CTL_T(key)`

A key that acts as the *left* `Control` when held, or used in conjunction with other keys, but as `key` when tapped in isolation. The `key` argument must be a plain old key, and can't have any modifiers or anything else applied.

`ALT_T(key)`

A key that acts as the *left* `Alt` when held, or used in conjunction with other keys, but as `key` when tapped in isolation. The `key` argument must be a plain old key, and can't have any modifiers or anything else applied.

`SFT_T(key)`

A key that acts as the *left* `Shift` when held, or used in conjunction with other keys, but as `key` when tapped in isolation. The `key` argument must be a plain old key, and can't have any modifiers or anything else applied.

`GUI_T(key)`

A key that acts as the *left* `GUI` when held, or used in conjunction with other keys, but as `key` when tapped in isolation. The `key` argument must be a plain old key, and can't have any modifiers or anything else applied.

`MT(mod, key)`

A key that acts as `mod` when held, or used in conjunction with other keys, but as `key` when tapped in isolation. The `key` argument must be a plain old key, and can't have any modifiers or anything else applied. The `mod` argument can be any of the modifiers, *left* or *right* alike.

`LT(layer, key)`

A key that momentarily switches to `layer` when held, or used in conjunction with other keys, but as `key` when tapped in isolation. The `key` argument must be a plain old key, and can't have any modifiers or anything else applied.

DualUse keys are more limited than `Qukey` definitions, which can contain any valid `Key` value for both the primary and alternate keys, but they take up less space in program memory, and are just as functional for typical definitions.

SpaceCadet Emulation

It is possible to define a `Qukey` on a key with a *primary* value that is a modifier. In this case, the qukey is treated specially, and the *primary* value is used when the key is held, rather than the alternate value. The *alternate* value is only used if the qukey is tapped on its own, without rolling over to any other key. This is a reasonable facsimile of the behaviour of the SpaceCadet plugin, and is much more suitable for keys that are mainly used as modifiers, with an additional "tap" feature.

In addition to working this way on keyboard modifiers (`shift`, `control`, *et al*), this works for keys that are primarily layer shift keys (e.g. `ShiftToLayer(N)`).

As an added bonus, if Qukeys is deactivated, such a key reverts to being a modifier, because that's what's in the keymap.

The Wildcard Layer

There is a special value (`Qukeys::layer_wildcard`) that can be used in place of the layer number in the definition of a `Qukey`. This will define a qukey with the given alternate value on all layers, regardless of what the primary value is for that key on the top currently active layer.

Tap-Repeat Behaviour

If a qukey is tapped, then immediately pressed and held, Qukeys will turn that sequence of events into a single press and hold of the primary key value (whereas merely holding the key yeilds the alternate value). This is particularly useful on macOS apps that use Apple's Cocoa input system, where holding a key gives the user access to a menu of accented characters, rather than merely repeating the same character until the key is released.

Design & Implementation

When a qukey is pressed, it doesn't immediately add a corresponding keycode to the HID report; it adds that key to a queue, and waits until one of three things happens:

1. the qukey is released
2. a subsequently-pressed key is released
3. a time limit is reached

Until one of those conditions is met, all subsequent keypresses are simply added to the queue, and no new reports are sent to the host. Once a condition is met, the qukey is flushed from the queue, and so are any subsequent keypresses (up to, but not including, the next qukey that is still pressed).

Basically, if you hold the qukey, then press and release some other key, you'll get the alternate keycode (probably a modifier) for the qukey, even if you don't wait for a timeout. If you're typing quickly, and there's some overlap between two keypresses, you won't get the alternate keycode, and the keys will be reported in the order that they were pressed – as long as the keys are released in the same order they were pressed.

The time limit is mainly there so that a qukey can be used as a modifier (in its alternate state) with a second input device (e.g. a mouse). It can be quite short (200ms is probably short enough) – as long as your "taps" while typing are shorter than the time limit, you won't get any unintended alternate keycodes.

Further reading

The *example* can help to learn how to use this plugin.

10.8.54 Ranges

kaleidoscope::ranges enum

This plugin defines the `ranges` enum that many plugins use.

To explain its purpose, first a brief digression to explain how Kaleidoscope implements keys.

Keys in Kaleidoscope are defined as follows:

```
// in Kaleidoscope/src/key_defs.h
typedef union Key_ {

  struct {
    uint8_t keyCode;
    uint8_t flags;
  };
  uint16_t raw;
// bunch of operator overloads elided...
} Key;
```

That is, a key is a 16-bit integer that, by default, has 8 bits for the keyCode and 8 bits for flags. For normal keypresses, this is straightforward: the 8-bit keyCode is the normal HID code corresponding to a normal keyboard key and the flags are used to indicate the modifiers being held.

However, many plugins want to be able to make key presses do something special and need some way of indicating that a key is not just an ordinary key. To do this, we take advantage of the fact that our flags field is 8 bits, but there are only five normal modifiers (control, shift, GUI, right alt, and left alt).

In this case, by setting the high bit of the flags field, we indicate that this is a reserved key and isn't going to be interpreted normally.

This way, a plugin can make a key be sent with the high bit of the flags field set, then it is free to use the rest of the 16 bits as it sees fit and be assured that it won't conflict with the built-in keys.

However, there is a problem with this: Many plugins will want to do this, so how can they be sure that the key codes they make won't be interpreted by another plugin? Thus, we come to the purpose of this enum: The `range` enum gives the ranges of possible raw key values that each plugin will use; by referring to it, plugins can be sure none of them will step on each others' toes.

In summary, the values in the enum here gives the possible raw keycode values that the various plugins will inject; if you're trying to make a plugin that will be injecting special key events, you should probably add yourself to the enum here.

10.8.55 Redial

If you ever wanted to just repeat the last key pressed, no matter what it was, this plugin is made for you. It allows you to configure a key that will repeat whatever the last previously pressed key was. Of course, one can limit which keys are remembered…

Using the plugin

To use the plugin, we'll need to enable it, and configure a key to act as the "redial" key. This key should be on the keymap too.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Redial.h>

// Place Key_Redial somewhere on the keymap...

KALEIDOSCOPE_INIT_PLUGINS(Redial);

void setup() {
  Kaleidoscope.setup();
}
```

Overridable plugin methods

`bool shouldRemember(Key mapped_key)`

If one wants to change what keys the plugin remembers, simply override the `kaleidoscope::Redial::shouldRemember` function. Whenever a key is to be remembered, this function will be called with the key as argument. It should return `true` if the key should be remembered (and repeated by Redial), `false` otherwise.

By default, the plugin will remember alphanumeric keys only.

Dependencies

- Kaleidoscope-Ranges

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.56 ShapeShifter

`ShapeShifter` is a plugin that makes it considerably easier to change what symbol is input when a key is pressed together with `Shift`. If one wants to rearrange the symbols on the number row for example, without modifying the layout on the operating system side, this plugin is where one can turn to.

What it does, is very simple: if any key in its dictionary is found pressed while `Shift` is held, it will press another key instead of the one triggering the event. For example, if it sees `Shift + 1` pressed together, which normally results in a `!`, it will press `4` instead of `1`, inputting `$`.

Using the plugin

To use the plugin, one needs to include the header, create a dictionary, and configure the provided `ShapeShifter` object to use the dictionary:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-ShapeShifter.h>

static const kaleidoscope::plugin::ShapeShifter::dictionary_t shape_shift_dictionary[]␣
↪PROGMEM = {
 {Key_1, Key_4},
 {Key_4, Key_1},
 {Key_NoKey, Key_NoKey},
};

KALEIDOSCOPE_INIT_PLUGINS(ShapeShifter);

void setup() {
  Kaleidoscope.setup();

  ShapeShifter.dictionary = shape_shift_dictionary;
}
```

The dictionary is made up of `Key` pairs: the first one is to replace, the second is the replacement. The dictionary must be closed with a `{Key_NoKey, Key_NoKey}` pair, and **must** reside in `PROGMEM`.

Plugin methods

The plugin provides the ShapeShifter object, with the following methods and properties:

.dictionary

Set this property to the dictionary ShapeShifter should use. The dictionary is an array of kaleidoscope::ShapeShifter::dictionary_t elements, which is just a very verbose way of saying that it is a pair of keys. The first one is the one to replace, and the other is to replace it with.

Be aware that the replacement key will be pressed with Shift held, so do keep that in mind!

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.57 SpaceCadet

Space Cadet is a way to make it more convenient to input parens - those (and) things -, symbols that a lot of programming languages use frequently. If you are working with Lisp, you are using these all the time.

What it does, is that it turns your left and right Shift keys into parens if you tap and release them, without pressing any other key while holding them. Therefore, to input, say, (print foo), you don't need to press Shift, hold it, and press 9 to get a (, you simply press and release Shift, and continue writing. You use it as if you had a dedicated key for parens!

But if you wish to write capital letters, you hold it, as usual, and you will not see any parens when you release it. You can also hold it for a longer time, and it still would act as a Shift, without the parens inserted on release: this is useful when you want to augment some mouse action with Shift, to select text, for example.

After getting used to the Space Cadet style of typing, you may wish to enable this sort of functionality on other keys, as well. Fortunately, the Space Cadet plugin is configurable and extensible to support adding symbols to other keys. Along with (on your left Shift key and) on your right Shift key, you may wish to add other such programming mainstays as { to your left-side cmd key, } to your right-side alt key, [to your left Control key, and] to your right Control key. You can map the keys in whatever way you may wish to do, so feel free to experiment with different combinations and discover what works best for you!

Using the plugin

Using the plugin with its defaults is as simple as including the header, and enabling the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-SpaceCadet.h>

KALEIDOSCOPE_INIT_PLUGINS(SpaceCadet);

void setup() {
  Kaleidoscope.setup();
}
```

This assumes a US QWERTY layout on the host computer, though the plugin sends the correct keymap code for each symbol. Because the mapping is entirely configurable, though, you may switch out keys at your leisure.

If you wish to enable additional modifier keys (or disable the default behavior for the shift and parentheses combinations), configuration is as simple as passing a new keymap into the SpaceCadet object, as shown below:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-SpaceCadet.h>

KALEIDOSCOPE_INIT_PLUGINS(SpaceCadet);

void setup() {
  Kaleidoscope.setup();

  //Set the keymap with a 250ms timeout per-key
  //Setting is {KeyThatWasPressed, AlternativeKeyToSend, TimeoutInMS}
  //Note: must end with the SPACECADET_MAP_END delimiter
  static kaleidoscope::plugin::SpaceCadet::KeyBinding spacecadetmap[] = {
    {Key_LeftShift, Key_LeftParen, 250}
    , {Key_RightShift, Key_RightParen, 250}
    , {Key_LeftGui, Key_LeftCurlyBracket, 250}
    , {Key_RightAlt, Key_RightCurlyBracket, 250}
    , {Key_LeftAlt, Key_RightCurlyBracket, 250}
    , {Key_LeftControl, Key_LeftBracket, 250}
    , {Key_RightControl, Key_RightBracket, 250}
    , SPACECADET_MAP_END
  };
  //Set the map.
  SpaceCadet.setMap(spacecadetmap);
}
```

Plugin methods

The plugin provides two objects, `SpaceCadet` and `SpaceCadetConfig`. The latter requires the first, and allows configuring some aspects of `SpaceCadet` through Focus.

The `SpaceCadet` object provides the following methods:

`.setMap(map)`

Set the key map. This takes an array of `kaleidoscope::plugin::SpaceCadet::KeyBinding` objects with the special `SPACECADET_MAP_END` sentinel to mark the end of the map. Each KeyBinding object takes, in order, the key that was pressed, the key that should be sent instead, and an optional per-key timeout override

If not explicitly set, defaults to mapping left `shift` to `(` and right `shift` to `)`.

`kaleidoscope::plugin::SpaceCadet::KeyBinding`

An object consisting of the key that is pressed, the key that should be sent in its place, and the timeout (in milliseconds) until the key press is considered to be a "held" key press. The third parameter, the timeout, is optional and may be set per-key or left out entirely (or set to `0`) to use the default timeout value.

`.setTimeout(timeout)`

Sets the number of milliseconds to wait before considering a held key in isolation as its secondary role. That is, we'd have to hold a `Shift` key this long, by itself, to trigger the `Shift` role in itself. This timeout setting can be overridden by an individual key in the keymap, but if it is omitted or set to `0` in the key map, the global timeout will be used.

Defaults to 200.

`.getTimeout()`

Returns the number of milliseconds SpaceCadet will wait before considering a key held in isolation as its secondary role. This returns the *global* setting, as set by `.setTimeout()`. If any key in the mapping set by `.setMap()` has a different timeout, that is not considered here.

`.enable()`

This method enables the SpaceCadet plugin. This is useful for interfacing with other plugins or macros, especially where SpaceCadet functionality isn't always desired.

The default behavior is `enabled`.

`.enableWithoutDelay()`

This method enables the SpaceCadet plugin in "no-delay" mode. In this mode, SpaceCadet immediately sends the primary (modifier) value of the SpaceCadet key when it is pressed. If it is then released before timing out, it sends the alternate "tap" value, replacing the modifier.

`.disable()`

This method disables the SpaceCadet behavior. This is useful for interfacing with other plugins or macros, especially where SpaceCadet functionality isn't always desired.

`.active()`

This method returns `true` if SpaceCadet is enabled and `false` if SpaceCadet is disabled. This is useful for interfacing with other plugins or macros, especially where SpaceCadet functionality isn't always desired.

`.activeWithoutDelay()`

This method returns `true` if SpaceCadet is enabled, and is in "no-delay" mode, as set by `.enableWithoutDelay()`.

`Key_SpaceCadetEnable`

This provides a key for placing on a keymap for enabling the SpaceCadet behavior. This is only triggered on initial press, and does not trigger again if held down or when the key is released.

`Key_SpaceCadetDisable`

This provides a key for placing on a keymap for disabling the SpaceCadet behavior. This is only triggered on initial press, and does not trigger again if held down or when the key is released.

Focus commands

When using the `SpaceCadetConfig` plugin, the following Focus commands become available:

`spacecadet.mode`

Without arguments, returns the mode SpaceCadet is currently in, as a number. When `SpaceCadet` is enabled in normal mode, this returns 0. When it is turned off, it returns 1. When it is active in no-delay mode, it returns 2.

When an argument is supplied, it must be one of the above, and will set the SpaceCadet mode appropriately. Giving a numeric argument other than the allowed ones will disable SpaceCadet.

`spacecadet.timeout`

Without arguments, prints the global timeout used by SpaceCadet.

When an argument is given, it sets the global timeout.

Dependencies

- Kaleidoscope-Ranges

Optional dependencies, if using the `SpaceCadetConfig` object

- Kaleidoscope-EEPROM-Settings
- Kaleidoscope-FocusSerial

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.58 Steno

Stenography is a way to write in shorthand, a chorded input system that allows very fast input (considerably higher than normal touch typing), by using shorthand chords and a dictionary. This plugin implements the `GeminiPR` protocol that supports a number of systems, including Plover.

While Plover supports a normal QWERTY keyboard too, having a dedicated plugin comes with important advantages:

- No need to toggle Plover on and off, because the normal keys are not taken over by Plover anymore.
- Easier toggling, because you only have to toggle the layer, not Plover too. If you switch back to a keyboard layer, without toggling Plover off, nothing unexpected will happen. Plover will not take over the keys.
- The `GeminiPR` protocol supports language systems other than English.

Do note that the `GeminiPR` protocol is implemented over the virtual serial port, so any plugin that wants to use that port too, will run into conflicts with the Steno plugin. In other words, don't use it together with Focus.

What is Steno? Why should I use it? How do I learn?

As mentioned above, steno (short for "stenography") is a shorthand, chorded input system that allows very fast input - licensed stenographers are required to type **225 WPM at 95% accuracy** to get their license. Although reaching that speed typically takes 2-6 years of practice and training, lower speeds comparable to or exceeding that of touch typing can reportedly be reached in only a few months.

This talk (YouTube link) gives a brief introduction to Steno, how it works, and why it is cool.

One recommend way to get started with learning Steno is with Plover. Plover is software for your computer that will interpret the steno input from your Model 01 (or other NKRO QWERTY keyboard); it is available for Windows, macOS, and Linux. Plover's Beginner's Guide is a great place to get started with Steno in general and Plover in particular.

Using the plugin

To use the plugin, simply include the header in your Sketch, tell the firmware to use the `GeminiPR` object, and place Steno keys on your keymap. It is best illustrated with an example:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Steno.h>

// Somewhere in the keymap:
S(S1), S(S2), etc

KALEIDOSCOPE_INIT_PLUGINS(GeminiPR);

void setup() {
  Kaleidoscope.setup();
}
```

Keys provided by the plugin

The plugin provides a number of keys one can put on the keymap, that allow correspond to various Steno keys. All of these must be used together with the `S()` macro provided by the plugin, as can be seen in the example above.

The provided keys are: `FN`, `N1`, `N2`, `N3`, `N4`, `N5`, `N6`, `S1`, `S2`, `TL`, `KL`, `PL`, `WL`, `HL`, `RL`, `A`, `O`, `ST1`, `ST2`, `RE1`, `RE2`, `PWR`, `ST3`, `ST4`, `E`, `U`, `FR`, `RR`, `PR`, `BR`, `LR`, `GR`, `TR`, `SR`, `DR`, `N7`, `N8`, `N9`, `NA`, `NB`, `NC`, `ZR`.

See the *example* for the default/suggested placements of each of these keys.

Plugin methods and properties

The plugin provides a `GeminiPR` object, with no public methods or properties.

Dependencies

- Kaleidoscope-Ranges

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.59 Syster

Syster is a way to input symbols in a different way: instead of macros, Leader sequences or the like, we trigger the special input mode, and enter the symbol's name. Once finished, we hit `Space`, and this plugin will do the rest: delete everything we typed, look up an action for the entered symbol, and execute that.

There are a number of ways this can be useful, but the easiest showcase is symbolic Unicode input: `SYSTER coffee SPACE` turns into , with just a little code.

Using the plugin

To use the plugin, one needs to include the header and set up a function that will handle the symbol actions:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-HostOS.h>
#include <Kaleidoscope-Syster.h>
#include <Kaleidoscope-Unicode.h>

void systerAction(kaleidoscope::plugin::Syster::action_t action, const char *symbol) {
  switch (action) {
  case kaleidoscope::plugin::Syster::StartAction:
    Unicode.type (0x2328);
    break;
  case kaleidoscope::plugin::Syster::EndAction:
    handleKeyswitchEvent (Key_Backspace, UnknownKeyswitchLocation, IS_PRESSED |␣
↪INJECTED);
    Kaleidoscope.hid().keyboard().sendReport();
    handleKeyswitchEvent (Key_Backspace, UnknownKeyswitchLocation, WAS_PRESSED |␣
```

(continues on next page)

(continued from previous page)

```
↪INJECTED);
    Kaleidoscope.hid().keyboard().sendReport();
    break;
  case kaleidoscope::plugin::Syster::SymbolAction:
    Kaleidoscope.serialPort().print ("systerAction: symbol=");
    Kaleidoscope.serialPort().println (symbol);
    if (strcmp (symbol, "coffee") == 0) {
      Unicode.type (0x2615);
    }
    break;
  }
}

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings, HostOS, Unicode, Syster);

void setup() {
  Kaleidoscope.serialPort().begin(9600);
  Kaleidoscope.setup ();
}
```

Note that we need to use the `Syster` object before any other that adds or changes key behaviour! Failing to do so may result in unpredictable behaviour.

Plugin methods

The plugin provides the `Syster` object, with no public methods. There are two methods outside of the object, however, that can be overridden:

`systerAction(action, symbol)`

Called whenever an action needs to be taken, which can happen in three cases:

First, when the `Syster` key is pressed and the alternate processing starts. In this case, `action` will be set to `kaleidoscope::plugin::Syster::StartAction`, and `symbol` will be `NULL`. This function can be used to do some setup to make it more obvious that the Syster input mode is active, such as sending a Unicode symbol to the host, or lighting up LEDs, or anything else we'd like.

Second, when the sequence is finished with a `Space`. In this case, `action` will be set to `kaleidoscope::plugin::Syster::EndAction` and `symbol` will be `NULL`. This can be used to undo anything that the start action did, if need be.

Third, when the action for the symbol should be made. In this case, `action` is set to `kaleidoscope::plugin::Syster::SymbolAction`, and `symbol` will be a C string. It is up to us, what we do with this information, how we handle it.

`keyToChar(key)`

A function that turns a keycode into a character. If using QWERTY on the host, the default implementation is sufficient. When using something else, you may have to reimplement this function.

Dependencies

- Kaleidoscope-Ranges

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.60 TapDance

Tap-dance keys are general purpose, multi-use keys, which trigger a different action based on the number of times they were tapped in sequence. As an example to make this clearer, one can have a key that inputs `A` when tapped once, inputs `B` when tapped twice, and lights up the keyboard in Christmas colors when tapped a third time.

This behaviour is most useful in cases where we have a number of things we perform rarely, where tapping a single key repeatedly is not counter-productive. Such cases include - for example - multimedia forward / backward keys: forward on single tap, backward on double. Of course, one could use modifiers to achieve a similar effect, but that's two keys to use, this is only one. We can also hide some destructive functionality behind a number of taps: reset the keyboard after 4 taps, and light up LEDs in increasingly frightful colors until then.

How does it work?

To not interfere with normal typing, tap-dance keys have two ways to decide when to call an action: they either get interrupted, or they time out. Every time a tap-dance key is pressed, the timer resets, so one does not have to finish the whole tapping sequence within a short time limit. The tap-dance counter continues incrementing until one of these cases happen.

When a tap-dance key is pressed and released, and nothing is pressed on the keyboard until the timeout is reached, then the key will time out, and trigger an action. Which action, depends on the number of times it has been tapped up until this point.

When a tap-dance key is pressed and released, and another key is hit before the timer expires, then the tap-dance key will trigger an action first, perform it, and only then will the firmware continue handling the interrupting key press. This is to preserve the order of keys pressed.

In both of these cases, the user-defined `tapDanceAction()` function will be called, with `tap_dance_index` set to the index of the tap-dance action (as set in the keymap), the `tap_count`, and `tap_dance_action` set to one of the following values:

- `kaleidoscope::plugin::TapDance::Hold`, if the tap-dance key is still being held when its timeout expires.
- `kaleidoscope::plugin::TapDance::Timeout`, if the tap-dance key has been released when its timeout expires.
- `kaleidoscope::plugin::TapDance::Interrupt`, if another key is pressed before the tap-dance key's timeout expires.

These actions allow us to create sophisticated tap-dance setups, where one can tap a key twice and hold it, and have it repeat, for example.

There is one additional value the `tap_dance_action` parameter can take: `kaleidoscope::plugin::TapDance::Tap`. It is called with this argument for each and every tap, even if no action is to be triggered yet. This is so that we can have a way to do some side-effects, like light up LEDs to show progress, and so on.

Using the plugin

To use the plugin, we need to include the header, and declare the behaviour used. Then, we need to place tap-dance keys on the keymap. And finally, we need to implement the *`tapDanceAction`* function that gets called each time an action is to be performed.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-TapDance.h>

// Somewhere in the keymap:
TD(0)

// later in the Sketch:
void tapDanceAction(uint8_t tap_dance_index, KeyAddr key_addr, uint8_t tap_count,
                    kaleidoscope::plugin::TapDance::ActionType tap_dance_action) {
  switch (tap_dance_index) {
  case 0:
    return tapDanceActionKeys(tap_count, tap_dance_action,
                              Consumer_ScanNextTrack, Consumer_ScanPreviousTrack);
  }
}

KALEIDOSCOPE_INIT_PLUGINS(TapDance);

void setup() {
  Kaleidoscope.setup ();
}
```

Keymap markup

`TD(id)`

A key that acts as a tap-dance key. The actions performed depend on the implementation for the `id` index within the [`tapDanceActions`][tdactions] function.

The `id` parameter here is what will be used as `tap_dance_index` in the handler function.

Plugin methods

The plugin provides a `TapDance` object, but to implement the actions, we need to define a function (*tapDanceAction*) outside of the object. A handler, of sorts. Nevertheless, the plugin provides one macro that is particularly useful: `tapDanceActionKeys`. Apart from that, it provides only one configuration method:

`.setTimeout(timeout)`

Set the number of milliseconds to wait before a tap-dance sequence times out. Once the sequence timed out, the action for it will trigger, even without an interruptor. Defaults to 5, and the timer resets with every tap of the same

`tapDanceActionKeys(tap_count, tap_dance_action, keys...)`

Sets up an action where for each subsequent tap, a different key will be chosen from the list of keys supplied in the `keys...` argument.

If we have `Key_A` and `Key_B` in the list, then, if tapped once, this function will input `A`, but when tapped twice, will input `B`.

When all our actions are just different keys, this is a very handy macro to use.

The `tap_count` and `tap_dance_actions` parameters should be the same as the similarly named parameters of the `tapDanceAction` function.

`tapDanceAction(tap_dance_index, key_addr, tap_count, tap_dance_action)`

The heart of the tap-dance plugin is the handler method. This is called every time any kind of tap-dance action is to be performed. See the How does it work? section for details about when and how this function is called.

The `tap_dance_index` and `tap_count` parameters help us choose which action to perform. The `key_addr` parameter tells us where the tap-dance key is on the keyboard.

Dependencies

- Kaleidoscope-Ranges

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.61 TopsyTurvy

TopsyTurvy is a plugin that inverts the behaviour of the Shift key for some selected keys. That is, if configured so, it will input ! when pressing the 1 key without Shift, but with the modifier pressed, it will input the original 1 symbol.

Using the plugin

To use the plugin, one needs to include the header, mark keys to apply plugin effects to, and use the plugin:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-TopsyTurvy.h>

// In the keymap:
TOPSY(1), TOPSY(2), TOPSY(3)

KALEIDOSCOPE_INIT_PLUGINS(TopsyTurvy);

void setup () {
  Kaleidoscope.setup ();
}
```

Keymap markup

There is only one macro that the plugin provides, which one can use in keymap definitions:

TOPSY(key)

Mark the specified key (without the Key_ prefix!) for TopsyTurvy, and swap the effect of Shift when the key is used. One can have any number of topsy-turvy keys on a keymap.

The keys must be plain old keys, modifiers or anything other augmentation cannot be applied.

The plugin provides a number of macros one can use in keymap definitions:

Plugin methods

The plugin provides the TopsyTurvy object, without any public methods or properties.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.62 Turbo

The Turbo plugin provides an extra key one can place on their keymap. While the key is pressed or toggled, pressing other keys will generate quick repeated inputs independent of the OS key repetition mechanics.

Using the plugin

To use the plugin, simply include the header and enable the plugin and place Key_Turbo somewhere on your keymap. You may add additionally configure specific behaviors of the plugin as shown:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Turbo.h>
#include <Kaleidoscope-LEDControl.h>

// somewhere in the keymap...
Key_Turbo

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, Turbo);

void setup() {
    Kaleidoscope.setup();

    Turbo.interval(30);
    Turbo.sticky(true);
    Turbo.flash(true);
    Turbo.flashInterval(80);
    Turbo.activeColor(CRGB(0x64, 0x96, 0xed));
}
```

Plugin properties

The Turbo object has the following user-configurable properties:

`.interval([uint16_t])`

This property adjusts the timing between simulated keypresses. If you set this too low, some programs might not like it. The default repeat rate for X11 is 25.

Defaults to 10

`.flashInterval([uint16_t])`

This property adjusts the timing between the on/off states of the key LED.

Defaults to 69

`.sticky([bool])`

This method makes the Turbo functionality sticky, so it remains in effect not only while it is held, but after it is released too, until it is toggled off with another tap. Without arguments, the method enables the sticky functionality. Passing a boolean argument sets stickiness to the given value.

Defaults to `false`.

`.flash([bool])`

This property indicates whether the key should flash when enabled or remain a solid color.

Defaults to true.

`.activeColor([cRGB])`

This property indicates the color the key should become when enabled.

Defaults to `CRGB(160, 0, 0)` (same as solidRed in default firmware).

Dependencies

- Kaleidoscope-LEDControl

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.63 TypingBreaks

Typing on the keyboard for an extended period of time may lead to injuries, which is why it is highly recommended to take frequent breaks from the keyboard - and from the computer as well. But sometimes - more often than one would wish to admit - we tend to forget about this, and plow through, at the cost of hand's health.

No more.

With the `TypingBreaks` plugin, we can instruct the keyboard to lock itself up after some time, or after a number of key presses. It will stay locked for a few minutes (or whatever amount we told it to), forcing us to take a break.

Using the plugin

The plugin comes with reasonable defaults (see below), and can be used out of the box, without any further configuration:

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-TypingBreaks.h>

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings, TypingBreaks);

void setup () {
```

(continues on next page)

(continued from previous page)

```
  Kaleidoscope.setup ();

  TypingBreaks.settings.idle_time_limit = 60;
}
```

Plugin methods

The plugin provides a single object, `TypingBreaks`, with the following properties. All times are in seconds.

`.settings.idle_time_limit`

The amount of time that can pass between two pressed keys, before the plugin considers it a new session, and starts all timers and counters over.

Defaults to 300 seconds (5 minutes).

`.settings.lock_time_out`

The length of the session, after which the keyboard will be locked.

Defaults to 2700 seconds (45 minutes).

`.settings.lock_length`

The length until the keyboard lock is held. Any key pressed while the lock is active, will be discarded.

Defaults to 300 seconds (5 minutes).

`.settings.left_hand_max_keys`

It is possible to lock the keyboard after a number of keys pressed, too. If this happens sooner than the timeout, the keyboard will still be locked.

This property controls how many keys can be pressed on the left side.

Defaults to 0 (off).

`.settings.right_hand_max_keys`

It is possible to lock the keyboard after a number of keys pressed, too. If this happens sooner than the timeout, the keyboard will still be locked.

This property controls how many keys can be pressed on the right side.

Defaults to 0 (off).

Focus commands

`typingbreaks.idleTimeLimit [limit]`

Get or set the `.settings.idle_time_limit` property.

`typingbreaks.lockTimeOut [time_out]`

Get or set the `.settings.lock_time_out` property.

`typingbreaks.lockLength [length]`

Get or set the `.settings.lock_length` property.

`typingbreaks.leftMaxKeys [max]`

Get or set the `.settings.left_hand_max_keys` property.

`typingbreaks.rightMaxKeys [max]`

Get or set the `.settings.right_hand_max_keys` property.

`typingbreaks.leftKeys`

Get the current counter of keys pressed on the left half of the keyboard.

`typingbreaks.rightKeys`

Get the current counter of keys pressed on the right half of the keyboard.

`typingbreaks.lockSecsRemaining`

Get the duration the keyboard remains locked in seconds.

Dependencies

- Kaleidoscope-EEPROM-Settings

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.64 USB-Quirks

USB-Quirks provides a few methods to deal with more obscure parts of the USB spec, such as changing the behavior around the boot protocol. These are in a separate plugin, because these features are not part of the USB spec, and are often workarounds for various issues. See the provided methods for more information about what they're useful for.

Using the plugin

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-Macros.h>
#include <Kaleidoscope-USB-Quirks.h>

KALEIDOSCOPE_INIT_PLUGINS(USBQuirks, Macros);

const macro_t *macroAction(uint8_t macroIndex, uint8_t keyState) {
  if (macroIndex == 0) {
    USBQuirks.toggleKeyboardProtocol();
  }
  return MACRO_NONE;
}

void setup() {
  Kaleidoscope.setup();
}
```

Plugin methods

The plugin provides one object, `USBQuirks`, which provides the following method:

`.toggleKeyboardProtocol()`

Toggle whether the keyboard is able to send extended key reports (the default), or instead always sends boot reports, regardless of the protocol requested by the host. Switching the toggle causes the keyboard to detach and then re-attach to the host. (This re-attach is necessary to force re-enumeration with a different Report Descriptor.)

Switching the toggle also lights up a key indicating the mode being switched to: by default, `B` for boot reports only, and `N` for extended reports enabled.

The extended key report supports n-key rollover (NKRO), and is actually a hybrid, having a prefix containing the boot report, for compatibility with older hosts. The boot report only supports 6-key rollover (6KRO), and is meant to support constrained hosts, such as BIOS, UEFI, or other pre-boot environments. The keyboard changes protocols as requested by the host.

The USB HID specification requires that hosts explicitly request boot protocol if they need it, and that devices default to the non-boot protocol. Some hosts do not follow the specification, and expect boot protocol, even without requesting it. The backwards compatibility prefix of the hybrid extended report should accommodate some of these hosts. This toggle helps to work with hosts that neither request boot protocol nor tolerate the longer hybrid report.

`.setKeys(Key boot_led, Key nkro_led)`

Set which keys to light up to indicate the target mode. Defaults to `(Key_B, Key_N)`.

10.8.65 Unicode

The `Unicode` extension makes it easier to write plugins that input Unicode symbols on the host. Because inputting Unicode varies from OS to OS, this helper library was made to hide most of the differences. All one has to do, is set up the `HostOS` singleton properly, and the `Unicode` library will handle the rest, by providing an easy interface for inputting Unicode symbols by their 32-bit codepoints.

Using the extension

Using the extension is as simple as including the header, registering it with `Kaleidoscope.use()`, and then using any of the methods provided by the `Unicode` singleton object.

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-HostOS.h>
#include <Kaleidoscope-Unicode.h>

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings, HostOS, Unicode);

void setup() {
  Kaleidoscope.setup();

  Unicode.type(0x2328);
}
```

Extension methods

The extension provides a number of methods on the `Unicode` object, but also has symbols that can be *overridden*, to add custom functionality.

`.type(code_point)`

Starts the Unicode input method using the *`.start()`* method, inputs the `code_point` using *`.typeCode()`*, and finishes up with the *`.end()`* method. For each hexadecimal digit sent to the host, the *`.input()`* method will also be called.

This method is most useful when one knows the code point of the Unicode symbol to enter ahead of time, when the code point does not depend on anything else.

`.typeCode(code_point)`

Inputs the hex codes for `code_point`, and the hex codes only. Use when the input method is to be started and ended separately.

For example, a macro that starts Unicode input, and switches to a layer full of macros that send the hex codes is one scenario where this function is of use.

`.start()`

Starts the Unicode input method. The way it starts it, depends on the host operating system.

`.input()`

If the host operating system requires keys being held during the Unicode input, this function will hold them for us.

`.end()`

Finishes the Unicode input method, in an OS-specific way.

`.input_delay([delay])`

Sets or returns (if called without an argument) the number of milliseconds to wait between inputting each part of the sequence. In some cases, inputting too fast does not give the host enough time to process, and a delay is needed.

Defaults to zero, no delay.

Overridable methods

`hexToKey(hex_digit)`

A function that returns a `Key` struct, given a 8-bit hex digit. For most uses, the built-in version of this function is sufficient, but if the keymap on the OS-side has any of the hexadecimal symbols on other scancodes than QWERTY, this function should be overridden to use the correct scan codes.

`unicodeCustomStart()`

If the host OS type is set to `kaleidoscope::hostos::Custom`, then this function will be called whenever the *`.start()`* method is called. The default implementation does nothing, and should be overridden to implement the custom magic needed to enter unicode input mode.

`unicodeCustomInput()`

If the host OS type is set to `kaleidoscope::hostos::Custom`, then this function will be called whenever the *`.input()`* method is called. The default implementation does nothing, and should be overridden to implement the custom magic needed while inputting the hex code itself (such as holding additional keys).

`unicodeCustomEnd()`

If the host OS type is set to `kaleidoscope::hostos::Custom`, then this function will be called whenever the *`.end()`* method is called. The default implementation does nothing, and should be overridden to implement the custom magic needed to leave unicode input mode.

Dependencies

- Kaleidoscope-HostOS

Other Configuration

On OS X/macOS, you'll need to change the input method to be "Unicode Hex Input". You can do this by going to System Preferences > Keyboard > Input Sources, clicking the + button, selecting it from the list, then setting it as the active input method.

On Windows, you will need to change a registry key to enable the input method our unicode plugin uses. Under `HKEY_Current_User/Control Panel/Input Method`, set `EnableHexNumpad` to `"1"`. If the key does not exist, you need to create it, and use `REG_SZ` as the type.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.8.66 WinKeyToggle

If you ever played games on Windows on a traditional keyboard, you likely ran into the issue of the Windows key: in the heat of the moment, you accidentally hit the windows key, and find yourself out of the game on the desktop, with the Start menu open. Annoying, is it? So you'd like to *temporarily* disable the key while gaming, and this plugin will help you achieve that.

This plugin provides a method to toggle the windows keys on and off. Bind it to a macro, or a magic combo, and you have an easy way to toggle the key on and off.

Using the extension

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-MagicCombo.h>
#include <Kaleidoscope-WinKeyToggle.h>

enum { WINKEY_TOGGLE };

void toggleWinKey(uint8_t combo_index) {
```

(continues on next page)

(continued from previous page)

```
  WinKeyToggle.toggle();
}

USE_MAGIC_COMBOS(
[WINKEY_TOGGLE] = {
  .action = toggleWinKey,
  .keys = {R3C6, R3C9} // Left Fn + Right Fn
});

KALEIDOSCOPE_INIT_PLUGINS(MagicCombo, WinKeyToggle);

void setup() {
  Kaleidoscope.setup();
}
```

Plugin properties

The extension provides a `WinKeyToggle` singleton object, with the following method:

`.toggle`

Toggles the Windows keys on and off.

Defaults to off.

Further reading

Starting from the *example* is the recommended way of getting started with the plugin.

10.9 Keyboardio Atreus

This is a plugin for Kaleidoscope, that provides hardware support for the Keyboardio Atreus.

The default firmware sketch for the Atreus is included with Kaleidoscope

10.10 Keyboardio Model 100

This is a plugin for Kaleidoscope, that adds hardware support for the Keyboardio Model 100.

The default firmware sketch for the Model 100 is available on GitHub

10.11 Keyboardio Model 01

This is a plugin for Kaleidoscope, that adds hardware support for the Keyboardio Model 01.

The default firmware sketch for the Model 01 is available on GitHub

10.12 ErgoDox EZ

This is a plugin for Kaleidoscope, that adds hardware support for the ErgoDox. To be able to flash the firmware, one will need the Teensy Loader CLI tool in addition to Arduino.

10.13 OLKB Planck

This is a plugin for Kaleidoscope, that adds hardware support for the OLKB Planck. For now, this only supports variants with an ATMega32U4 MCU. Speakers and LEDs are not supported.

10.14 SOFTHRUF Splitography

Hardware support for the SOFTHRUF Splitography Steno keyboard.

10.15 Technomancy Atreus

This is a plugin for Kaleidoscope, that adds hardware support for the Atreus. Supports both the pre- and post-2016 variants with an A* MCU, the hand-wired variant from FalbaTech with a Teensy2, and the legacy teensy2 variant too.

To select which one to build for, you can either use the Arduino IDE, and select the appropriate Pinout and CPU, or, if using `kaleidoscope-builder`, you can add a `LOCAL_CFLAGS` setting to your Makefile

For the post-2016 variant (the default, also used by the handwired variant from FalbaTech), if you want to explicitly select it, add `-DKALEIDOSCOPE_HARDWARE_ATREUS_PINOUT_ASTAR=1`. For the pre-2016 variant, add `-DKALEIDOSCOPE_HARDWARE_ATREUS_PINOUT_ASTAR_DOWN`. For the legacy teensy2 variant, add `-DKALEIDOSCOPE_HARDWARE_ATREUS_PINOUT_LEGACY_TEENSY2=1`.

To be able to flash the firmware, one will need the Teensy Loader CLI tool in addition to Arduino, if using a Teensy. If using an A* MCU, the additional tool is not required.

CHAPTER

ELEVEN

FOR DEVELOPERS

11.1 Kaleidoscope C++ Coding Style

11.1.1 Important warning

This document is currently a work in progress. While you certainly won't be penalized for following the style described herein, it's still a moving target as of June 20, 2017.

Our style guide is based on the Google C++ style guide which was current as of June 2, 2017, but has been modified to better reflect the constraints of embedded development and the peculiarities of an Arduino-compatible environment.

11.1.2 Table of Contents

11.1.3 Background

C++ is the language of Arduino libraries, and as a consequence, the language in which Kaleidoscope was written. As every C++ programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and to maintain.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively.

Style, also known as readability, is what we call the conventions that govern our C++ code. The term Style is a bit of a misnomer, since these conventions cover far more than just source file formatting.

Note that this guide is not a C++ tutorial: we assume that the reader is familiar with the language.

Goals of the Style Guide

Why do we have this document?

There are a few core goals that we believe this guide should serve. These are the fundamental **why**s that underlie all of the individual rules. By bringing these ideas to the fore, we hope to ground discussions and make it clearer to our broader community why the rules are in place and why particular decisions have been made. If you understand what goals each rule is serving, it should be clearer to everyone when a rule may be waived (some can be), and what sort of argument or alternative would be necessary to change a rule in the guide.

The goals of the style guide as we currently see them are as follows:

The intent of this document is to provide maximal guidance with reasonable restriction. As always, common sense and good taste should prevail. By this we specifically refer to the established conventions of the entire community (both Kaleidoscope and Arduino communities), not just personal preferences. Be skeptical about and reluctant to use clever or unusual constructs: the absence of a prohibition is not the same as a license to proceed. Use your judgment, and if you are unsure, please don't hesitate to ask (e.g. on Discourse or on Discord), to get additional input.

11.1.4 Library Guidelines

Before looking at the coding style guidelines, we must first talk about libraries. Every Kaleidoscope plugin is also an Arduino library. The core firmware is an Arduino library too. As such, libraries should follow the Arduino library specification (revision 2.1 or later), with a few additional recommendations:

11.1.5 Header Files

In general, every `.cpp` file should have an associated `.h` file. There are some common exceptions, such as unittests and examples.

Correct use of header files can make a huge difference to the readability, size and performance of your code.

The following rules will guide you through the various pitfalls of using header files.

Self-contained Headers

> Header files should be self-contained (compile on their own) and end in `.h`. Non-header files that are meant for inclusion should end in `.inc` and be used sparingly.

All header files should be self-contained. Users and refactoring tools should not have to adhere to special conditions to include the header. Specifically, a header should have *header guards* and include all other headers it needs.

Prefer placing the definitions for template and inline functions in the same file as their declarations. The definitions of these constructs must be included into every `.cpp` file that uses them, or the program may fail to link in some build configurations. If declarations and definitions are in different files, including the former should transitively include the latter. Do not move these definitions to separately included header files (e.g. `-inl.h` files).

As an exception, a template that is explicitly instantiated for all relevant sets of template arguments, or that is a private implementation detail of a class, is allowed to be defined in the one and only `.cpp` file that instantiates the template.

There are rare cases where a file designed to be included is not self-contained. These are typically intended to be included at unusual locations, such as the middle of another file. They might not use *header guards*, and might not include their prerequisites. Name such files with the `.inc` extension. Use sparingly, and prefer self-contained headers when possible.

Header Guards

All header files should have `#pragma once` guards at the top to prevent multiple inclusion.

Include What You Use

If a source or header file refers to a symbol defined elsewhere, the file should directly include a header file which provides a declaration or definition of that symbol.

Do not rely on transitive inclusions. This allows maintainers to remove no-longer-needed `#include` statements from their headers without breaking clients code. This also applies to directly associated headers - `foo.cpp` should include `bar.h` if it uses a symbol defined there, even if `foo.h` (currently) includes `bar.h`.

Forward Declarations

> Avoid using forward declarations where possible. Just `#include` the headers you need.

Definition

A *"forward declaration"* is a declaration of a class, function, or template without an associated definition.

Pros

- Forward declarations can save compile time, as `#include`s force the compiler to open more files and process more input.
- Forward declarations can save on unnecessary recompilation. `#include`s can force your code to be recompiled more often, due to unrelated changes in the header.

Cons

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change.
- A forward declaration may be broken by subsequent changes to the library. Forward declarations of functions and templates can prevent the header owners from making otherwise-compatible changes to their APIs, such as widening a parameter type, adding a template parameter with a default value, or migrating to a new namespace.

- It can be difficult to determine whether a forward declaration or a full `#include` is needed. Replacing an `#include` with a forward declaration can silently change the meaning of code.
- Forward declaring multiple symbols from a header can be more verbose than simply `#include`ing the header.
- Structuring code to enable forward declarations (e.g. using pointer members instead of object members) can make the code slower and more complex.

Decision

- Try to avoid forward declarations of entities defined in another project.
- When using a function declared in a header file, always `#include` that header.
- When using a class template, prefer to `#include` its header file.

Please see *Names and Order of Includes* for rules about when to `#include` a header.

Inline Functions

Define inline functions when required to do so in order to get the compiler to generate more compact code.

Definition

You can declare functions in a way that allows the compiler to expand them inline rather than calling them through the usual function call mechanism. Sometimes the compiler does this automatically, sometimes we need to instruct it explicitly for the sake of either performance, or code size.

Pros

Inlining a function can generate more efficient object code, as long as the inlined function is small. Feel free to inline accessors and mutators, and other short, performance-critical functions.

Cons

Overuse of inlining can actually make programs slower. Depending on a function's size, inlining it can cause the code size to increase or decrease. Inlining a very small accessor function will usually decrease code size while inlining a very large function can dramatically increase code size.

Decision

A decent rule of thumb is to not inline a function if it is more than 10 lines long. Beware of destructors, which are often longer than they appear because of implicit member- and base-destructor calls!

Another useful rule of thumb: it's typically not cost effective to inline functions with loops or switch statements (unless, in the common case, the loop or switch statement is never executed).

It is important to know that functions are not always inlined even if they are declared as such; for example, virtual and recursive functions are not normally inlined. Usually recursive functions should not be inline. The main reason for making a virtual function inline is to place its definition in the class, either for convenience or to document its behavior, e.g., for accessors and mutators.

Organization of Includes

Use standard order for readability and to avoid hidden dependencies:

- The header associated with this source file, if any
- System headers and Arduino library headers (including other Kaleidoscope plugins, but not Kaleidoscope itself)
- Kaleidoscope headers and headers for the individual plugin (other than the associated header above)

These three sections should be separated by single blank lines, and should be sorted alphabetically.

When including system headers and Arduino library headers (including Kaleidoscope plugins), use angle brackets to indicate that those sources are external.

For headers inside the current library and for Kaleidoscope core headers, use double quotes and a full pathname (starting below the `src/` directory). This applies to the source file's associated header, as well; don't use a pathname relative to the source file's directory.

For the sake of clarity, the above sections can be further divided to make it clear where each included header file can be found, but this is probably not necessary in most cases, because the path name of a header usually indicates which library it is located in.

For example, the includes in `Kaleidoscope-Something/src/kaleidoscope/Something.cpp` might look like this:

```
#include "kaleidoscope/Something.h"

#include <Arduino.h>
#include <Kaleidoscope-Ranges.h>
#include <stdint.h>

#include "kaleidoscope/KeyAddr.h"
#include "kaleidoscope/KeyEvent.h"
#include "kaleidoscope/key_defs.h"
#include "kaleidoscope/plugin/something/utils.h"
```

Exception

Sometimes, system-specific code needs conditional includes. Such code can put conditional includes after other includes. Of course, keep your system-specific code small and localized. Example:

```
#if defined(ARDUINO_AVR_MODEL01)
#include "kaleidoscope/Something-AVR-Model01.h"
#endif

#if defined(ARDUINO_AVR_SHORTCUT)
#include "kaleidoscope/Something-AVR-Shortcut.h"
#endif
```

Top-level Arduinio Library Headers

All libraries must have at least one header in their top-level `src/` directory, to be included without any path components. This is the way Arduino finds libraries, and a limitation we must adhere to. These headers should - in general - include any other headers they may need, so that the consumer of the library only has to include one header. The name of this header must be the same as the name of the library.

The naming convention for Kaleidoscope plugins is to use the `Kaleidoscope-` prefix: e.g. `Kaleidoscope-Something`, which would have a top-level header named `Kaleidoscope-Something.h` in its `src/` directory.

In the case of Kaleidoscope plugin libraries, the number of source and header files tends to be very small (usually just one `*.cpp` file and its associated header, in addition to the library's top-level header). When one plugin depends on another, we therefore only include the top-level header of the dependency. For example, if `Kaleidoscope-OtherThing` depends on `Kaleidoscope-Something`, the file `kaleidoscope/plugin/OtherThing.h` will contain the line:

```
#include <Kaleidoscope-Something.h>
```

. . . and `Kaleidoscope-Something.h` will look like this:

```
#include "kaleidoscope/plugin/Something.h"
```

This both makes it clearer where to find the included code, and allows the restructuring of that code without breaking the dependent library (assuming the symbols haven't changed as well).

If a plugin library has symbols meant to be exported, and more than one header file in which those symbols are defined, all such header files should be included in the top-level header for the library. For example, if `Kaleidoscope-Something` defines types `kaleidoscope::plugin::Something` and `kaleidoscope::plugin::something::Helper`, both of which are meant to be accessible by `Kaleidoscope-OtherThing`, the top-level header `Kaleidoscope-Something.h` should look like this:

```
#include "kaleidoscope/plugin/Something.h"
#include "kaleidoscope/plugin/something/Helper.h"
```

Automated header includes checking

We have an automated wrapper for the `include-what-you-use` program from LLVM that processes most Kaleidoscope source files and updates their header includes to comply with the style guide. It requires at least version 0.18 of `include-what-you-use` in order to function properly (because earlier versions do not return a useful exit code, so determining if there was an error was difficult). It can be run by using the `make check-includes` target in the Kaleidoscope Makefile.

11.1.6 Scoping

Namespaces

> With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Do not use *using-directives* (e.g. `using namespace foo`). Do not use inline namespaces. For unnamed namespaces, see *Unnamed Namespaces and Static Variables*.

Definition

Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.

Pros

Namespaces provide a method for preventing name conflicts in large programs while allowing most code to use reasonably short names.

For example, if two different projects have a class Foo in the global scope, these symbols may collide at compile time or at runtime. If each project places their code in a namespace, project1::Foo and project2::Foo are now distinct symbols that do not collide, and code within each project's namespace can continue to refer to Foo without the prefix.

Inline namespaces automatically place their names in the enclosing scope. Consider the following snippet, for example:

```
namespace X {
inline namespace Y {
  void foo();
}  // namespace Y
}  // namespace X
```

The expressions X::Y::foo() and X::foo() are interchangeable. Inline namespaces are primarily intended for ABI compatibility across versions.

Cons

Namespaces can be confusing, because they complicate the mechanics of figuring out to what definition a name refers.

Inline namespaces, in particular, can be confusing because names aren't actually restricted to the namespace where they are declared. They are only useful as part of some larger versioning policy.

In some contexts, it's necessary to repeatedly refer to symbols by their fully-qualified names. For deeply-nested namespaces, this can add a lot of clutter.

Decision

Namespaces should be used as follows:

- Follow the rules on *Namespace Names*.
- Terminate namespaces with comments as shown in the given examples.
- Namespaces wrap the entire source file after includes, gflags definitions/declarations and forward declarations of classes from other namespaces.

```
// In the .h file
namespace mynamespace {

// All declarations are within the namespace scope.
// Notice the lack of indentation.
class MyClass {
 public:
  ...
  void Foo();
};

}  // namespace mynamespace
```

```
// In the .cc file
namespace mynamespace {

// Definition of functions is within scope of the namespace.
void MyClass::Foo() {
```

(continues on next page)

(continued from previous page)

```
  ...
}

}  // namespace mynamespace
```

More complex `.cpp` files might have additional details, like flags or using-declarations.

```
#include "a.h"

DEFINE_FLAG(bool, someflag, false, "dummy flag");

namespace a {

using ::foo::bar;

...code for a...         // Code goes against the left margin.

}  // namespace a
```

- Do not declare anything in namespace `std`, including forward declarations of standard library classes. Declaring entities in namespace `std` is undefined behavior, i.e., not portable. To declare entities from the standard library, include the appropriate header file.
- You may not use a `using`-directive to make all names from a namespace available.

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

- Do not use *Namespace aliases* at namespace scope in header files except in explicitly marked internal-only namespaces, because anything imported into a namespace in a header file becomes part of the public API exported by that file.

```
// Shorten access to some commonly used names in .cc files.
namespace baz = ::foo::bar::baz;
```

```
// Shorten access to some commonly used names (in a .h file).
namespace librarian {
namespace impl {  // Internal, not part of the API.
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace impl

inline void my_inline_function() {
  // namespace alias local to a function (or method).
  namespace baz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

- Do not use inline namespaces.

Unnamed Namespaces and Static Variables

> When definitions in a `.cpp` file do not need to be referenced outside that file, place them in an unnamed namespace or declare them `static`. Do not use either of these constructs in `.h` files.

Definition

All declarations can be given internal linkage by placing them in unnamed namespaces, and functions and variables can be given internal linkage by declaring them `static`. This means that anything you're declaring can't be accessed from another file. If a different file declares something with the same name, then the two entities are completely independent.

Decision

Use of internal linkage in `.cpp` files is encouraged for all code that does not need to be referenced elsewhere. Do not use internal linkage in `.h` files.

Format unnamed namespaces like named namespaces. In the terminating comment, leave the namespace name empty:

```
namespace {
...
}  // namespace
```

Nonmember, Static Member, and Global Functions

> Prefer placing nonmember functions in a namespace; use completely global functions rarely. Prefer grouping functions with a namespace instead of using a class as if it were a namespace. Static methods of a class should generally be closely related to instances of the class or the class's static data.

Pros

Nonmember and static member functions can be useful in some situations. Putting nonmember functions in a namespace avoids polluting the global namespace.

Cons

Nonmember and static member functions may make more sense as members of a new class, especially if they access external resources or have significant dependencies.

Decision

Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables, and should nearly always exist in a namespace. Rather than creating classes only to group static member functions which do not share static data, use *namespaces* instead. For a header `myproject/foo_bar.h`, for example, write

```
namespace myproject {
namespace foo_bar {
void Function1();
void Function2();
}  // namespace foo_bar
}  // namespace myproject
```

instead of

```
namespace myproject {
class FooBar {
 public:
  static void Function1();
```

(continues on next page)

(continued from previous page)

```
  static void Function2();
};
}  // namespace myproject
```

If you define a nonmember function and it is only needed in its `.cpp` file, use *internal linkage* to limit its scope.

Local Variables

Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.:

```
int i;
i = f();      // Bad -- initialization separate from declaration.
```

```
int j = g();  // Good -- declaration has initialization.
```

```
std::vector<int> v;
v.push_back(1);  // Prefer initializing using brace initialization.
v.push_back(2);
```

```
std::vector<int> v = {1, 2};  // Good -- v starts initialized.
```

Variables needed for `if`, `while` and `for` statements should normally be declared within those statements, so that such variables are confined to those scopes. E.g.:

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

There is one caveat: if the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope.

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}
```

It may be more efficient to declare such a variable used in a loop outside that loop:

```
Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

Static and Global Variables

> Variables of class type with static storage duration are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction. However, such variables are allowed if they are `constexpr`: they have no dynamic initialization or destruction.

Objects with static storage duration, including global variables, static variables, static class member variables, and function static variables, must be Plain Old Data (POD): only ints, chars, floats, or pointers, or arrays/structs of POD.

The order in which class constructors and initializers for static variables are called is only partially specified in C++ and can even change from build to build, which can cause bugs that are difficult to find. Therefore in addition to banning globals of class type, we do not allow non-local static variables to be initialized with the result of a function, unless that function (such as `getenv()`, or `getpid()`) does not itself depend on any other globals. However, a static POD variable within function scope may be initialized with the result of a function, since its initialization order is well-defined and does not occur until control passes through its declaration.

Likewise, global and static variables are destroyed when the program terminates, regardless of whether the termination is by returning from `main()` or by calling `exit()`. The order in which destructors are called is defined to be the reverse of the order in which the constructors were called. Since constructor order is indeterminate, so is destructor order. For example, at program-end time a static variable might have been destroyed, but code still running - perhaps in another thread - tries to access it and fails. Or the destructor for a static `string` variable might be run prior to the destructor for another variable that contains a reference to that string.

One way to alleviate the destructor problem is to terminate the program by calling `quick_exit()` instead of `exit()`. The difference is that `quick_exit()` does not invoke destructors and does not invoke any handlers that were registered by calling `atexit()`. If you have a handler that needs to run when a program terminates via `quick_exit()` (flushing logs, for example), you can register it using `at_quick_exit()`. (If you have a handler that needs to run at both `exit()` and `quick_exit()`, you need to register it in both places.)

As a result we only allow static variables to contain POD data. This rule completely disallows `std::vector` (use C arrays instead), or `string` (use `const char []`).

If you need a static or global variable of a class type, consider initializing a pointer (which will never be freed), from either your `main()` function or from `pthread_once()`. Note that this must be a raw pointer, not a “smart” pointer, since the smart pointer’s destructor will have the order-of-destructor issue that we are trying to avoid.

11.1.7 Classes

Classes are the fundamental unit of code in C++. Naturally, we use them extensively. This section lists the main dos and don’ts you should follow when writing a class.

Doing Work in Constructors

> Avoid virtual method calls in constructors, and avoid initialization that can fail if you can’t signal an error.

Definition

It is possible to perform arbitrary initialization in the body of the constructor.

Pros

- No need to worry about whether the class has been initialized or not.
- Objects that are fully initialized by constructor call can be `const` and may also be easier to use with standard containers or algorithms.

Cons

- If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Future modification to your class can quietly introduce this problem even if your class is not currently subclassed, causing much confusion.
- There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions (which are *forbidden*).
- If the work fails, we now have an object whose initialization code failed, so it may be an unusual state requiring a `bool IsValid()` state checking mechanism (or similar) which is easy to forget to call.
- You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

Decision

Constructors should never call virtual functions. If appropriate for your code , terminating the program may be an appropriate error handling response. Otherwise, consider a factory function or `Init()` method. Avoid `Init()` methods on objects with no other states that affect which public methods may be called (semi-constructed objects of this form are particularly hard to work with correctly).

Implicit Conversions

> Do not define implicit conversions. Use the `explicit` keyword for conversion operators and single-argument constructors.

Definition

Implicit conversions allow an object of one type (called the *source type*) to be used where a different type (called the *destination type*) is expected, such as when passing an `int` argument to a function that takes a `double` parameter.

In addition to the implicit conversions defined by the language, users can define their own, by adding appropriate members to the class definition of the source or destination type. An implicit conversion in the source type is defined by a type conversion operator named after the destination type (e.g. `operator bool()`). An implicit conversion in the destination type is defined by a constructor that can take the source type as its only argument (or only argument with no default value).

The `explicit` keyword can be applied to a constructor or (since C++11) a conversion operator, to ensure that it can only be used when the destination type is explicit at the point of use, e.g. with a cast. This applies not only to implicit conversions, but to C++11's list initialization syntax:

```
class Foo {
  explicit Foo(int x, double y);
  ...
};

void Func(Foo f);
```

```
Func({42, 3.14});  // Error!
```

This kind of code isn't technically an implicit conversion, but the language treats it as one as far as `explicit` is concerned.

Pros

- Implicit conversions can make a type more usable and expressive by eliminating the need to explicitly name a type when it's obvious.
- Implicit conversions can be a simpler alternative to overloading.
- List initialization syntax is a concise and expressive way of initializing objects.

Cons

- Implicit conversions can hide type-mismatch bugs, where the destination type does not match the user's expectation, or the user is unaware that any conversion will take place.
- Implicit conversions can make code harder to read, particularly in the presence of overloading, by making it less obvious what code is actually getting called.
- Constructors that take a single argument may accidentally be usable as implicit type conversions, even if they are not intended to do so.
- When a single-argument constructor is not marked `explicit`, there's no reliable way to tell whether it's intended to define an implicit conversion, or the author simply forgot to mark it.
- It's not always clear which type should provide the conversion, and if they both do, the code becomes ambiguous.
- List initialization can suffer from the same problems if the destination type is implicit, particularly if the list has only a single element.

Decision

Type conversion operators, and constructors that are callable with a single argument, must be marked `explicit` in the class definition. As an exception, copy and move constructors should not be `explicit`, since they do not perform type conversion. Implicit conversions can sometimes be necessary and appropriate for types that are designed to transparently wrap other types. In that case, contact the Kaleidoscope maintainers to request a waiver of this rule.

Constructors that cannot be called with a single argument should usually omit `explicit`. Constructors that take a single `std::initializer_list` parameter should also omit `explicit`, in order to support copy-initialization (e.g. `MyType m = {1, 2};`).

Copyable and Movable Types

> Support copying and/or moving if these operations are clear and meaningful for your type. Otherwise, disable the implicitly generated special functions that perform copies and moves.

Definition

A copyable type allows its objects to be initialized or assigned from any other object of the same type, without changing the value of the source. For user-defined types, the copy behavior is defined by the copy constructor and the copy-assignment operator. `string` is an example of a copyable type.

A movable type is one that can be initialized and assigned from temporaries (all copyable types are therefore movable). `std::unique_ptr<int>` is an example of a movable but not copyable type. For user-defined types, the move behavior is defined by the move constructor and the move-assignment operator.

The copy/move constructors can be implicitly invoked by the compiler in some situations, e.g. when passing objects by value.

Pros

Objects of copyable and movable types can be passed and returned by value, which makes APIs simpler, safer, and more general. Unlike when passing objects by pointer or reference, there's no risk of confusion over ownership, lifetime, mutability, and similar issues, and no need to specify them in the contract. It also prevents non-local interactions between the client and the implementation, which makes them easier to understand, maintain, and optimize by the compiler. Further, such objects can be used with generic APIs that require pass-by-value, such as most containers, and they allow for additional flexibility in e.g., type composition.

Copy/move constructors and assignment operators are usually easier to define correctly than alternatives like `Clone()`, `CopyFrom()` or `Swap()`, because they can be generated by the compiler, either implicitly or with `= default`. They are concise, and ensure that all data members are copied. Copy and move constructors are also generally more efficient, because they don't require heap allocation or separate initialization and assignment steps, and they're eligible for optimizations such as copy elision.

Move operations allow the implicit and efficient transfer of resources out of rvalue objects. This allows a plainer coding style in some cases.

Cons

Some types do not need to be copyable, and providing copy operations for such types can be confusing, nonsensical, or outright incorrect. Types representing singleton objects (`Registerer`), objects tied to a specific scope (`Cleanup`), or closely coupled to object identity (`Mutex`) cannot be copied meaningfully. Copy operations for base class types that are to be used polymorphically are hazardous, because use of them can lead to object slicing. Defaulted or carelessly-implemented copy operations can be incorrect, and the resulting bugs can be confusing and difficult to diagnose.

Copy constructors are invoked implicitly, which makes the invocation easy to miss. This may cause confusion for programmers used to languages where pass-by-reference is conventional or mandatory. It may also encourage excessive copying, which can cause performance problems.

Decision

Provide the copy and move operations if their meaning is clear to a casual user and the copying/moving does not incur unexpected costs. If you define a copy or move constructor, define the corresponding assignment operator, and vice-versa. If your type is copyable, do not define move operations unless they are significantly more efficient than the corresponding copy operations. If your type is not copyable, but the correctness of a move is obvious to users of the type, you may make the type move-only by defining both of the move operations.

If your type provides copy operations, it is recommended that you design your class so that the default implementation of those operations is correct. Remember to review the correctness of any defaulted operations as you would any other code, and to document that your class is copyable and/or cheaply movable if that's an API guarantee.

```
class Foo {
 public:
  Foo(Foo&& other) : field_(other.field) {}
  // Bad, defines only move constructor, but not operator=.

 private:
  Field field_;
};
```

Due to the risk of slicing, avoid providing an assignment operator or public copy/move constructor for a class that's intended to be derived from (and avoid deriving from a class with such members). If your base class needs to be copyable, provide a public virtual `Clone()` method, and a protected copy constructor that derived classes can use to implement it.

If you do not want to support copy/move operations on your type, explicitly disable them using `= delete` in the `public:` section:

```
// MyClass is neither copyable nor movable.
MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
```

Structs vs. Classes

Use a `struct` only for passive objects that carry data; everything else is a `class`.

The `struct` and `class` keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

`structs` should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, `Initialize()`, `Reset()`, `Validate()`.

If more functionality is required, a `class` is more appropriate. If in doubt, make it a `class`.

For consistency with STL, you can use `struct` instead of `class` for functors and traits.

Note that member variables in structs and classes have *different naming rules*.

Inheritance

Composition is often more appropriate than inheritance. When using inheritance, make it `public`.

Definition

When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the base class defines. In practice, inheritance is used in two major ways in C++: implementation inheritance, in which actual code is inherited by the child, and *interface inheritance*, in which only method names are inherited.

Pros

Implementation inheritance reduces code size by re-using the base class code as it specializes an existing type. Because inheritance is a compile-time declaration, you and the compiler can understand the operation and detect errors. Interface inheritance can be used to programmatically enforce that a class expose a particular API. Again, the compiler can detect errors, in this case, when a class does not define a necessary method of the API.

Cons

For implementation inheritance, because the code implementing a sub-class is spread between the base and the sub-class, it can be more difficult to understand an implementation. The sub-class cannot override functions that are not virtual, so the sub-class cannot change implementation. The base class may also define some data members, so that specifies physical layout of the base class.

Decision

All inheritance should be `public`. If you want to do private inheritance, you should be including an instance of the base class as a member instead.

Do not overuse implementation inheritance. Composition is often more appropriate. Try to restrict use of inheritance to the *"is-a"* case: `Bar` subclasses `Foo` if it can reasonably be said that `Bar` "is a kind of" `Foo`.

Make your destructor `virtual` if necessary. If your class has virtual methods, its destructor should be virtual.

Limit the use of `protected` to those member functions that might need to be accessed from subclasses. Note that *data members should be private*.

Explicitly annotate overrides of virtual functions or virtual destructors with an `override` or (less frequently) `final` specifier. Older (pre-C++11) code will use the `virtual` keyword as an inferior alternative annotation. For clarity, use exactly one of `override`, `final`, or `virtual` when declaring an override. Rationale: A function or destructor marked `override` or `final` that is not an override of a base class virtual function will not compile, and this helps catch common errors. The specifiers serve as documentation; if no specifier is present, the reader has to check all ancestors of the class in question to determine if the function or destructor is virtual or not.

Multiple Inheritance

Only very rarely is multiple implementation inheritance actually useful. We allow multiple inheritance only when at most one of the base classes has an implementation; all other base classes must be *pure interface* classes tagged with the `Interface` suffix.

Definition

Multiple inheritance allows a sub-class to have more than one base class. We distinguish between base classes that are *pure interfaces* and those that have an *implementation*.

Pros

Multiple implementation inheritance may let you re-use even more code than single inheritance (see *Inheritance*).

Cons

Only very rarely is multiple *implementation* inheritance actually useful. When multiple implementation inheritance seems like the solution, you can usually find a different, more explicit, and cleaner solution.

Decision

Multiple inheritance is allowed only when all superclasses, with the possible exception of the first one, are *pure interfaces*. In order to ensure that they remain pure interfaces, they must end with the `Interface` suffix.

Interfaces

Classes that satisfy certain conditions are allowed, but not required, to end with an `Interface` suffix.

Definition

A class is a pure interface if it meets the following requirements:

- It has only public pure virtual (”`= 0`”) methods and static methods (but see below for destructor).
- It may not have non-static data members.
- It need not have any constructors defined. If a constructor is provided, it must take no arguments and it must be protected.
- If it is a subclass, it may only be derived from classes that satisfy these conditions and are tagged with the `Interface` suffix.

An interface class can never be directly instantiated because of the pure virtual method(s) it declares. To make sure all implementations of the interface can be destroyed correctly, the interface must also declare a virtual destructor (in an exception to the first rule, this should not be pure). See Stroustrup, The C++ Programming Language, 3rd edition, section 12.4 for details.

Pros

Tagging a class with the `Interface` suffix lets others know that they must not add implemented methods or non static data members. This is particularly important in the case of *multiple inheritance*. Additionally, the interface concept is already well-understood by Java programmers.

Cons

The `Interface` suffix lengthens the class name, which can make it harder to read and understand. Also, the interface property may be considered an implementation detail that shouldn’t be exposed to clients.

Decision

A class may end with `Interface` only if it meets the above requirements. We do not require the converse, however: classes that meet the above requirements are not required to end with `Interface`.

Operator Overloading

Overload operators judiciously. Do not create user-defined literals.

Definition

C++ permits user code to declare overloaded versions of the built-in operators using the `operator` keyword, so long as one of the parameters is a user-defined type. The `operator` keyword also permits user code to define new kinds of literals using `operator""`, and to define type-conversion functions such as `operator bool()`.

Pros

Operator overloading can make code more concise and intuitive by enabling user-defined types to behave the same as built-in types. Overloaded operators are the idiomatic names for certain operations (e.g. ==, <, =, and <<), and adhering to those conventions can make user-defined types more readable and enable them to interoperate with libraries that expect those names.

User-defined literals are a very concise notation for creating objects of user-defined types.

Cons

- Providing a correct, consistent, and unsurprising set of operator overloads requires some care, and failure to do so can lead to confusion and bugs.
- Overuse of operators can lead to obfuscated code, particularly if the overloaded operator's semantics don't follow convention.
- The hazards of function overloading apply just as much to operator overloading, if not more so.
- Operator overloads can fool our intuition into thinking that expensive operations are cheap, built-in operations.
- Finding the call sites for overloaded operators may require a search tool that's aware of C++ syntax, rather than e.g. grep.
- If you get the argument type of an overloaded operator wrong, you may get a different overload rather than a compiler error. For example, `foo < bar` may do one thing, while `&foo < &bar` does something totally different.
- Certain operator overloads are inherently hazardous. Overloading unary `&` can cause the same code to have different meanings depending on whether the overload declaration is visible. Overloads of `&&`, `||`, and `,` (comma) cannot match the evaluation-order semantics of the built-in operators.
- Operators are often defined outside the class, so there's a risk of different files introducing different definitions of the same operator. If both definitions are linked into the same binary, this results in undefined behavior, which can manifest as subtle run-time bugs.
- User-defined literals allow the creation of new syntactic forms that are unfamiliar even to experienced C++ programmers.

Decision

Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators. For example, use `|` as a bitwise- or logical-or, not as a shell-style pipe.

Define operators only on your own types. More precisely, define them in the same headers, `.cpp` files, and namespaces as the types they operate on. That way, the operators are available wherever the type is, minimizing the risk of multiple definitions. If possible, avoid defining operators as templates, because they must satisfy this rule for any possible template arguments. If you define an operator, also define any related operators that make sense, and make sure they are defined consistently. For example, if you overload <, overload all the comparison operators, and make sure < and > never return true for the same arguments.

Prefer to define non-modifying binary operators as non-member functions. If a binary operator is defined as a class member, implicit conversions will apply to the right-hand argument, but not the left-hand one. It will confuse your users if `a < b` compiles but `b < a` doesn't.

Don't go out of your way to avoid defining operator overloads. For example, prefer to define ==, =, and <<, rather than `Equals()`, `CopyFrom()`, and `PrintTo()`. Conversely, don't define operator overloads just because other libraries expect them. For example, if your type doesn't have a natural ordering, but you want to store it in a `std::set`, use a custom comparator rather than overloading <.

Do not overload &&, `||`, `,` (comma), or unary &. Do not overload `operator""`, i.e. do not introduce user-defined literals.

Type conversion operators are covered in the section on *implicit conversions*. The = operator is covered in the section on *copy constructors* . Overloading << for use with streams is covered in the section on *streams*. See also the rules on *function overloading* , which apply to operator overloading as well.

Access Control

> Make data members `private`, unless they are `static const` (and follow the *naming convention for constants*). For technical reasons, we allow data members of a test fixture class to be `protected` when using Google Test).

Declaration Order

> Group similar declarations together, placing public parts earlier.

A class definition should usually start with a `public:` section, followed by `protected:`, then `private:`. Omit sections that would be empty.

Within each section, generally prefer grouping similar kinds of declarations together, and generally prefer the following order: types (including `typedef`, `using`, and nested structs and classes), constants, factory functions, constructors, assignment operators, destructor, all other methods, data members.

Do not put large method definitions inline in the class definition. Usually, only trivial or performance-critical, and very short, methods may be defined inline. See *Inline Functions* for more details.

11.1.8 Functions

11.1.9 Parameter Ordering

> When defining a function, parameter order is: inputs, then outputs.

Parameters to C/C++ functions are either input to the function, output from the function, or both. Input parameters are usually values or `const` references, while output and input/output parameters will be pointers to non-`const`. When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters.

This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule.

11.1.10 Write Short Functions

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

11.1.11 Reference Arguments

All parameters passed by reference must be labeled `const`.

Definition

In C, if a function needs to modify a variable, the parameter must use a pointer, eg `int foo(int *pval)`. In C++, the function can alternatively declare a reference parameter: `int foo(int &val)`.

Pros

Defining a parameter as reference avoids ugly code like `(*pval)++`. Necessary for some applications like copy constructors. Makes it clear, unlike with pointers, that a null pointer is not a possible value.

Cons

References can be confusing, as they have value syntax but pointer semantics.

Decision

Within function parameter lists all references must be `const`:

```
void Foo(const string &in, string *out);
```

In fact it is a very strong convention in Google code that input arguments are values or `const` references while output arguments are pointers. Input parameters may be `const` pointers, but we never allow non-`const` reference parameters except when required by convention, e.g., `swap()`.

However, there are some instances where using `const T*` is preferable to `const T&` for input parameters. For example:

- You want to pass in a null pointer.
- The function saves a pointer or reference to the input.

Remember that most of the time input parameters are going to be specified as `const T&`. Using `const T*` instead communicates to the reader that the input is somehow treated differently. So if you choose `const T*` rather than `const T&`, do so for a concrete reason; otherwise it will likely confuse readers by making them look for an explanation that doesn't exist.

11.1.12 Function Overloading

> Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called.

Definition

You may write a function that takes a `const string&` and overload it with another that takes `const char*`.

```
class MyClass {
 public:
  void Analyze(const string &text);
  void Analyze(const char *text, size_t textlen);
};
```

Pros

Overloading can make code more intuitive by allowing an identically-named function to take different arguments. It may be necessary for templatized code, and it can be convenient for Visitors.

Cons

If a function is overloaded by the argument types alone, a reader may have to understand C++'s complex matching rules in order to tell what's going on. Also many people are confused by the semantics of inheritance if a derived class overrides only some of the variants of a function.

Decision

If you want to overload a function, consider qualifying the name with some information about the arguments, e.g., `AppendString()`, `AppendInt()` rather than just `Append()`. If you are overloading a function to support variable number of arguments of the same type, consider making it take a `std::vector` so that the user can use an *initializer list* to specify the arguments.

Default Arguments

> Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value. Follow the same restrictions as for *function overloading*, and prefer overloaded functions if the readability gained with default arguments doesn't outweigh the downsides below.

Pros

Often you have a function that uses default values, but occasionally you want to override the defaults. Default parameters allow an easy way to do this without having to define many functions for the rare exceptions. Compared to overloading the function, default arguments have a cleaner syntax, with less boilerplate and a clearer distinction between 'required' and 'optional' arguments.

Cons

Defaulted arguments are another way to achieve the semantics of overloaded functions, so all the *reasons not to overload functions* apply.

The defaults for arguments in a virtual function call are determined by the static type of the target object, and there's no guarantee that all overrides of a given function declare the same defaults.

Default parameters are re-evaluated at each call site, which can bloat the generated code. Readers may also expect the default's value to be fixed at the declaration instead of varying at each call.

Function pointers are confusing in the presence of default arguments, since the function signature often doesn't match the call signature. Adding function overloads avoids these problems.

Decision

Default arguments are banned on virtual functions, where they don't work properly, and in cases where the specified default might not evaluate to the same value depending on when it was evaluated. (For example, don't write `void f(int n = counter++);`.)

In some other cases, default arguments can improve the readability of their function declarations enough to overcome the downsides above, so they are allowed. When in doubt, use overloads.

Trailing Return Type Syntax

> Use trailing return types only where using the ordinary syntax (leading return types) is impractical or much less readable.

Definition

C++ allows two different forms of function declarations. In the older form, the return type appears before the function name. For example:

```
int foo(int x);
```

The new form, introduced in C++11, uses the `auto` keyword before the function name and a trailing return type after the argument list. For example, the declaration above could equivalently be written:

```
auto foo(int x) -> int;
```

The trailing return type is in the function's scope. This doesn't make a difference for a simple case like `int` but it matters for more complicated cases, like types declared in class scope or types written in terms of the function parameters.

Pros

Trailing return types are the only way to explicitly specify the return type of a *lambda expression*. In some cases the compiler is able to deduce a lambda's return type, but not in all cases. Even when the compiler can deduce it automatically, sometimes specifying it explicitly would be clearer for readers.

Sometimes it's easier and more readable to specify a return type after the function's parameter list has already appeared. This is particularly true when the return type depends on template parameters. For example:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

versus

```
template <class T, class U> decltype(declvall<T>() + declval<U>()) add(T t, U u);
```

Cons

Trailing return type syntax is relatively new and it has no analogue in C++-like languages like C and Java, so some readers may find it unfamiliar.

Existing code bases have an enormous number of function declarations that aren't going to get changed to use the new syntax, so the realistic choices are using the old syntax only or using a mixture of the two. Using a single version is better for uniformity of style.

Decision

In most cases, continue to use the older style of function declaration where the return type goes before the function name. Use the new trailing-return-type form only in cases where it's required (such as lambdas) or where, by putting the type after the function's parameter list, it allows you to write the type in a much more readable way. The latter case should be rare; it's mostly an issue in fairly complicated template code, which is *discouraged in most cases*.

11.1.13 Google-Specific Magic

There are various tricks and utilities that we use to make C++ code more robust, and various ways we use C++ that may differ from what you see elsewhere.

Ownership and Smart Pointers

> Prefer to have single, fixed owners for dynamically allocated objects. Prefer to transfer ownership with smart pointers.

Definition

"Ownership" is a bookkeeping technique for managing dynamically allocated memory (and other resources). The owner of a dynamically allocated object is an object or function that is responsible for ensuring that it is deleted when no longer needed. Ownership can sometimes be shared, in which case the last owner is typically responsible for deleting it. Even when ownership is not shared, it can be transferred from one piece of code to another.

"Smart" pointers are classes that act like pointers, e.g. by overloading the * and -> operators. Some smart pointer types can be used to automate ownership bookkeeping, to ensure these responsibilities are met. `std::unique_ptr` is a smart pointer type introduced in C++11, which expresses exclusive ownership of a dynamically allocated object; the object is deleted when the `std::unique_ptr` goes out of scope. It cannot be copied, but can be *moved* to represent ownership transfer. `std::shared_ptr` is a smart pointer type that expresses shared ownership of a dynamically allocated object. `std::shared_ptr`s can be copied; ownership of the object is shared among all copies, and the object is deleted when the last `std::shared_ptr` is destroyed.

Pros

- It's virtually impossible to manage dynamically allocated memory without some sort of ownership logic.
- Transferring ownership of an object can be cheaper than copying it (if copying it is even possible).
- Transferring ownership can be simpler than 'borrowing' a pointer or reference, because it reduces the need to coordinate the lifetime of the object between the two users.
- Smart pointers can improve readability by making ownership logic explicit, self-documenting, and unambiguous.
- Smart pointers can eliminate manual ownership bookkeeping, simplifying the code and ruling out large classes of errors.
- For const objects, shared ownership can be a simple and efficient alternative to deep copying.

Cons

- Ownership must be represented and transferred via pointers (whether smart or plain). Pointer semantics are more complicated than value semantics, especially in APIs: you have to worry not just about ownership, but also aliasing, lifetime, and mutability, among other issues.
- The performance costs of value semantics are often overestimated, so the performance benefits of ownership transfer might not justify the readability and complexity costs.
- APIs that transfer ownership force their clients into a single memory management model.
- Code using smart pointers is less explicit about where the resource releases take place.
- `std::unique_ptr` expresses ownership transfer using C++11's move semantics, which are relatively new and may confuse some programmers.
- Shared ownership can be a tempting alternative to careful ownership design, obfuscating the design of a system.
- Shared ownership requires explicit bookkeeping at run-time, which can be costly.
- In some cases (e.g. cyclic references), objects with shared ownership may never be deleted.

- Smart pointers are not perfect substitutes for plain pointers.

Decision

If dynamic allocation is necessary, prefer to keep ownership with the code that allocated it. If other code needs access to the object, consider passing it a copy, or passing a pointer or reference without transferring ownership. Prefer to use `std::unique_ptr` to make ownership transfer explicit. For example:

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Do not design your code to use shared ownership without a very good reason. One such reason is to avoid expensive copy operations, but you should only do this if the performance benefits are significant, and the underlying object is immutable (i.e. `std::shared_ptr<const Foo>`). If you do use shared ownership, prefer to use `std::shared_ptr`.

Never use `std::auto_ptr`. Instead, use `std::unique_ptr`.

cpplint

> Use `cpplint.py` to detect style errors.

`cpplint.py` is a tool that reads a source file and identifies many style errors. It is not perfect, and has both false positives and false negatives, but it is still a valuable tool. False positives can be ignored by putting `// NOLINT` at the end of the line or `// NOLINTNEXTLINE` in the previous line.

Some projects have instructions on how to run `cpplint.py` from their project tools. If the project you are contributing to does not, you can download `cpplint.py` separately.

11.1.14 Other C++ Features

Rvalue References

> Use rvalue references only to define move constructors and move assignment operators, or for perfect forwarding.

Definition

Rvalue references are a type of reference that can only bind to temporary objects. The syntax is similar to traditional reference syntax. For example, `void f(string&& s);` declares a function whose argument is an rvalue reference to a string.

Pros

- Defining a move constructor (a constructor taking an rvalue reference to the class type) makes it possible to move a value instead of copying it. If `v1` is a `std::vector<string>`, for example, then `auto v2(std::move(v1))` will probably just result in some simple pointer manipulation instead of copying a large amount of data. In some cases this can result in a major performance improvement.
- Rvalue references make it possible to write a generic function wrapper that forwards its arguments to another function, and works whether or not its arguments are temporary objects. (This is sometimes called "perfect forwarding".)
- Rvalue references make it possible to implement types that are movable but not copyable, which can be useful for types that have no sensible definition of copying but where you might still want to pass them as function arguments, put them in containers, etc.
- `std::move` is necessary to make effective use of some standard-library types, such as `std::unique_ptr`.

Cons

- Rvalue references are a relatively new feature (introduced as part of C++11), and not yet widely understood. Rules like reference collapsing, and automatic synthesis of move constructors, are complicated.

Decision

Use rvalue references only to define move constructors and move assignment operators (as described in *Copyable and Movable Types* and, in conjunction with `std::forward`, to support perfect forwarding. You may use `std::move` to express moving a value from one object to another rather than copying it.

Friends

We allow use of `friend` classes and functions, within reason.

Friends should usually be defined in the same file so that the reader does not have to look in another file to find uses of the private members of a class. A common use of `friend` is to have a `FooBuilder` class be a friend of `Foo` so that it can construct the inner state of `Foo` correctly, without exposing this state to the world. In some cases it may be useful to make a unittest class a friend of the class it tests.

Friends extend, but do not break, the encapsulation boundary of a class. In some cases this is better than making a member public when you want to give only one other class access to it. However, most classes should interact with other classes solely through their public members.

Exceptions

We do not use C++ exceptions.

Pros

- Exceptions allow higher levels of an application to decide how to handle "can't happen" failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.
- Exceptions are used by most other modern languages. Using them in C++ would make it more consistent with Python, Java, and the C++ that others are familiar with.
- Some third-party C++ libraries use exceptions, and turning them off internally makes it harder to integrate with those libraries.
- Exceptions are the only way for a constructor to fail. We can simulate this with a factory function or an `Init()` method, but these require heap allocation or a new "invalid" state, respectively.
- Exceptions are really handy in testing frameworks.

Cons

- When you add a `throw` statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if `f()` calls `g()` calls `h()`, and `h` throws an exception that `f` catches, `g` has to be careful or it may not clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimize this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.
- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.

- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
- The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so. For example, invalid user input should not cause exceptions to be thrown. We would need to make the style guide even longer to document these restrictions!

Decision

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code. Because most existing C++ code at Google is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.

Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

This prohibition also applies to the exception-related features added in C++11, such as `noexcept`, `std::exception_ptr`, and `std::nested_exception`.

Run-Time Type Information (RTTI)

Avoid using Run Time Type Information (RTTI).

Definition

RTTI allows a programmer to query the C++ class of an object at run time. This is done by use of `typeid` or `dynamic_cast`.

Cons

Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed.

Undisciplined use of RTTI makes code hard to maintain. It can lead to type-based decision trees or switch statements scattered throughout the code, all of which must be examined when making further changes.

Pros

The standard alternatives to RTTI (described below) require modification or redesign of the class hierarchy in question. Sometimes such modifications are infeasible or undesirable, particularly in widely-used or mature code.

RTTI can be useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type. It is also useful in managing the relationship between objects and their mocks.

RTTI is useful when considering multiple abstract objects. Consider

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == NULL)
    return false;
```

(continues on next page)

(continued from previous page)

```
  ...
}
```

Decision

RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unittests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code. If you find yourself needing to write code that behaves differently based on the class of an object, consider one of the following alternatives to querying the type:

- Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
- If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

When the logic of a program guarantees that a given instance of a base class is in fact an instance of a particular derived class, then a `dynamic_cast` may be used freely on the object. Usually one can use a `static_cast` as an alternative in such situations.

Decision trees based on type are a strong indication that your code is on the wrong track.

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

Code such as this usually breaks when additional subclasses are added to the class hierarchy. Moreover, when properties of a subclass change, it is difficult to find and modify all the affected code segments.

Do not hand-implement an RTTI-like workaround. The arguments against RTTI apply just as much to workarounds like class hierarchies with type tags. Moreover, workarounds disguise your true intent.

Casting

Use C++-style casts like `static_cast<float>(double_value)`, or brace initialization for conversion of arithmetic types like `int64 y = int64{1} << 42`. Do not use cast formats like `int y = (int)x` or `int y = int(x)` (but the latter is okay when invoking a constructor of a class type).

Definition

C++ introduced a different cast system from C that distinguishes the types of cast operations.

Pros

The problem with C casts is the ambiguity of the operation; sometimes you are doing a *conversion* (e.g., `(int)3.5`) and sometimes you are doing a *cast* (e.g., `(int)"hello"`). Brace initialization and C++ casts can often help avoid this ambiguity. Additionally, C++ casts are more visible when searching for them.

Cons The C++-style cast syntax is verbose and cumbersome.

Decision

Do not use C-style casts. Instead, use these C++-style casts when explicit type conversion is necessary.

- Use brace initialization to convert arithmetic types (e.g. `int64{x}`). This is the safest approach because code will not compile if conversion can result in information loss. The syntax is also concise.
- Use `static_cast` as the equivalent of a C-style cast that does value conversion, when you need to explicitly up-cast a pointer from a class to its superclass, or when you need to explicitly cast a pointer from a superclass to a subclass. In this last case, you must be sure your object is actually an instance of the subclass.
- Use `const_cast` to remove the `const` qualifier (see *const*).
- Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.

See the *RTTI section* for guidance on the use of `dynamic_cast`.

Streams

Use streams where appropriate, and stick to "simple" usages.

Definition

Streams are the standard I/O abstraction in C++, as exemplified by the standard header `<iostream>`. They are widely used in Google code, but only for debug logging and test diagnostics.

Pros

The ``<<`` and >>stream operators provide an API for formatted I/O that is easily learned, portable, reusable, and extensible.printf, by contrast, doesn't even support string, to say nothing of user-defined types, and is very difficult to use portably.`` printf` also obliges you to choose among the numerous slightly different versions of that function, and navigate the dozens of conversion specifiers.

Streams provide first-class support for console I/O via `std::cin`, `std::cout`, `std::cerr`, and `std::clog`. The C APIs do as well, but are hampered by the need to manually buffer the input.

Cons

- Stream formatting can be configured by mutating the state of the stream. Such mutations are persistent, so the behavior of your code can be affected by the entire previous history of the stream, unless you go out of your way to restore it to a known state every time other code might have touched it. User code can not only modify the built-in state, it can add new state variables and behaviors through a registration system.
- It is difficult to precisely control stream output, due to the above issues, the way code and data are mixed in streaming code, and the use of operator overloading (which may select a different overload than you expect).
- The practice of building up output through chains of << operators interferes with internationalization, because it bakes word order into the code, and streams' support for localization is flawed.
- The streams API is subtle and complex, so programmers must develop experience with it in order to use it effectively. However, streams were historically banned in Google code (except for logging and diagnostics), so Google engineers tend not to have that experience. Consequently, streams-based code is likely to be less readable and maintainable by Googlers than code based on more familiar abstractions.
- Resolving the many overloads of << is extremely costly for the compiler. When used pervasively in a large code base, it can consume as much as 20% of the parsing and semantic analysis time.

Decision

Use streams only when they are the best tool for the job. This is typically the case when the I/O is ad-hoc, local, human-readable, and targeted at other developers rather than end-users. Be consistent with the code around you, and with the codebase as a whole; if there's an established tool for your problem, use that tool instead.

Avoid using streams for I/O that faces external users or handles untrusted data. Instead, find and use the appropriate templating libraries to handle issues like internationalization, localization, and security hardening.

If you do use streams, avoid the stateful parts of the streams API (other than error state), such as `imbue()`, `xalloc()`, and `register_callback()`. Use explicit formatting functions rather than stream manipulators or formatting flags to control formatting details such as number base, precision, or padding.

Overload << as a streaming operator for your type only if your type represents a value, and << writes out a human-readable string representation of that value. Avoid exposing implementation details in the output of <<; if you need to print object internals for debugging, use named functions instead (a method named `DebugString()` is the most common convention).

Preincrement and Predecrement

Use prefix form (`++i`) of the increment and decrement operators with iterators and other template objects.

Definition

When a variable is incremented (`++i` or `i++`) or decremented (`--i` or `i--`) and the value of the expression is not used, one must decide whether to preincrement (decrement) or postincrement (decrement).

Pros

When the return value is ignored, the "pre" form (`++i`) is never less efficient than the "post" form (`i++`), and is often more efficient. This is because post-increment (or decrement) requires a copy of `i` to be made, which is the value of the expression. If `i` is an iterator or other non-scalar type, copying `i` could be expensive. Since the two types of increment behave the same when the value is ignored, why not just always pre-increment?

Cons

The tradition developed, in C, of using post-increment when the expression value is not used, especially in `for` loops. Some find post-increment easier to read, since the "subject" (`i`) precedes the "verb" (`++`), just like in English.

Decision

For simple scalar (non-object) values there is no reason to prefer one form and we allow either. For iterators and other template types, use pre-increment.

Use of const

Use `const` whenever it makes sense. With C++11, `constexpr` is a better choice for some uses of const.

Definition

Declared variables and parameters can be preceded by the keyword `const` to indicate the variables are not changed (e.g., `const int foo`). Class functions can have the `const` qualifier to indicate the function does not change the state of the class member variables (e.g., `class Foo { int Bar(char c) const; };`).

Pros

Easier for people to understand how variables are being used. Allows the compiler to do better type checking, and, conceivably, generate better code. Helps people convince themselves of program correctness because they know the functions they call are limited in how they can modify your variables. Helps people know what functions are safe to use without locks in multi-threaded programs.

Cons

`const` is viral: if you pass a `const` variable to a function, that function must have `const` in its prototype (or the variable will need a `const_cast`). This can be a particular problem when calling library functions.

Decision

`const` variables, data members, methods and arguments add a level of compile-time type checking; it is better to detect errors as soon as possible. Therefore we strongly recommend that you use `const` whenever it makes sense to do so:

- If a function guarantees that it will not modify an argument passed by reference or by pointer, the corresponding function parameter should be a reference-to-const (`const T&`) or pointer-to-const (`const T*`), respectively.
- Declare methods to be `const` whenever possible. Accessors should almost always be `const`. Other methods should be const if they do not modify any data members, do not call any non-`const` methods, and do not return a non-`const` pointer or non-`const` reference to a data member.
- Consider making data members `const` whenever they do not need to be modified after construction.

The `mutable` keyword is allowed but is unsafe when used with threads, so thread safety should be carefully considered first.

Where to put the const

Some people favor the form `int const *foo` to `const int* foo`. They argue that this is more readable because it's more consistent: it keeps the rule that `const` always follows the object it's describing. However, this consistency argument doesn't apply in codebases with few deeply-nested pointer expressions since most `const` expressions have only one `const`, and it applies to the underlying value. In such cases, there's no consistency to maintain. Putting the `const` first is arguably more readable, since it follows English in putting the "adjective" (`const`) before the "noun" (`int`).

That said, while we encourage putting `const` first, we do not require it. But be consistent with the code around you!

Use of constexpr

In C++11, use `constexpr` to define true constants or to ensure constant initialization.

Definition

Some variables can be declared `constexpr` to indicate the variables are true constants, i.e. fixed at compilation/link time. Some functions and constructors can be declared `constexpr` which enables them to be used in defining a `constexpr` variable.

Pros

Use of `constexpr` enables definition of constants with floating-point expressions rather than just literals; definition of constants of user-defined types; and definition of constants with function calls.

Cons

Prematurely marking something as constexpr may cause migration problems if later on it has to be downgraded. Current restrictions on what is allowed in constexpr functions and constructors may invite obscure workarounds in these definitions.

Decision

`constexpr` definitions enable a more robust specification of the constant parts of an interface. Use `constexpr` to specify true constants and the functions that support their definitions. Avoid complexifying function definitions to enable their use with `constexpr`. Do not use `constexpr` to force inlining.

Integer Types

Of the built-in C++ integer types, the only one used is `int`. If a program needs a variable of a different size, use a precise-width integer type from `<stdint.h>`, such as `int16_t`. If your variable represents a value that could ever be greater than or equal to 2^31 (2GiB), use a 64-bit type such as `int64_t`. Keep in mind that even if your value won't ever be too large for an `int`, it may be used in intermediate calculations which may require a larger type. When in doubt, choose a larger type.

Definition

C++ does not specify the sizes of its integer types. Typically people assume that `short` is 16 bits, `int` is 32 bits, `long` is 32 bits and `long long` is 64 bits.

Pros

Uniformity of declaration.

Cons

The sizes of integral types in C++ can vary based on compiler and architecture.

Decision

`<stdint.h>` defines types like `int16_t`, `uint32_t`, `int64_t`, etc. You should always use those in preference to `short`, `unsigned long long` and the like, when you need a guarantee on the size of an integer. Of the C integer types, only `int` should be used. When appropriate, you are welcome to use standard types like `size_t` and `ptrdiff_t`.

We use `int` very often, for integers we know are not going to be too big, e.g., loop counters. Use plain old `int` for such things. You should assume that an `int` is at least 32 bits, but don't assume that it has more than 32 bits. If you need a 64-bit integer type, use `int64_t` or `uint64_t`.

For integers we know can be "big", use `int64_t`.

You should not use the unsigned integer types such as `uint32_t`, unless there is a valid reason such as representing a bit pattern rather than a number, or you need defined overflow modulo 2^N. In particular, do not use unsigned types to say a number will never be negative. Instead, use assertions for this.

If your code is a container that returns a size, be sure to use a type that will accommodate any possible usage of your container. When in doubt, use a larger type rather than a smaller type.

Use care when converting integer types. Integer conversions and promotions can cause non-intuitive behavior.

On Unsigned Integers

Some people, including some textbook authors, recommend using unsigned types to represent numbers that are never negative. This is intended as a form of self-documentation. However, in C, the advantages of such documentation are outweighed by the real bugs it can introduce. Consider:

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

This code will never terminate! Sometimes gcc will notice this bug and warn you, but often it will not. Equally bad bugs can occur when comparing signed and unsigned variables. Basically, C's type-promotion scheme causes unsigned types to behave differently than one might expect.

So, document that a variable is non-negative using assertions. Don't use an unsigned type.

Preprocessor Macros

> Avoid defining macros, especially in headers; prefer inline functions, enums, and `const` variables. Name macros with a project-specific prefix. Do not use macros to define pieces of a C++ API.

Macros mean that the code you see is not the same as the code the compiler sees. This can introduce unexpected behavior, especially since macros have global scope.

The problems introduced by macros are especially severe when they are used to define pieces of a C++ API, and still more so for public APIs. Every error message from the compiler when developers incorrectly use that interface now must explain how the macros formed the interface. Refactoring and analysis tools have a dramatically harder time updating the interface. As a consequence, we specifically disallow using macros in this way. For example, avoid patterns like:

```
class WOMBAT_TYPE(Foo) {
  // ...

 public:
  EXPAND_PUBLIC_WOMBAT_API(Foo)

  EXPAND_WOMBAT_COMPARISONS(Foo, ==, <)
};
```

Luckily, macros are not nearly as necessary in C++ as they are in C. Instead of using a macro to inline performance-critical code, use an inline function. Instead of using a macro to store a constant, use a `const` variable. Instead of using a macro to "abbreviate" a long variable name, use a reference. Instead of using a macro to conditionally compile code . . . well, don't do that at all. It makes testing much more difficult.

Macros can do things these other techniques cannot, and you do see them in the codebase, especially in the lower-level libraries. And some of their special features (like stringifying, concatenation, and so forth) are not available through the language proper. But before using a macro, consider carefully whether there's a non-macro way to achieve the same result. If you need to use a macro to define an interface, contact your project leads to request a waiver of this rule.

The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:

- Don't define macros in a `.h` file.
- `#define` macros right before you use them, and `#undef` them right after.
- Do not just `#undef` an existing macro before replacing it with your own; instead, pick a name that's likely to be unique.
- Try not to use macros that expand to unbalanced C++ constructs, or at least document that behavior well.
- Prefer not using `##` to generate function/class/variable names.

Exporting macros from headers (i.e. defining them in a header without `#undef`ing them before the end of the header) is extremely strongly discouraged. If you do export a macro from a header, it must have a globally unique name. To achieve this, it must be named with a prefix consisting of your project's namespace name (but upper case).

0 and nullptr/NULL

> Use 0 for integers, 0.0 for reals, nullptr (or NULL) for pointers, and '\0' for chars.

Use 0 for integers and 0.0 for reals. This is not controversial.

For pointers (address values), there is a choice between 0, NULL, and nullptr. For projects that allow C++11 features, use nullptr. For C++03 projects, we prefer NULL because it looks like a pointer. In fact, some C++ compilers provide special definitions of NULL which enable them to give useful warnings, particularly in situations where sizeof(NULL) is not equal to sizeof(0).

Use '\0' for chars. This is the correct type and also makes code more readable.

sizeof

> Prefer sizeof(varname) to sizeof(type).

Use sizeof(varname) when you take the size of a particular variable. sizeof(varname) will update appropriately if someone changes the variable type either now or later. You may use sizeof(type) for code unrelated to any particular variable, such as code that manages an external or internal data format where a variable of an appropriate C++ type is not convenient.

Good code

```
Struct data;
memset(&data, 0, sizeof(data));
```

Bad code

```
memset(&data, 0, sizeof(Struct));
```

Good code

```
if (raw_size < sizeof(int)) {
  LOG(ERROR) << "compressed record not big enough for count: " << raw_size;
  return false;
}
```

auto

> Use auto to avoid type names that are noisy, obvious, or unimportant - cases where the type doesn't aid in clarity for the reader. Continue to use manifest type declarations when it helps readability.

Pros

- C++ type names can be long and cumbersome, especially when they involve templates or namespaces.
- When a C++ type name is repeated within a single declaration or a small code region, the repetition may not be aiding readability.
- It is sometimes safer to let the type be specified by the type of the initialization expression, since that avoids the possibility of unintended copies or type conversions.

Cons

Sometimes code is clearer when types are manifest, especially when a variable's initialization depends on things that were declared far away. In expressions like:

```
auto foo = x.add_foo();
auto i = y.Find(key);
```

it may not be obvious what the resulting types are if the type of `y` isn't very well known, or if `y` was declared many lines earlier.

Programmers have to understand the difference between `auto` and `const auto&` or they'll get copies when they didn't mean to.

If an `auto` variable is used as part of an interface, e.g. as a constant in a header, then a programmer might change its type while only intending to change its value, leading to a more radical API change than intended.

Decision

`auto` is permitted when it increases readability, particularly as described below. Never initialize an `auto`-typed variable with a braced initializer list.

Specific cases where `auto` is allowed or encouraged:

- (*Encouraged*) For iterators and other long/cluttery type names, particularly when the type is clear from context (calls to `find`, `begin`, or `end` for instance).
- (*Allowed*) When the type is clear from local context (in the same expression or within a few lines). Initialization of a pointer or smart pointer with calls to `new` commonly falls into this category, as does use of `auto` in a range-based loop over a container whose type is spelled out nearby.
- (*Allowed*) When the type doesn't matter because it isn't being used for anything other than equality comparison.
- (*Encouraged*) When iterating over a map with a range-based loop (because it is often assumed that the correct type is `std::pair<KeyType, ValueType>` whereas it is actually `std::pair<const KeyType, ValueType>`). This is particularly well paired with local `key` and `value` aliases for `.first` and `.second` (often const-ref).

  ```
  for (const auto& item : some_map) {
    const KeyType& key = item.first;
    const ValType& value = item.second;
    // The rest of the loop can now just refer to key and value,
    // a reader can see the types in question, and we've avoided
    // the too-common case of extra copies in this iteration.
  }
  ```

Braced Initializer List

You may use braced initializer lists.

In C++03, aggregate types (arrays and structs with no constructor) could be initialized with braced initializer lists.

```
struct Point { int x; int y; };
Point p = {1, 2};
```

In C++11, this syntax was generalized, and any object type can now be created with a braced initializer list, known as a *braced-init-list* in the C++ grammar. Here are a few examples of its use.

```
// Vector takes a braced-init-list of elements.
std::vector<string> v{"foo", "bar"};

// Basically the same, ignoring some small technicalities.
// You may choose to use either form.
```

(continues on next page)

(continued from previous page)

```
std::vector<string> v = {"foo", "bar"};

// Usable with 'new' expressions.
auto p = new vector<string>{"foo", "bar"};

// A map can take a list of pairs. Nested braced-init-lists work.
std::map<int, string> m = {{1, "one"}, {2, "2"}};

// A braced-init-list can be implicitly converted to a return type.
std::vector<int> test_function() { return {1, 2, 3}; }

// Iterate over a braced-init-list.
for (int i : {-1, -2, -3}) {}

// Call a function using a braced-init-list.
void TestFunction2(std::vector<int> v) {}
TestFunction2({1, 2, 3});
```

A user-defined type can also define a constructor and/or assignment operator that take `std::initializer_list<T>`, which is automatically created from *braced-init-list*:

```
class MyType {
 public:
  // std::initializer_list references the underlying init list.
  // It should be passed by value.
  MyType(std::initializer_list<int> init_list) {
    for (int i : init_list) append(i);
  }
  MyType& operator=(std::initializer_list<int> init_list) {
    clear();
    for (int i : init_list) append(i);
  }
};
MyType m{2, 3, 5, 7};
```

Finally, brace initialization can also call ordinary constructors of data types, even if they do not have `std::initializer_list<T>` constructors.

```
double d{1.23};
// Calls ordinary constructor as long as MyOtherType has no
// std::initializer_list constructor.
class MyOtherType {
 public:
  explicit MyOtherType(string);
  MyOtherType(int, string);
};
MyOtherType m = {1, "b"};
// If the constructor is explicit, you can't use the "= {}" form.
MyOtherType m{"b"};
```

Never assign a *braced-init-list* to an auto local variable. In the single element case, what this means can be confusing.

Bad code

```
auto d = {1.23};        // d is a std::initializer_list<double>
```

Good code

```
auto d = double{1.23};  // Good -- d is a double, not a std::initializer_list.
```

See *Braced Initializer List Format* for formatting.

Lambda expressions

Use lambda expressions where appropriate. Prefer explicit captures when the lambda will escape the current scope.

Definition

Lambda expressions are a concise way of creating anonymous function objects. They're often useful when passing functions as arguments. For example:

```
std::sort(v.begin(), v.end(), [](int x, int y) {
  return Weight(x) < Weight(y);
});
```

They further allow capturing variables from the enclosing scope either explicitly by name, or implicitly using a default capture. Explicit captures require each variable to be listed, as either a value or reference capture:

```
int weight = 3;
int sum = 0;
// Captures `weight` by value and `sum` by reference.
std::for_each(v.begin(), v.end(), [weight, &sum](int x) {
  sum += weight * x;
});
```

Default captures implicitly capture any variable referenced in the lambda body, including `this` if any members are used:

```
const std::vector<int> lookup_table = ...;
std::vector<int> indices = ...;
// Captures `lookup_table` by reference, sorts `indices` by the value
// of the associated element in `lookup_table`.
std::sort(indices.begin(), indices.end(), [&](int a, int b) {
  return lookup_table[a] < lookup_table[b];
});
```

Lambdas were introduced in C++11 along with a set of utilities for working with function objects, such as the polymorphic wrapper `std::function`.

Pros

- Lambdas are much more concise than other ways of defining function objects to be passed to STL algorithms, which can be a readability improvement.
- Appropriate use of default captures can remove redundancy and highlight important exceptions from the default.
- Lambdas, `std::function`, and `std::bind` can be used in combination as a general purpose callback mechanism; they make it easy to write functions that take bound functions as arguments.

Cons

- Variable capture in lambdas can be a source of dangling-pointer bugs, particularly if a lambda escapes the current scope.
- Default captures by value can be misleading because they do not prevent dangling-pointer bugs. Capturing a pointer by value doesn't cause a deep copy, so it often has the same lifetime issues as capture by reference. This is especially confusing when capturing 'this' by value, since the use of 'this' is often implicit.
- It's possible for use of lambdas to get out of hand; very long nested anonymous functions can make code harder to understand.

Decision

- Use lambda expressions where appropriate, with formatting as described *below*.
- Prefer explicit captures if the lambda may escape the current scope. For example, instead of:

```
{
  Foo foo;
  ...
  executor->Schedule([&] { Frobnicate(foo); })
  ...
}
// BAD! The fact that the lambda makes use of a reference to `foo` and
// possibly `this` (if `Frobnicate` is a member function) may not be
// apparent on a cursory inspection. If the lambda is invoked after
// the function returns, that would be bad, because both `foo`
// and the enclosing object could have been destroyed.
```

  prefer to write:

```
{
  Foo foo;
  ...
  executor->Schedule([&foo] { Frobnicate(foo); })
  ...
}
// BETTER - The compile will fail if `Frobnicate` is a member
// function, and it's clearer that `foo` is dangerously captured by
// reference.
```

- Use default capture by reference (`[&]`) only when the lifetime of the lambda is obviously shorter than any potential captures.
- Use default capture by value (`[=]`) only as a means of binding a few variables for a short lambda, where the set of captured variables is obvious at a glance. Prefer not to write long or complex lambdas with default capture by value.
- Keep unnamed lambdas short. If a lambda body is more than maybe five lines long, prefer to give the lambda a name, or to use a named function instead of a lambda.
- Specify the return type of the lambda explicitly if that will make it more obvious to readers, as with *auto*.

Template metaprogramming

Avoid complicated template programming.

Definition

Template metaprogramming refers to a family of techniques that exploit the fact that the C++ template instantiation mechanism is Turing complete and can be used to perform arbitrary compile-time computation in the type domain.

Pros

Template metaprogramming allows extremely flexible interfaces that are type safe and high performance. Facilities like Google Test, `std::tuple`, `std::function`, and Boost.Spirit would be impossible without it.

Cons

The techniques used in template metaprogramming are often obscure to anyone but language experts. Code that uses templates in complicated ways is often unreadable, and is hard to debug or maintain.

Template metaprogramming often leads to extremely poor compiler time error messages: even if an interface is simple, the complicated implementation details become visible when the user does something wrong.

Template metaprogramming interferes with large scale refactoring by making the job of refactoring tools harder. First, the template code is expanded in multiple contexts, and it's hard to verify that the transformation makes sense in all of them. Second, some refactoring tools work with an AST that only represents the structure of the code after template expansion. It can be difficult to automatically work back to the original source construct that needs to be rewritten.

Decision

Template metaprogramming sometimes allows cleaner and easier-to-use interfaces than would be possible without it, but it's also often a temptation to be overly clever. It's best used in a small number of low level components where the extra maintenance burden is spread out over a large number of uses.

Think twice before using template metaprogramming or other complicated template techniques; think about whether the average member of your team will be able to understand your code well enough to maintain it after you switch to another project, or whether a non-C++ programmer or someone casually browsing the code base will be able to understand the error messages or trace the flow of a function they want to call. If you're using recursive template instantiations or type lists or metafunctions or expression templates, or relying on SFINAE or on the `sizeof` trick for detecting function overload resolution, then there's a good chance you've gone too far.

If you use template metaprogramming, you should expect to put considerable effort into minimizing and isolating the complexity. You should hide metaprogramming as an implementation detail whenever possible, so that user-facing headers are readable, and you should make sure that tricky code is especially well commented. You should carefully document how the code is used, and you should say something about what the "generated" code looks like. Pay extra attention to the error messages that the compiler emits when users make mistakes. The error messages are part of your user interface, and your code should be tweaked as necessary so that the error messages are understandable and actionable from a user point of view.

C++11

Use libraries and language extensions from C++11 when appropriate. Consider portability to other environments before using C++11 features in your project.

Definition

C++11 contains significant changes significant changes both to the language and libraries.

Pros

C++11 was the official standard until august 2014, and is supported by most C++ compilers. It standardizes some common C++ extensions that we use already, allows shorthands for some operations, and has some performance and safety improvements.

Cons

The C++11 standard is substantially more complex than its predecessor (1,300 pages versus 800 pages), and is unfamiliar to many developers. The long-term effects of some features on code readability and maintenance are unknown. We cannot predict when its various features will be implemented uniformly by tools that may be of interest, particularly in the case of projects that are forced to use older versions of tools.

Some C++11 extensions encourage coding practices that hamper readability - for example by removing checked redundancy (such as type names) that may be helpful to readers, or by encouraging template metaprogramming. Other extensions duplicate functionality available through existing mechanisms, which may lead to confusion and conversion costs.

Decision

C++11 features may be used unless specified otherwise. In addition to what's described in the rest of the style guide, the following C++11 features may not be used:

- Compile-time rational numbers (`<ratio>`), because of concerns that it's tied to a more template-heavy interface style.
- The `<cfenv>` and `<fenv.h>` headers, because many compilers do not support those features reliably.
- Ref-qualifiers on member functions, such as `void X::Foo() &` or `void X::Foo() &&`, because of concerns that they're an overly obscure feature.

Nonstandard Extensions

GCC Extensions to C++ may be used where they improve readability, performance or compiled code size.

Definition

Compilers support various extensions that are not part of standard C++. Such extensions include GCC's `__attribute__`, intrinsic functions such as `__builtin_prefetch`, designated initializers (e.g. `Foo f = {.field = 3}`), inline assembly, `__COUNTER__`, `__PRETTY_FUNCTION__`, compound statement expressions (e.g. `foo = ({ int x; Bar(&x); x })`, variable-length arrays and `alloca()`, and the `a?:b` syntax.

Pros

- Nonstandard extensions may provide useful features that do not exist in standard C++. For example, some people think that designated initializers are more readable than standard C++ features like constructors.
- Important performance guidance to the compiler can only be specified using extensions.

Cons

- Nonstandard extensions do not work in all compilers. Use of nonstandard extensions reduces portability of code.
- Even if they are supported in all targeted compilers, the extensions are often not well-specified, and there may be subtle behavior differences between compilers.
- Nonstandard extensions add to the language features that a reader must know to understand the code.

Decision

Use GCC extensions where necessary.

Aliases

Public aliases are for the benefit of an API's user, and should be clearly documented.

Definition

There are several ways to create names that are aliases of other entities:

```
typedef Foo Bar;
using Bar = Foo;
using other_namespace::Foo;
```

Like other declarations, aliases declared in a header file are part of that header's public API unless they're in a function definition, in the private portion of a class, or in an explicitly-marked internal namespace. Aliases in such areas or in .cc files are implementation details (because client code can't refer to them), and are not restricted by this rule.

Pros

- Aliases can improve readability by simplifying a long or complicated name.
- Aliases can reduce duplication by naming in one place a type used repeatedly in an API, which *might* make it easier to change the type later.

Cons

- When placed in a header where client code can refer to them, aliases increase the number of entities in that header's API, increasing its complexity.
- Clients can easily rely on unintended details of public aliases, making changes difficult.
- It can be tempting to create a public alias that is only intended for use in the implementation, without considering its impact on the API, or on maintainability.
- Aliases can create risk of name collisions
- Aliases can reduce readability by giving a familiar construct an unfamiliar name
- Type aliases can create an unclear API contract: it is unclear whether the alias is guaranteed to be identical to the type it aliases, to have the same API, or only to be usable in specified narrow ways

Decision

Don't put an alias in your public API just to save typing in the implementation; do so only if you intend it to be used by your clients.

When defining a public alias, document the intent of the new name, including whether it is guaranteed to always be the same as the type it's currently aliased to, or whether a more limited compatibility is intended. This lets the user know whether they can treat the types as substitutable or whether more specific rules must be followed, and can help the implementation retain some degree of freedom to change the alias.

Don't put namespace aliases in your public API. (See also Namespaces).

For example, these aliases document how they are intended to be used in client code:

```
namespace a {
// Used to store field measurements. DataPoint may change from Bar* to some internal␣
↪type.
// Client code should treat it as an opaque pointer.
using DataPoint = foo::bar::Bar*;

// A set of measurements. Just an alias for user convenience.
using TimeSeries = std::unordered_set<DataPoint, std::hash<DataPoint>,␣
```

(continues on next page)

(continued from previous page)

```
↪DataPointComparator>;
}  // namespace a
```

These aliases don't document intended use, and half of them aren't meant for client use:

```
namespace a {
// Bad: none of these say how they should be used.
using DataPoint = foo::bar::Bar*;
using std::unordered_set;  // Bad: just for local convenience
using std::hash;           // Bad: just for local convenience
typedef unordered_set<DataPoint, hash<DataPoint>, DataPointComparator> TimeSeries;
}  // namespace a
```

However, local convenience aliases are fine in function definitions, private sections of classes, explicitly marked internal namespaces, and in `.cpp` files:

```
// In a .cpp file
using std::unordered_set;
```

11.1.15 Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

General Naming Rules

Names should be descriptive; avoid abbreviation.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

```
int price_count_reader;    // No abbreviation.
int num_errors;            // "num" is a widespread convention.
int num_dns_connections;   // Most people know what "DNS" stands for.
```

Bad code

```
int n;                     // Meaningless.
int nerr;                  // Ambiguous abbreviation.
int n_comp_conns;          // Ambiguous abbreviation.
int wgc_connections;       // Only your group knows what this stands for.
int pc_reader;             // Lots of things can be abbreviated "pc".
int cstmr_id;              // Deletes internal letters.
```

Note that certain universally-known abbreviations are OK, such as `i` for an iteration variable and `T` for a template parameter.

Template parameters should follow the naming style for their category: type template parameters should follow the rules for *type names*, and non-type template parameters should follow the rules for *variable names*.

File Names

> Filenames should be all lowercase and can include underscores (_) or dashes (-). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "_".

Examples of acceptable file names:

- `my_useful_class.cpp`
- `my-useful-class.cpp`
- `myusefullclass.cpp`
- `myusefulclass_test.cpp`

C++ files should end in `.cpp` and header files should end in `.h`. Files that rely on being textually included at specific points should end in `.inc` (see also the section on *self-contained headers*).

Do not use filenames that already exist in `/usr/include`, such as `db.h`.

In general, make your filenames very specific. For example, use `http_server_logs.h` rather than `logs.h`. A very common case is to have a pair of files called, e.g., `foo_bar.h` and `foo_bar.cpp`, defining a class called `FooBar`.

Inline functions must be in a `.h` file. If your inline functions are very short, they should go directly into your `.h` file.

Type Names

> Type names start with a capital letter and have a capital letter for each new word, with no underscores: `MyExcitingClass`, `MyExcitingEnum`.

The names of all types - classes, structs, type aliases, enums, and type template parameters - have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores. For example:

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// using aliases
using PropertiesMap = hash_map<UrlTableProperties *, string>;

// enums
enum UrlTableErrors { ...
```

Variable Names

The names of variables (including function parameters) and data members are all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores. For instance: `a_local_variable`, `a_struct_data_member`, `a_class_data_member_`.

Common Variable Names

For example:

```
string table_name;  // OK - uses underscore.
string tablename;   // OK - all lowercase.
string tableName;   // Bad - mixed case.
```

Class Data Members

Data members of classes, both static and non-static, are named like ordinary nonmember variables, but with a trailing underscore.

```
class TableInfo {
  ...
 private:
  string table_name_;  // OK - underscore at end.
  string tablename_;   // OK.
  static Pool<TableInfo>* pool_;  // OK.
};
```

Struct Data Members

Data members of structs, both static and non-static, are named like ordinary nonmember variables. They do not have the trailing underscores that data members in classes have.

```
struct UrlTableProperties {
  string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;
};
```

See *Structs vs. Classes* for a discussion of when to use a struct versus a class.

Constant Names

Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case. For example:

```
const int kDaysInAWeek = 7;
```

All such variables with static storage duration (i.e. statics and globals, see Storage Duration for details) should be named this way. This convention is optional for variables of other storage classes, e.g. automatic variables, otherwise the usual variable naming rules apply.

Function Names

> Regular functions have mixed case

Ordinarily, functions should start with a lower-case letter and have a capital letter for each new word (a.k.a. "Camel Case" or "Pascal case"). Such names should not have underscores. Prefer to capitalize acronyms as single words (i.e. `startRpc()`, not `startRPC()`).

```
addTableEntry()
deleteUrl()
openFileOrDie()
```

(The same naming rule applies to class- and namespace-scope constants that are exposed as part of an API and that are intended to look like functions, because the fact that they're objects rather than functions is an unimportant implementation detail.)

Namespace Names

> Namespace names are all lower-case. Top-level namespace names are based on the project name . Avoid collisions between nested namespaces and well-known top-level namespaces.

The name of a top-level namespace should usually be the name of the project or team whose code is contained in that namespace. The code in that namespace should usually be in a directory whose basename matches the namespace name (or subdirectories thereof).

Keep in mind that the *rule against abbreviated names* applies to namespaces just as much as variable names. Code inside the namespace seldom needs to mention the namespace name, so there's usually no particular need for abbreviation anyway.

Avoid nested namespaces that match well-known top-level namespaces. Collisions between namespace names can lead to surprising build breaks because of name lookup rules. In particular, do not create any nested `std` namespaces. Prefer unique project identifiers (`websearch::index`, `websearch::index_util`) over collision-prone names like `websearch::util`.

For `internal` namespaces, be wary of other code being added to the same `internal` namespace causing a collision (internal helpers within a team tend to be related and may lead to collisions). In such a situation, using the filename to make a unique internal name is helpful (`websearch::index::frobber_internal` for use in `frobber.h`)

Enumerator Names

> Enumerators (for both scoped and unscoped enums) should be named *either* like *constants* or like *macros*: either `kEnumName` or `ENUM_NAME`.

Preferably, the individual enumerators should be named like *constants*. However, it is also acceptable to name them like *macros*. The enumeration name, `UrlTableErrors` (and `AlternateUrlTableErrors`), is a type, and therefore mixed case.

```
enum UrlTableErrors {
  kOK = 0,
  kErrorOutOfMemory,
  kErrorMalformedInput,
};
enum AlternateUrlTableErrors {
  OK = 0,
  OUT_OF_MEMORY = 1,
```

(continues on next page)

(continued from previous page)

```
  MALFORMED_INPUT = 2,
};
```

Macro Names

> You're not really going to [define a macro](#preprocessor-macros, are you? If you do, they're like this: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN`.

Please see the *description of macros*; in general macros should *not* be used. However, if they are absolutely needed, then they should be named with all capitals and underscores.

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

Exceptions to Naming Rules

> If you are naming something that is analogous to an existing C or C++ entity then you can follow the existing naming convention scheme.

11.1.16 Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous - the next one may be you!

Comment Style

> Use either the `//` or `/* */` syntax, as long as you are consistent.

You can use either the `//` or the `/* */` syntax; however, `//` is *much* more common. Be consistent with how you comment and what style you use where.

File Comments

> Start each file with license boilerplate.

File comments describe the contents of a file. If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, file comments are not required. All other files must have file comments.

Legal Notice and Author Line

Every file should contain license boilerplate. Choose the appropriate boilerplate for the license used by the project (for example, Apache 2.0, BSD, LGPL, GPL).

If you make significant changes to a file with an author line, consider deleting the author line.

File Contents

If a `.h` declares multiple abstractions, the file-level comment should broadly describe the contents of the file, and how the abstractions are related. A 1 or 2 sentence file-level comment may be sufficient. The detailed documentation about individual abstractions belongs with those abstractions, not at the file level.

Do not duplicate comments in both the `.h` and the `.cpp`. Duplicated comments diverge.

Class Comments

> Every non-obvious class declaration should have an accompanying comment that describes what it is for and how it should be used.

```
// Iterates over the contents of a GargantuanTable.
// Example:
//    GargantuanTableIterator* iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
//    delete iter;
class GargantuanTableIterator {
  ...
};
```

The class comment should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class. Document the synchronization assumptions the class makes, if any. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

The class comment is often a good place for a small example code snippet demonstrating a simple and focused usage of the class.

When sufficiently separated (e.g. `.h` and `.cpp` files), comments describing the use of the class should go together with its interface definition; comments about the class operation and implementation should accompany the implementation of the class's methods.

Function Comments

> Declaration comments describe use of the function (when it is non-obvious); comments at the definition of a function describe operation.

Function Declarations

Almost every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments may be omitted only if the function is simple and obvious (e.g. simple accessors for obvious properties of the class). These comments should be descriptive (”Opens the file”) rather than imperative (”Open the file”); the comment describes the function, it does not tell the function what to do. In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

- What the inputs and outputs are.
- For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.
- If the function allocates memory that the caller must free.
- Whether any of the arguments can be a null pointer.
- If there are any performance implications of how a function is used.
- If the function is re-entrant. What are its synchronization assumptions?

Here is an example:

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

However, do not be unnecessarily verbose or state the completely obvious. Notice below that it is not necessary to say “returns false otherwise” because this is implied.

```
// Returns true if the table cannot hold any more entries.
bool IsTableFull();
```

When documenting function overrides, focus on the specifics of the override itself, rather than repeating the comment from the overridden function. In many of these cases, the override needs no additional documentation and thus no comment is required.

When commenting constructors and destructors, remember that the person reading your code knows what constructors and destructors are for, so comments that just say something like “destroys this object” are not useful. Document what constructors do with their arguments (for example, if they take ownership of pointers), and what cleanup the destructor does. If this is trivial, just skip the comment. It is quite common for destructors not to have a header comment.

Function Definitions

If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

Note you should *not* just repeat the comments given with the function declaration, in the `.h` file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

Variable Comments

In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required.

Class Data Member Comments

The purpose of each class data member (also called an instance variable or member variable) must be clear. If there are any invariants (special values, relationships between members, lifetime requirements) not clearly expressed by the type and name, they must be commented. However, if the type and name suffice (`int num_events_;`), no comment is needed.

In particular, add comments to describe the existence and meaning of sentinel values, such as nullptr or -1, when they are not obvious. For example:

```
private:
 // Used to bounds-check table accesses. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

Global Variable Comments

All global variables should have a comment describing what they are, what they are used for, and (if unclear) why it needs to be global. For example:

```
// The total number of tests cases that we run through in this regression test.
const int kNumTestCases = 6;
```

Implementation Comments

In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.

Explanatory Comments

Tricky or complicated code blocks should have comments before them. Example:

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

Line Comments

Also, lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code by 2 spaces. Example:

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  // Error already logged.
```

Note that there are both comments that describe what the code is doing, and comments that mention that an error has already been logged when the function returns.

If you have several comments on subsequent lines, it can often be more readable to line them up:

```
DoSomething();                  // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Two spaces between the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse();  // Two spaces before line comments normally.
}
std::vector<string> list{
                    // Comments in braced lists describe the next element...
                    "First item",
                    // .. and should be aligned appropriately.
                    "Second item"};
DoSomething(); /* For trailing block comments, one space is fine. */
```

Function Argument Comments

When the meaning of a function argument is nonobvious, consider one of the following remedies:

- If the argument is a literal constant, and the same constant is used in multiple function calls in a way that tacitly assumes they're the same, you should use a named constant to make that constraint explicit, and to guarantee that it holds.
- Consider changing the function signature to replace a `bool` argument with an `enum` argument. This will make the argument values self-describing.
- For functions that have several configuration options, consider defining a single class or struct to hold all the options , and pass an instance of that. This approach has several advantages. Options are referenced by name at

the call site, which clarifies their meaning. It also reduces function argument count, which makes function calls easier to read and write. As an added benefit, you don't have to change call sites when you add another option.

- Replace large or complex nested expressions with named variables.
- As a last resort, use comments to clarify argument meanings at the call site.

Consider the following example:

```
// What are these arguments?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

versus:

```
ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

Don'ts

Do not state the obvious. In particular, don't literally describe what code does, unless the behavior is nonobvious to a reader who understands C++ well. Instead, provide higher level comments that describe *why* the code does what it does, or make the code self describing.

Compare this:

```
// Find the element in the vector.  <-- Bad: obvious!
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

To this:

```
// Process "element" unless it was already processed.
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

Self-describing code doesn't need a comment. The comment from the example above would be obvious:

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

Punctuation, Spelling and Grammar

> Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

TODO Comments

> Use `TODO` comments for code that is temporary, a short-term solution, or good-enough but not perfect.

`TODO`s should include the string `TODO` in all caps, followed by the name, e-mail address, bug ID, or other identifier of the person or issue with the best context about the problem referenced by the `TODO`. The main purpose is to have a consistent `TODO` that can be searched to find out how to get more details upon request. A `TODO` is not a commitment that the person referenced will fix the problem. Thus when you create a `TODO` with a name, it is almost always your name that is given.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(bug 12345): remove the "Last visitors" feature
```

If your `TODO` is of the form “At a future date do something” make sure that you either include a very specific date (”Fix by November 2005”) or a very specific event (”Remove this code when all clients can handle XML responses.”).

Deprecation Comments

> Mark deprecated interface points with `DEPRECATED` comments.

You can mark an interface as deprecated by writing a comment containing the word `DEPRECATED` in all caps. The comment goes either before the declaration of the interface or on the same line as the declaration.

After the word `DEPRECATED`, write your name, e-mail address, or other identifier in parentheses.

A deprecation comment must include simple, clear directions for people to fix their callsites. In C++, you can implement a deprecated function as an inline function that calls the new interface point.

Marking an interface point `DEPRECATED` will not magically cause any callsites to change. If you want people to actually stop using the deprecated facility, you will have to fix the callsites yourself or recruit a crew to help you.

New code should not contain calls to deprecated interface points. Use the new interface point instead. If you cannot understand the directions, find the person who created the deprecation and ask them for help using the new interface point.

11.1.17 Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

To help format code in compliance with this style guide, we use `clang-format`, which many editors can be configured to call automatically. There is also `make format` target available in the Kaleidoscope Makefile that will use `clang-format` to format all the core and plugin code. Our CI infrastructure checks to ensure that code has been formatted to these specifications.

Line Length

> Each line of text in your code should be at most 80 characters long.

We recognize that this rule is controversial, but so much existing code already adheres to it, and we feel that consistency is important.

Pros

Those who favor this rule argue that it is rude to force them to resize their windows and there is no need for anything longer. Some folks are used to having several code windows side-by-side, and thus don't have room to widen their windows in any case. People set up their work environment assuming a particular maximum window width, and 80 columns has been the traditional standard. Why change it?

Cons

Proponents of change argue that a wider line can make code more readable. The 80-column limit is an hidebound throwback to 1960s mainframes; modern equipment has wide screens that can easily show longer lines.

Decision

80 characters is the maximum.

Exception

Comment lines can be longer than 80 characters if it is not feasible to split them without harming readability, ease of cut and paste or auto-linking – e.g. if a line contains an example command or a literal URL longer than 80 characters.

Exception

A raw-string literal may have content that exceeds 80 characters. Except for test code, such literals should appear near the top of a file.

Exception

An `#include` statement with a long path may exceed 80 columns.

Non-ASCII Characters

> Non-ASCII characters should be rare, and must use UTF-8 formatting.

You shouldn't hard-code user-facing text in source, even English, so use of non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in your code. For example, if your code parses data files from foreign sources, it may be appropriate to hard-code the non-ASCII string(s) used in those data files as delimiters. More commonly, unittest code (which does not need to be localized) might contain non-ASCII strings. In such cases, you should use UTF-8, since that is an encoding understood by most tools able to handle more than just ASCII.

Hex encoding is also OK, and encouraged where it enhances readability - for example, `"\xEF\xBB\xBF"`, or, even more simply, `u8"\uFEFF"`, is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

Use the `u8` prefix to guarantee that a string literal containing `\uXXXX` escape sequences is encoded as UTF-8. Do not use it for strings containing non-ASCII characters encoded as UTF-8, because that will produce incorrect output if the compiler does not interpret the source file as UTF-8.

You shouldn't use the C++11 `char16_t` and `char32_t` character types, since they're for non-UTF-8 text. For similar reasons you also shouldn't use `wchar_t` (unless you're writing code that interacts with the Windows API, which uses `wchar_t` extensively).

Spaces vs. Tabs

> Use only spaces, and indent 2 spaces at a time.

We use spaces for indentation. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key.

Function Declarations and Definitions

> Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a *function call*.

Functions look like this:

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

If you have too much text to fit on one line:

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

or if you cannot fit even the first parameter:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

Some points to note:

- Choose good parameter names.
- Parameter names may be omitted only if the parameter is unused and its purpose is obvious.
- If you cannot fit the return type and the function name on a single line, break between them.

- If you break after the return type of a function declaration or definition, do not indent.
- The open parenthesis is always on the same line as the function name.
- There is never a space between the function name and the open parenthesis.
- There is never a space between the parentheses and the parameters.
- The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
- The close curly brace is either on the last line by itself or on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be aligned if possible.
- Default indentation is 2 spaces.
- Wrapped parameters have a 4 space indent.

Unused parameters that are obvious from context may be omitted:

```
class Foo {
 public:
  Foo(Foo&&);
  Foo(const Foo&);
  Foo& operator=(Foo&&);
  Foo& operator=(const Foo&);
};
```

Unused parameters that might not be obvious should comment out the variable name in the function definition:

```
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
};

class Circle : public Shape {
 public:
  void Rotate(double radians) override;
};

void Circle::Rotate(double /*radians*/) {}
```

```
// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::Rotate(double) {}
```

Attributes, and macros that expand to attributes, appear at the very beginning of the function declaration or definition, before the return type:

```
MUST_USE_RESULT bool IsOK();
```

Formatting Lambda Expressions

Format parameters and bodies as for any other function, and capture lists like other comma-separated lists.

For by-reference captures, do not leave a space between the ampersand (&) and the variable name.

```
int x = 0;
auto x_plus_n = [&x](int n) -> int { return x + n; }
```

Short lambdas may be written inline as function arguments.

```
std::set<int> skip_list = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&skip_list](int i) {
               return skip_list.find(i) != skip_list.end();
             }),
             digits.end());
```

Function Calls

Either write the call all on a single line, wrap the arguments at the parenthesis, or start the arguments on a new line indented by four spaces and continue at that 4 space indent. In the absence of other considerations, use the minimum number of lines, including placing multiple arguments on each line where appropriate.

Function calls have the following format:

```
bool result = DoSomething(argument1, argument2, argument3);
```

If the arguments do not all fit on one line, they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. Do not add spaces after the open paren or before the close paren:

```
bool result = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

Arguments may optionally all be placed on subsequent lines with a four space indent:

```
if (...) {
  ...
  ...
  if (...) {
    bool result = DoSomething(
        argument1, argument2,  // 4 space indent
        argument3, argument4);
    ...
  }
```

Put multiple arguments on a single line to reduce the number of lines necessary for calling a function unless there is a specific readability problem. Some find that formatting with strictly one argument on each line is more readable and simplifies editing of the arguments. However, we prioritize for the reader over the ease of editing arguments, and most readability problems are better addressed with the following techniques.

If having multiple arguments in a single line decreases readability due to the complexity or confusing nature of the expressions that make up some arguments, try creating variables that capture those arguments in a descriptive name:

```
int my_heuristic = scores[x] * y + bases[x];
bool result = DoSomething(my_heuristic, x, y, z);
```

Or put the confusing argument on its own line with an explanatory comment:

```
bool result = DoSomething(scores[x] * y + bases[x],  // Score heuristic.
                          x, y, z);
```

If there is still a case where one argument is significantly more readable on its own line, then put it on its own line. The decision should be specific to the argument which is made more readable rather than a general policy.

Sometimes arguments form a structure that is important for readability. In those cases, feel free to format the arguments according to that structure:

```
// Transform the widget by a 3x3 matrix.
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

Braced Initializer List Format

Format a *braced initializer list* exactly like you would format a function call in its place.

If the braced list follows a name (e.g. a type or variable name), format as if the {} were the parentheses of a function call with that name. If there is no name, assume a zero-length name.

```
// Examples of braced init list on a single line.
return {foo, bar};
functioncall({foo, bar});
std::pair<int, int> p{foo, bar};

// When you have to wrap.
SomeFunction(
    {"assume a zero-length name before {"},
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",
        some, other values},
    SomeOtherType{"Slightly shorter string",
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};
MyType m = {  // Here, you could also break before {.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

Conditionals

Prefer no spaces inside parentheses. The `if` and `else` keywords belong on separate lines.

There are two acceptable formats for a basic conditional statement. One includes spaces between the parentheses and the condition, and one does not.

The most common form is without spaces. Either is fine, but *be consistent*. If you are modifying a file, use the format that is already present. If you are writing new code, use the format that the other files in that directory or project use. If in doubt and you have no personal preference, do not add the spaces.

```
if (condition) {  // no spaces inside parentheses
  ...  // 2 space indent.
} else if (...) {  // The else goes on the same line as the closing brace.
  ...
} else {
  ...
}
```

If you prefer you may add spaces inside the parentheses:

```
if ( condition ) {  // spaces inside parentheses - rare
  ...  // 2 space indent.
} else {  // The else goes on the same line as the closing brace.
  ...
}
```

Note that in all cases you must have a space between the `if` and the open parenthesis. You must also have a space between the close parenthesis and the curly brace, if you're using one.

```
if(condition) {   // Bad - space missing after IF.
if (condition){   // Bad - space missing before {.
if(condition){    // Doubly bad.

if (condition) {  // Good - proper space after IF and before {.
```

Short conditional statements may be written on one line if this enhances readability. You may use this only when the line is brief and the statement does not use the `else` clause.

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

This is not allowed when the if statement has an `else`:

```
// Not allowed - IF statement on one line when there is an ELSE clause
if (x) DoThis();
else DoThat();
```

In general, curly braces are not required for single-line statements, but they are allowed if you like them; conditional or loop statements with complex conditions or statements may be more readable with curly braces. Some projects require that an `if` must always always have an accompanying brace.

```
if (condition)
  DoSomething();  // 2 space indent.

```

(continues on next page)

(continued from previous page)

```
if (condition) {
  DoSomething();  // 2 space indent.
}
```

However, if one part of an `if-else` statement uses curly braces, the other part must too:

```
// Not allowed - curly on IF but not ELSE
if (condition) {
  foo;
} else
  bar;

// Not allowed - curly on ELSE but not IF
if (condition)
  foo;
else {
  bar;
}
```

```
// Curly braces around both IF and ELSE required because
// one of the clauses used braces.
if (condition) {
  foo;
} else {
  bar;
}
```

Loops and Switch Statements

> Switch statements may use braces for blocks. Annotate non-trivial fall-through between cases. Braces are optional for single-statement loops. Empty loop bodies should use empty braces or `continue`.

`case` blocks in `switch` statements can have curly braces or not, depending on your preference. If you do include curly braces they should be placed as shown below.

If not conditional on an enumerated value, switch statements should always have a `default` case (in the case of an enumerated value, the compiler will warn you if any values are not handled). If the default case should never execute, simply `assert`:

```
switch (var) {
  case 0: {  // 2 space indent
    ...      // 4 space indent
    break;
  }
  case 1: {
    ...
    break;
  }
  default: {
    assert(false);
  }
}
```

Braces are optional for single-statement loops.

```
for (int i = 0; i < kSomeNumber; ++i)
  printf("I love you\n");

for (int i = 0; i < kSomeNumber; ++i) {
  printf("I take it back\n");
}
```

Empty loop bodies should use an empty pair of braces or `continue`, but not a single semicolon.

```
while (condition) {
  // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - one newline is also OK.
while (condition) continue;  // Good - continue indicates no logic.
```

```
while (condition);  // Bad - looks like part of do/while loop.
```

Pointer and Reference Expressions

No spaces around period or arrow. Pointer operators do not have trailing spaces.

The following are examples of correctly-formatted pointer and reference expressions:

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

Note that:

- There are no spaces around the period or arrow when accessing a member.
- Pointer operators have no space after the * or &.

When declaring a pointer variable or argument, you may place the asterisk adjacent to either the type or to the variable name:

```
// These are fine, space preceding.
char *c;
const string &str;

// These are fine, space following.
char* c;
const string& str;
```

It is allowed (if unusual) to declare multiple variables in the same declaration, but it is disallowed if any of those have pointer or reference decorations. Such declarations are easily misread.

```
// Fine if helpful for readability.
int x, y;
int x, *y;  // Disallowed - no & or * in multiple declaration
char * c;  // Bad - spaces on both sides of *
const string & str;  // Bad - spaces on both sides of &
```

You should do this consistently within a single file, so, when modifying an existing file, use the style in that file.

Boolean Expressions

When you have a boolean expression that is longer than the *standard line length* , be consistent in how you break up the lines.

In this example, the logical AND operator is always at the end of the lines:

```
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another && last_one) {
  ...
}
```

Note that when the code wraps in this example, both of the && logical AND operators are at the end of the line. This is more common in Google code, though wrapping all operators at the beginning of the line is also allowed. Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately. Also note that you should always use the punctuation operators, such as && and ~ , rather than the word operators, such as `and` and `compl`.

Return Values

Do not needlessly surround the `return` expression with parentheses.

Use parentheses in `return expr;` only where you would use them in `x = expr;`.

```
return result;                  // No parentheses in the simple case.
// Parentheses OK to make a complex expression more readable.
return (some_long_condition &&
        another_condition);
```

```
return (value);                // You wouldn't write var = (value);
return(result);                // return is not a function!
```

Variable and Array Initialization

Your choice of =, (), or {}.

You may choose between =, (), or {}; the following are all correct:

```
int x = 3;
int x(3);
int x{3};
string name = "Some Name";
string name("Some Name");
string name{"Some Name"};
```

Be careful when using a braced initialization list `{...}` on a type with an `std::initializer_list` constructor. A nonempty *braced-init-list* prefers the `std::initializer_list` constructor whenever possible. Note that empty braces `{}` are special, and will call a default constructor if available. To force the non-`std::initializer_list` constructor, use parentheses instead of braces.

```
std::vector<int> v(100, 1);  // A vector of 100 1s.
std::vector<int> v{100, 1};  // A vector of 100, 1.
```

Also, the brace form prevents narrowing of integral types. This can prevent some types of programming errors.

```
int pi(3.14);  // OK -- pi == 3.
int pi{3.14};  // Compile error: narrowing conversion.
```

Preprocessor Directives

The hash mark that starts a preprocessor directive should always be at the beginning of the line.

Even when preprocessor directives are within the body of indented code, the directives should start at the beginning of the line.

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning of line
    DropEverything();
# if NOTIFY               // OK but not required -- Spaces after #
    NotifyClient();
# endif
#endif
    BackToNormal();
  }
```

```
// Bad - indented directives
  if (lopsided_score) {
    #if DISASTER_PENDING  // Wrong!  The "#if" should be at beginning of line
    DropEverything();
    #endif                // Wrong!  Do not indent "#endif"
    BackToNormal();
  }
```

Class Format

Sections in `public`, `protected` and `private` order, each indented one space.

The basic format for a class definition (lacking the comments, see *class comments* for a discussion of what comments are needed) is:

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit MyClass(int var);
  ~MyClass() {}
```

(continues on next page)

(continued from previous page)

```
  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
};
```

Things to note:

- Any base class name should be on the same line as the subclass name, subject to the 80-column limit.
- The `public:`, `protected:`, and `private:` keywords should be indented one space.
- Except for the first instance, these keywords should be preceded by a blank line. This rule is optional in small classes.
- Do not leave a blank line after these keywords.
- The `public` section should be first, followed by the `protected` and finally the `private` section.
- See *declaration order* for rules on ordering declarations within each of these sections.

Constructor Initializer Lists

Constructor initializer lists can be all on one line or with subsequent lines indented four spaces.

The acceptable formats for initializer lists are:

```
// When everything fits on one line:
MyClass::MyClass(int var) : some_var_(var) {
  DoSomething();
}

// If the signature and initializer list are not all on one line,
// you must wrap before the colon and indent 4 spaces:
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}

// When the list spans multiple lines, put each member on its own line
// and align them:
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  DoSomething();
}
```

(continues on next page)

(continued from previous page)

```
// As with any other code block, the close curly can be on the same
// line as the open curly, if it fits.
MyClass::MyClass(int var)
    : some_var_(var) {}
```

Namespace Formatting

The contents of namespaces are not indented.

Namespaces do not add an extra level of indentation. For example, use:

```
namespace {

void foo() {  // Correct.  No extra indentation within namespace.
  ...
}

}  // namespace
```

Do not indent within a namespace:

```
namespace {

  // Wrong.  Indented when it should not be.
  void foo() {
    ...
  }

}  // namespace
```

When declaring nested namespaces, put each namespace on its own line.

```
namespace foo {
namespace bar {
```

Horizontal Whitespace

Use of horizontal whitespace depends on location. Never put trailing whitespace at the end of a line.

General

```
void f(bool b) {  // Open braces should always have a space before them.
  ...
int i = 0;  // Semicolons usually have no space before them.
// Spaces inside braces for braced-init-list are optional.  If you use them,
// put them on both sides!
int x[] = { 0 };
int x[] = {0};

```

(continues on next page)

(continued from previous page)

```
// Spaces around the colon in inheritance and initializer lists.
class Foo : public Bar {
 public:
  // For inline function implementations, put spaces between the braces
  // and the implementation itself.
  Foo(int b) : Bar(), baz_(b) {}  // No spaces inside empty braces.
  void Reset() { baz_ = 0; }  // Spaces separating braces from implementation.
  ...
```

Adding trailing whitespace can cause extra work for others editing the same file, when they merge, as can removing existing trailing whitespace. So: Don't introduce trailing whitespace. Remove it if you're already changing that line, or do it in a separate clean-up operation (preferably when no-one else is working on the file).

Loops and Conditionals

```
if (b) {          // Space after the keyword in conditions and loops.
} else {          // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
// Loops and conditions may have spaces inside parentheses, but this
// is rare.  Be consistent.
switch ( i ) {
if ( test ) {
for ( int i = 0; i < 5; ++i ) {
// For loops always have a space after the semicolon.  They may have a space
// before the semicolon, but this is rare.
for ( ; i < 5 ; ++i) {
  ...

// Range-based for loops always have a space before and after the colon.
for (auto x : counts) {
  ...
}
switch (i) {
  case 1:         // No space before colon in a switch case.
    ...
  case 2: break;  // Use a space after a colon if there's code after it.
```

Operators

```
// Assignment operators always have spaces around them.
x = 0;

// Other binary operators usually have spaces around them, but it's
// OK to remove spaces around factors.  Parentheses should have no
// internal padding.
v = w * x + y / z;
```

(continues on next page)

(continued from previous page)

```
v = w*x + y/z;
v = w * (x + z);

// No spaces separating unary operators and their arguments.
x = -5;
++x;
if (x && !y)
  ...
```

Templates and Casts

```
// No spaces inside the angle brackets (< and >), before
// <, or between >( in a cast
std::vector<string> x;
y = static_cast<char*>(x);

// Spaces between type and pointer are OK, but be consistent.
std::vector<char *> x;
```

Vertical Whitespace

Minimize use of vertical whitespace.

This is more a principle than a rule: don't use blank lines when you don't have to. In particular, don't put more than one or two blank lines between functions, resist starting functions with a blank line, don't end functions with a blank line, and be discriminating with your use of blank lines inside functions.

The basic principle is: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Of course, readability can suffer from code being too dense as well as too spread out, so use your judgement. But in general, minimize use of vertical whitespace.

Some rules of thumb to help when blank lines may be useful:

- Blank lines at the beginning or end of a function very rarely help readability.
- Blank lines inside a chain of if-else blocks may well help readability.

11.1.18 Exceptions to the Rules

The coding conventions described above are mandatory. However, like all good rules, these sometimes have exceptions, which we discuss here.

Existing Non-conformant Code

You may diverge from the rules when dealing with code that does not conform to this style guide.

If you find yourself modifying code that was written to specifications other than those presented by this guide, you may have to diverge from these rules in order to stay consistent with the local conventions in that code. If you are in doubt about how to do this, ask the original author or the person currently responsible for the code. Remember that *consistency* includes local consistency, too.

11.1.19 Maintenance Tools

Kaleidoscope uses some automated tools to enforce compliance with this code style guide. Primarily, we use `clang-format` to format source files, `cpplint` to check for potential problems, and `include-what-you-use` to update header includes. These are invoked using python scripts that supply all of the necessary command-line parameters to both utilities. For convenience, there are also some shell scripts and makefile targets that further simplify the process of running these utilities properly.

Code Formatting

We use `clang-format`(version 12 or higher) to automatically format Kaleidoscope source files. There is a top-level `.clang-format` config file that contains the settings that best match the style described in this guide. However, there are some files in the repository that are, for one reason or another, exempt from formatting in this way, so we use a wrapper script, `format-code.py`, instead of invoking it directly.

`format-code.py` takes a list of target filenames, either as command-line parameters or from standard input (if reading from a pipe, with each line treated as a filename), and formats those files. If given a directory target, it will recursively search that directory for source files, and format all of the ones it finds.

By default, `format-code.py` will first check for unstaged changes in the Kaleidoscope git working tree, and exit before formatting source files if any are found. This is meant to make it easier for developers to see the changes made by the formatter in isolation. It can be given the `--force` option to skip this check.

It also has a `--check` option, which will cause `format-code.py` to check for unstaged git working tree changes after running `clang-format` on the target source files, and return an error code if there are any, allowing us to automatically verify that code submitted complies with the formatting rules.

The easiest way to invoke the formatter on the whole repository is by running `make format` at the top level of the Kaleidoscope repository.

For automated checking of PRs in a CI tool, there is also `make check-code-style`.

Linting

We use a copy of `cpplint.py` (with a modification that allows us to set the config file) to check for potential problems in the code. This can be invoked by running `make cpplint` at the top level of the Kaleidoscope repository.

Header Includes

We use `include-what-you-use` (version 18 or higher) to automatically manage header includes in Kaleidoscope source files. Because of the peculiarities of Kaleidoscope's build system, we use a wrapper script, `iwyu.py`, instead of invoking it directly.

`iwyu.py` takes a list of target filenames, either as command-line parameters or from standard input (if reading from a pipe, with each line treated as a filename), and makes changes to the header includes. If given a directory target, it will recursively search that directory for source files, and run `include-what-you-use` on all of the ones it finds.

A number of files can't be processed this way, and are enumerated in `.iwyu_ignore`. Files in the `testing` directory (for the test simulator) require different include path items, so cannot be combined in the same call to `iwyu.py`.

The easiest way to invoke `iwyu.py` is by running `make check-includes`, which will check all files that differ between the git working tree and the current master branch of the main Kaleidoscope repository, update their headers, and return a non-zero exit code if there were any errors processing the file(s) or any changes made.

For automated checking of header includes in a CI tool, there is also `make check-all-includes`, that checks the whole repository, not just the current branch's changes.

11.1.20 Parting Words

Use common sense and *BE CONSISTENT*.

If you are editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their `if` clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying, rather than on how you are saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a file looks drastically different from the existing code around it, the discontinuity throws readers out of their rhythm when they go to read it. Try to avoid this.

OK, enough writing about writing code; the code itself is much more interesting. Have fun!

11.2 Design philosophy

Kaleidoscope should, in order:

1. Work well as a keyboard
2. Be compatible with real hardware
3. Be compatible with the spec
4. Be easy to read and understand
5. Be easy to modify and customize

Our code indentation style is managed with 'make astyle.' For code we 'own', there should be an astyle target in the Makefile. For third party code we use and expect to update (ever), we try not to mess with the upstream house style.

11.3 Docker

It's possible to use Docker to run Kaleidoscope's test suite.

11.3.1 Running tests in Docker

```
# make docker-simulator-tests
```

11.3.2 Cleaning out stale data in the Docker image:

```
# make docker-clean
```

11.3.3 Removing the Kaleidoscope Docker image entirely:

```
# docker volume rm kaleidoscope-persist
# docker volume rm kaleidoscope-googletest-build
# docker volume rm kaleidoscope-build
# docker image rm kaleidoscope/docker
```

11.4 Glossary

This document is intended to name and describe the concepts, functions and data structures inside Kaleidoscope.

It is, as yet, incredibly incomplete.

When describing an identifier of any kind from the codebase, it should be written using identical capitalization to its use in the code and surrounded by backticks: `identifierName`

11.4.1 Firmware Terminology

These terms commonly arise when discussing the firmware.

Keyswitch

A single physical input, such as a keyswitch or other input like a knob or a slider

Key number

An integer representing a Keyswitch's position in the "Physical Layout". Represented in the code by the `KeyAddr` type.

Physical Layout

A mapping of keyswitches to key numbers

Key binding

A mapping from a key number to a behavior.

`Key`

A representation of a specific behavior. Most often a representation of a specific entry in the USB HID Usage Tables.

Keymap

A list of key bindings for all keyswitchess on the Physical Layout. Represented in the code by the `KeyMap` type.

Keymaps

An ordered list of all the Keymaps installed on a keyboard.

Layer

An entry in that ordered list of keymaps. Each layer has a unique id number that does not change. Layer numbers start at 0.

Active Layer Stack

An ordered list of all the currently-active layers, in the order they should be evaluated when figuring out what a key does.

Live keys

A representation of the current state of the keyboard's keys, where non-transparent entries indicate keys that are active (logically—usually, but not necessarily, physically held). Represented in the code by the `LiveKeys` type (and the `live_keys` object).

Active/inactive keys

In the `live_keys[]` array, an *active* key usually corresponds to a keyswitch that is physically pressed. In the common case of HID Keyboard keys, an active key will result in one or more keycodes being inserted in any new HID report. In some cases, an key can be active when its physical keyswitch is not pressed (e.g. OneShot keys that have been tapped), and in other cases a key might be *inactive* even though its keyswitch is pressed (e.g. a Qukeys key whose value has not yet been resolved). Inactive keys are represented in the `live_keys[]` array by the special value `Key_Inactive`.

Masked keys

In the `live_keys[]` array, a *masked* key is one whose next key press (either physical or logical) will be ignored. A masked key is automatically unmasked the next time it toggles off. Masked keys are represented by the special value `Key_Masked`.

11.4.2 Keyswitch state

Pressed

The state of a keyswitch that has been actuated by the user or a routine acting on behalf of the user

Unpressed

The state of a keyswitch that is not currently actuated

Toggled on

The state of a keyswitch that was not pressed during the last scan cycle and is now pressed.

Toggled off

The state of a keyswitch that was pressed during the last scan cycle and is no longer pressed.

Cycle

The `loop` method in one's sketch file is the heart of the firmware. It runs - as the name suggests - in a loop. We call these runs cycles. A lot of things happen within a cycle: from key scanning, through key event handling, LED animations, and so on and so forth.

Event handler

A function, usually provided by a *`Plugin`* that is run by a *`Hook`*.

At the time of this writing, the following event handlers are run by hooks:

- `onSetup`: Run once, when the plugin is initialised during `Kaleidoscope.setup()`.
- `beforeEachCycle`: Run as the first thing at the start of each *cycle*.
- `onKeyswitchEvent`: Run for every non-idle key, in each *cycle* the key isn't idle in. If a key gets pressed, released, or is held, it is not considered idle, and this event handler will run for it too.

- `beforeReportingState`: Runs each *cycle* right before sending the various reports (keys pressed, mouse events, etc) to the host.
- `afterEachCycle`: Runs at the very end of each *cycle*.

Hook

A point where the core firmware calls *event handlers*, allowing *plugins* to augment the firmware behaviour, by running custom code.

Plugin

An Arduino library prepared to work with Kaleidoscope. They implement methods from the `kaleidoscope::Plugin` (usually a subset of them). See *event handlers* above for a list of methods.

11.4.3 Testing

These terms arise when discussing the testing framworks and related tests.

Sim Harness

An abstraction used in testing to inject events into or invoke actions on the simulated firmware. This abstraction comprises half of the interface to the *test simulator*.

Sim State

An abstraction used in testing to encapsulate, snapshot, and examine firmware state. This abstraction comprises half of the interface to the *test simulator*.

Test

An indivial assertion or expectation that must hold for a *test case* to pass.

Test Case

An indivual `TEST*` macro invocation. Its body consists of one or *tests* and optionally other code, e.g. to invoke the *test harness*. Note that gtest uses the non-standard term *Test* for what we call a *Test Case*.

Test File

An individual file containing one or more *test suites*.

Test Fixture

A class comprising setup, teardown, and other code and common state to make writing *test cases* easieer. A fresh object of the fixture class associated with a *test suite* is constructed for each run of each *teset case* in the *test suite*.

Test Simulator

An abstraction wrapping a virtual firmware build that allows performing actions against the virtual firmware and reading state out of the virtual firmware. The interface to the test simular is comprised of the *sim harness* and the *sim state*.

Test Suite

A collection of related *test cases*, optionally with an associated *test fixture*.

11.5 Developing interdependent plugins

Say you have two Kaleidoscope plugins or, more general, two Arduino libraries `A` and `B`. Let's assume `B` depends on `A` in a sense that `B` references symbols (functions/variables) defined in `A`. Both libraries define header files `a_header.h` and `b_header.h` that specify their exported symbols.

The following sketch builds as expected.

```
// my_sketch.ino
#include "b_header.h"
#include "a_header.h"
...
```

If the header appear in opposite order the linker will throw undefined symbol errors regarding missing symbols from `A`.

```
// my_sketch.ino
#include "a_header.h"
#include "b_header.h"
...
```

The reason for this somewhat unexpected behavior is that the order of libraries' occurrence in the linker command line is crucial. The linker must see library `B` first to determine which symbols it needs to extract from `A`. If it encounters `A` first, it completely neglects its symbols as there are no references to it at that point.

To be on the safe side and only if the sketch does not reference symbols from `A` directly, it is better to include the headers in the following way.

```
// header_b.h
#include "header_a.h"
...
```

```
// my_sketch.ino
// Do not include a_header.h directly. It is already included by b_header.h.
#include "b_header.h"
...
```

Note: I did no thorough research on how Arduino internally establishes the linker command line, e.g. with respect to a recursive traversal of the include-tree. This means, I am not sure how the link command line order is generated when header files that are included by the main `.ino` do include other files that provide definitions of library symbols in different orders. There might be additional pitfalls when header includes are more complex given a larger project.

11.6 Kaleidoscope Maintainers

We consider pull requests to the Kaleidoscope GitHub repo.

- obra
- noseglasses
- algernon

11.7 Kaleidoscope Device API internals

This document is aimed at people interested in working on adding new devices - or improving support for existing ones - to Kaleidoscope. The APIs detailed here are a little bit more complex than most of the APIs our plugins provide. Nevertheless, we hope they're still reasonably easy to use, and this document is an attempt to explain some of the more intricate parts of it.

11.7.1 Overview

The core idea of the APIs is that to build up a device, we compose various components together, by describing their *properties*, and using fairly generic, templated helper classes with the properties as template parameters.

This way, we can assemble together a device with a given *MCU*, which uses a particular *Bootloader*, some kind of *Storage*, perhaps some *LEDs*, and it will more than likely have a *key scanner* component too.

The base and helper classes provide a lot of the functionality themselves, so for a device built up from components already supported by Kaleidoscope, the amount of custom code one has to write will be minimal.

11.7.2 Component details

Device

A `Device` is the topmost level component, it is the interface the rest of Kaleidoscope will work with. The `kaleidoscope::device::Base` class is the ancestor of *all* devices, everything derives from this. Devices that use an `ATmega32U4` MCU we also have the `kaleidoscope::device::ATmega32U4Keyboard` class, which sets up some of the components that is common to all `ATmega32U4`-based devices (such as the *MCU* and the *Storage*).

As hinted at above, a device - or rather, it's `Props` - describe the components used for the device, such as the MCU, the Bootloader, the Storage driver, LEDs, and the key scanner. If any of that is unneeded, there's no need to specify them in `Props` - the defaults are all no-ops.

All devices must also come with a `Props` struct, deriving from `kaleidoscope::device::BaseProps`.

As an example, the most basic device we can have, that does nothing, would look like this:

```
class ExampleDevice : public kaleidoscope::device::Base<kaleidoscope::device::BaseProps>
↪{};
```

That's not very useful, though. More often than not, we want to override at least some of the properties. In some cases, even override some of the pre-defined methods of the device. See the base class for an up-to-date list of methods and interfaces it provides. The most often changed methods are likely to be `setup()` and the constructor, and `enableHardwareTestMode()` if the device implements a hardware test mode. The rest are wrappers around the various components described by the `Props`.

In other words, the majority of customisation is in the `Props`, and in what components the device ends up using.

MCU

The heart of any device will be the main controller unit, or *MCU* for short. The `kaleidoscope::driver::mcu::Base` class is the ancestor of our MCU drivers, including `kaleidoscope::driver::mcu::ATmega32U4`.

The core firmware will use the `detachFromHost()` and `attachToHost()` methods of the MCU driver, along with `setup()`, but the driver - like any other driver - is free to have other methods, to be used by individual devices.

For example, the `ATmega32U4` driver implements a `disableJTAG()` and a `disableClockDivision()` method, which some of our devices use in their constructors.

Unlike some other components, the `MCU` component has no properties.

Bootloader

Another important component of a device is a bootloader. The bootloader is the thing that allows us to re-program the keyboard without additional hardware (aptly called a programmer). As such, the `base class` has a single method, `rebootBootloader()`, which our bootloader components implement.

Kaleidoscope currently supports `Caterina`, `HalfKay`, and `FLIP` bootloaders. Please consult them for more information. In many cases, setting up the bootloader in the device props is all one needs to do.

Like the *MCU* component, the *bootloader* does not use Props, either.

Storage

Not nearly as essential for a device is the `Storage` component. Storage is for persistent storage of configuration data, such as key maps, colormaps, feature toggles, and so on. It's not a required component, but a recommended one nevertheless. This storage component is what allows apps like Chrysalis to configure some aspects of the keyboard without having to flash new firmware.

The Storage API resembles the Arduino EEPROM API very closely. In fact, our `AVREEPROM` class is but a thin wrapper around that!

The `Storage` component does use Props, one that describes the length - or size - of it. We provide an `ATmega32U4EEPROMProps` helper, which is preconfigured for the 1k EEPROM size of the ATmega32U4.

LEDs

`kaleidoscope::driver::led::Base`

Keyscanner

`kaleidoscope::driver::keyscanner::Base`

11.7.3 Helpers

`kaleidoscope::device::ATmega32U4Keyboard` `kaleidoscope::driver::keyscanner::ATmega`

11.7.4 Putting it all together

To put things into perspective, and show a simple example, we'll build an imaginary mini keypad: `ATmega32U4` with `Caterina` as bootloader, no LEDs, and four keys only.

`ImaginaryKeypad.h`

```
#pragma once

#ifdef ARDUINO_AVR_IMAGINARY_KEYPAD

#include <Arduino.h>
#include "kaleidoscope/driver/keyscanner/ATmega.h"
#include "kaleidoscope/driver/bootloader/avr/Caterina.h"
#include "kaleidoscope/device/ATmega32U4Keyboard.h"

namespace kaleidoscope {
namespace device {
namespace imaginary {

struct KeypadProps : kaleidoscope::device::ATmega32U4KeyboardProps {
  struct KeyScannerProps : public kaleidoscope::driver::keyscanner::ATmegaProps {
    static constexpr uint8_t matrix_rows = 2;
    static constexpr uint8_t matrix_columns = 2;
    typedef MatrixAddr<matrix_rows, matrix_columns> KeyAddr;
    static constexpr uint8_t matrix_row_pins[matrix_rows] = {PIN_D0, PIN_D1};
    static constexpr uint8_t matrix_col_pins[matrix_columns] = {PIN_C0, PIN_C1};
  };

  typedef kaleidoscope::driver::keyscanner::ATmega<KeyScannerProps> KeyScanner;
  typedef kaleidoscope::driver::bootloader::avr::Caterina Bootloader;
  static constexpr const char *short_name = "imaginary-keypad";
};

class Keypad: public kaleidoscope::device::ATmega32U4Keyboard<KeypadProps> {};

#define PER_KEY_DATA(dflt, \
  R0C0, R0C1,              \
```

(continues on next page)

(continued from previous page)

```
  R1C0, R1C1                \
)                           \
  R0C0, R0C1, R1C0, R1C1

}
}

EXPORT_DEVICE(kaleidoscope::device::imaginary::Keypad);

}
#endif
```

ImaginaryKeypad.cpp

```
#ifdef ARDUINO_AVR_IMAGINARY_KEYPAD

#include <Kaleidoscope.h>

// Here, we set up aliases to the device's KeyScanner and KeyScannerProps in the
// global namespace within the scope of this file. We'll use these aliases to
// simplify some template initialization code below.
using KeyScannerProps = typename␣
↪kaleidoscope::device::imaginary::KeypadProps::KeyScannerProps;
using KeyScanner = typename kaleidoscope::device::imaginary::KeypadProps::KeyScanner;

namespace kaleidoscope {
namespace device {
namespace imaginary {

// `KeyScannerProps` here refers to the alias set up above. We do not need to
// prefix the `matrix_rows` and `matrix_columns` names within the array
// declaration, because those are resolved within the context of the class, so
// the `matrix_rows` in `KeyScannerProps::matrix_row_pins[matrix_rows]` gets
// resolved as `KeyScannerProps::matrix_rows`.
const uint8_t KeyScannerProps::matrix_rows;
const uint8_t KeyScannerProps::matrix_columns;
constexpr uint8_t KeyScannerProps::matrix_row_pins[matrix_rows];
constexpr uint8_t KeyScannerProps::matrix_col_pins[matrix_columns];

// `KeyScanner` here refers to the alias set up above, just like in the
// `KeyScannerProps` case above.
template<> KeyScanner::row_state_t KeyScanner::matrix_state_[KeyScannerProps::matrix_
↪rows] = {};

// We set up the TIMER1 interrupt vector here. Due to dependency reasons, this
// cannot be in a header-only driver, and must be placed here.
//
// Timer1 is responsible for setting a property on the KeyScanner, which will
// tell it to do a scan. We use this to make sure that scans happen at roughly
// the intervals we want. We do the scan outside of the interrupt scope for
```

(continues on next page)

(continued from previous page)

```
// practical reasons: guarding every codepath against interrupts that can be
// reached from the scan is far too tedious, for very little gain.
ISR(TIMER1_OVF_vect) {
  Runtime.device().keyScanner().do_scan_ = true;
}

}
}
}
#endif
```

That's it.

11.8 Kaleidoscope's Plugin Event Handlers

Kaleidoscope provides a set of hook functions that plugins can define in order to do their work. If one or more of the functions listed here are defined as methods in a plugin class, that plugin can act on the input events that drive Kaleidoscope.

In response to input events (plus a few other places), Kaleidoscope calls the event handlers for each plugin that defines them, in sequence.

11.8.1 Return values

Every Kaleidoscope event handler function returns a value of type `EventHandlerResult`, an enum with several variants. In some handlers, Kaleidoscope ignores the return value, but for others, the result is used as a signal to control Kaleidoscope's behavior. In particular, some event handler hooks are "abortable". For those hooks, the return value of the plugin handlers are used to control what Kaleidoscope does after each plugin's event handler returns.

- `EventHandlerResult::OK` is used to signal that Kaleidoscope should continue on to the next handler in the sequence.
- `EventHandlerResult::ABORT` is used to signal that Kaleidoscope should not continue to call the other plugin handlers in the sequence, and stop processing the event entirely. This is used by some plugins to cancel events and/or delay them so that they occur at a later time, possibly with different values.
- `EventHandlerResult::EVENT_CONSUMED` is used to signal that the plugin has successfully handled the event, and that there is nothing further to be done, so there is no point in continuing to call further plugin event handlers for the event.

11.8.2 Non-event "event" handlers

There are three special "event" handlers that are not called in response to input events, but are instead called at fixed points during Kaleidoscope's run time.

`onSetup()`

This handler is called when Kaleidoscope first starts, at the end of the `setup()` method. If a plugin needs to do some work after its constructor is called, but before Kaleidoscope enters its main loop and starts scanning for keyswitch events, it can do it in this function.

It takes no arguments, and must return `kaleidoscope::EventHandlerResult::OK`.

`beforeEachCycle()`

This handler gets called at the beginning of every keyswitch scan cycle, before the scan. It can be used by plugins to do things that need to be done repeatedly, regardless of any input from the user. Typically, this involves things like checking for timeouts.

Takes no arguments, must return `kaleidoscope::EventHandlerResult::OK`.

`afterEachCycle()`

This is just like `beforeEachCycle()`, but gets called after the keyswitches have been scanned (and any input events handled).

11.8.3 Keyswitch input event handlers

This group of event handlers is triggered when keys on the keyboard are pressed and released. With one exception, they use a `KeyEvent` object as their one parameter. The `KeyEvent` class encapsulates the essential data about a key press (or release):

- `event.addr` contains the `KeyAddr` of the key that toggled on or off.
- `event.state` contains information about the current and former state of the key in the form of a `uint8_t` bitfield.
- `event.key` contains the `Key` value of the event. For key presses, this is generally determined by means of a keymap lookup. For releases, the value is taken from the `live_keys` structure. Because the `event` is passed by reference, changing this value in a plugin handler will affect which value ends up in the `live_keys` array, and thus, the output of the keyboard.
- `event.id` contains a `KeyEventId` value: an integer, usually monotonically increasing. This is useful as a tool to allow plugins to avoid re-processing the same event, thus avoiding infinite loops without resorting to an `INJECTED` key state flag which would cause other plugins to ignore events that they might otherwise be interested in.

`onKeyswitchEvent(KeyEvent &event)`

This handler is called in response to changes detected in the state of keyswitches, via the `Runtime.handleKeyswitchEvent()` function. After the keyswitches are scanned in each cycle, Kaleidoscope goes through them all and compares the state of each one to its previous state. For any of them that have either toggled on or off, plugins that define this function get called (until one of them returns either `ABORT` or `EVENT_CONSUMED`).

This handler should be defined by any plugin that is concerned only with physical keyswitch events, where the user has pressed or released a physical key. For example, plugins that determine key values based on the timing of these physical events should define this handler (for example, Qukeys and TapDance). Plugins that don't explicitly need to use this handler should define `onKeyEvent()` instead.

Plugins that use this handler should abide by certain rules in order to interact with each other to avoid infinite loops. A plugin might return `ABORT` to delay an event (until some other event or a timeout occurs), then later re-start processing of the same event by calling `Runtime.handleKeyswitchEvent()`. When it does this, it must take care to use the same `KeyEventId` value as that event's `id` parameter, and it should also take care to preserve the order of any such events. This way, plugins implementing `onKeyswitchEvent()` are able to keep track of event id numbers that they have already processed fully, and ignore those events when plugins later in the sequence re-start them.

In more specific detail, plugins that implement `onKeyswitchEvent()` must guarantee that the `event.id` values they emit when returning `OK` are monotonically increasing, and should only include `id` values that the plugin has already received as input. Additionally, such plugins must ignore any event with an `id` value that it has recently received and finished processing. The class `KeyEventTracker` can help simplify following these rules.

`onKeyEvent(KeyEvent &event)`

After a physical keyswitch event is processed by all of the plugins with `onKeyswitchEvent()` handlers (and they all return `OK`), Kaleidoscope passes that event on to the `Runtime.handleKeyEvent()` function, which calls plugins' `onKeyEvent()` handlers. This is also the starting point for events which do not correspond to physical key events, and can have an invalid `event.addr` value.

Plugins that need to respond to keyboard input, but which do not need to be closely tied to physical key events (and only those events) should use `onKeyEvent()` to do their work.

After all `onKeyEvent()` handlers have returned `OK` for an event, the `live_keys` state array gets updated. For a key press event, the final `event.key` value gets inserted into `live_keys[event.addr]`. From that point on, the keyboard will behave as though a key with that value is being held until that entry in `live_keys` is cleared (most likely as a result of a key release event's `onKeyEvent()` handlers returning `OK`). Thus, if an `onKeyEvent()` handler returns `ABORT` for a key release event, the keyboard will behave as though that key is still held after it has been released. This is what enables plugins like OneShot to function, but it also means that plugin authors need to take care about returning `ABORT` (but not `EVENT_CONSUMED`) from an `onKeyEvent()` handler, because it could result in "stuck" keys.

`onKeyEvent()` handlers should not store events and release them later (by calling `Runtime.handleKeyEvent()`), and must never call `Runtime.handleKeyswitchEvent()`.

`onAddToReport(Key key)`

After the `onKeyEvent()` handlers have all returned `OK`, Kaleidoscope moves on to sending Keyboard HID reports. It clears the current report, and iterates through the `live_keys` array, looking for non-empty values, and adding them to the report. For System Control, Consumer Control, and Keyboard HID type `Key` values, Kaleidoscope handles adding the keycodes to the correct report, but it also calls this handler, in case a plugin needs to alter that report.

A return value of `OK` allows Kaleidoscope to proceed with adding the corresponding keycode(s) to the HID report, and `ABORT` causes it to leave and keycodes from `key` out of the report.

Note that this only applies to the Keyboard and Consumer Control HID reports, not the System Control report, which has different semantics, and only supports a single keycode at a time.

`beforeReportingState(const KeyEvent &event)`

This gets called right before a set of HID reports is sent. At this point, plugins have access to a (tentative) complete HID report, as well as the full state of all live keys on the keyboard. This is especially useful for plugins that might need to do things like remove keycodes (such as keyboard modifiers) from the forthcoming report just before it gets sent.

This event handler still has access to the event information for the event that triggered the report, but because it is passed as a `const` reference, it is no longer possible to change any of its values.

[Note: The older version of `beforeReportingState()` got called once per cycle, regardless of the pattern of keyswitches toggling on and off, and many plugins used it as a place to do things like check for timeouts. This new version does not get called every cycle, so when porting old code to the newer handlers, it's important to move any code that must be called every cycle to either `beforeEachCycle()` or `afterEachCycle()`.]

[Also note: Unlike the deprecated `beforeReportingState()`, this one is abortable. That is, if it returns a result other than `OK` it will stop the subsequent handlers from getting called, and if it returns `ABORT`, it will also stop the report from being sent.]

`afterReportingState(const KeyEvent &event)`

This gets called after the HID report is sent. This handler allows a plugin to react to an event, but wait until after that event has been fully processed to do so. For example, the OneShot plugin releases keys that are in the "one-shot" state in response to key press events, but it does so after those triggering press events take place.

11.8.4 Other events

`onLayerChange()`

Called whenever one or more keymap layers are activated or deactivated (just after the change takes place).

`onLEDModeChange()`

Called by `LEDControl` whenever the active LED mode changes.

`beforeSyncingLeds()`

Called immediately before Kaleidoscope sends updated color values to the LEDs. This event handler is particularly useful to plugins that need to override the active LED mode (e.g. LED-ActiveModColor).

`onFocusEvent()`

Used to implement *bi-directional communication*. This is called whenever the firmware receives a command from the host. The only argument is the command name. Can return `kaleidoscope::EventHandlerResult::OK` to let other plugins process the event further, or `kaleidoscope::EventHandlerResult::EVENT_CONSUMED` to stop processing.

`onNameQuery()`

Used by the *Focus* plugin, when replying to a `plugins` command. Should either send the plugin name, or not be implemented at all, if the host knowing about the plugin isn't important.

`exploreSketch()`

11.8.5 Deprecated

Two existing "event" handlers have been deprecated. In the old version of Kaleidoscope's main loop, the keyboard's state information was stored in the keyscanner (which physical switches were on in the current and former scans), and in the HID reports. The Keyboard HID report would be cleared at the start of every cycle, and re-populated, on key at a time, calling every `onKeyswitchEvent()` handler for every active key. Then, once the tentative HID report was complete, the `beforeReportingState()` handlers would be called, and the complete report would be sent to the host. In most cycles, that report would be identical to the previous report, and would be suppressed.

The new system stores the keyboard's current state in the `live_keys` array instead, and only calls event handlers in response to keyswitch state changes (and artificially generated events), ultimately sending HID reports in response to events, rather than at the end of every cycle.

`onKeyswitchEvent(Key &key, KeyAddr key_addr, uint8_t key_state)`

This handler was called in every cycle, for every non-idle key. Its concept of an "event" included held keys that did not have a state change. These deprecated handlers are still called, in response to events and also when preparing the HID reports, but there is no longer a reasonable mechanism to call them in every cycle, for every active key, so some functionality could be lost.

It is strongly recommended to switch to using one of the two `KeyEvent` functions instead, depending on the needs of the plugin (either `onKeyEvent()` if it is fit for the purpose, or `onKeyswitchEvent()` if necessary). The `onAddToReport()` function might also be useful, particularly if the plugin in question uses special `Key` values not recognized by Kaleidoscope itself, but which should result in keycodes being added to HID reports.

`beforeReportingState()`

The old version of this handler has been deprecated, but it will still be called both before HID reports are sent and also once per cycle. It is likely that these handlers will continue to function, but the code therein should be moved either to the new `KeyEvent` version of `beforeReportingState()` and/or `afterEachCycle()` (or `beforeEachCycle()`), depending on whether it needs to be run only in response to input events or if it must execute every cycle, respectively.

11.9 Kaleidoscope Plugin API Internals

In this document we explain how the plugin system works behind the scenes.

This is useful because there are some unorthodox solutions in play that make the code incredibly hard to untangle. It's an unavoidable side effect of employing a system that uses non-virtual functions, which lets us save large amounts of RAM.

Let's start at the top:

11.9.1 KALEIDOSCOPE_INIT_PLUGINS

```
#define KALEIDOSCOPE_INIT_PLUGINS(...) _KALEIDOSCOPE_INIT_PLUGINS(__VA_ARGS__)
```

So far so good, this is pretty simple. The reason we use an indirection here is because this is in `Kaleidoscope.h`, and we do not want the complexity of the `_KALEIDOSCOPE_INIT_PLUGINS` macro here, nor do we want to move the macro to another header, because it belongs to `Kaleidoscope.h`.

11.9.2 _KALEIDOSCOPE_INIT_PLUGINS

```
#define _KALEIDOSCOPE_INIT_PLUGINS(...)                                    \
  namespace kaleidoscope_internal {                                        \
  struct EventDispatcher {                                                 \
                                                                           \
    template<typename EventHandler__, typename... Args__ >                 \
    static kaleidoscope::EventHandlerResult apply(Args__&&... hook_args) { \
                                                                           \
      kaleidoscope::EventHandlerResult result;                             \
      MAP(_INLINE_EVENT_HANDLER_FOR_PLUGIN, __VA_ARGS__)                   \
                                                                           \
      return result;                                                       \
    }                                                                      \
  };                                                                       \
                                                                           \
  }                                                                        \
  _FOR_EACH_EVENT_HANDLER(_REGISTER_EVENT_HANDLER)
```

This is where things get interesting. This macro does two things:

- It creates `kaleidoscope_internal::EventDispatcher`, a class with a single method, `apply`. This is a templated method. The template argument is the method `apply` calls. Therefore, `EventDispatcher::template apply<foo>` resolves to a function that calls the `foo` method of each plugin we list for `KALEIDOSCOPE_INIT_PLUGINS`. We'll see in a bit how this happens.
- The other part creates overrides for the `Kaleidoscope::Hooks::` family of functions. These are wrappers around `EventDispatcher::template apply<foo>`. We have these so that higher level code does not need to be concerned with the implementation details. It can invoke the hooks as if they were ordinary functions.

11.9.3 `_FOR_EACH_EVENT_HANDLER(_REGISTER_EVENT_HANDLER)`

Let's look at `_FOR_EACH_EVENT_HANDLER` and `_REGISTER_EVENT_HANDLER` first, because that's easier to explain, and does not lead down another rabbit hole.

`_REGISTER_EVENT_HANDLER`

```
#define _REGISTER_EVENT_HANDLER(                                          \
    HOOK_NAME, SHOULD_ABORT_ON_CONSUMED_EVENT, SIGNATURE, ARGS_LIST)      \
                                                                          \
  namespace kaleidoscope_internal {                                       \
                                                                          \
   struct EventHandler_##HOOK_NAME {                                      \
                                                                          \
      static bool shouldAbortOnConsumedEvent() {                          \
        return SHOULD_ABORT_ON_CONSUMED_EVENT;                            \
      }                                                                   \
                                                                          \
      template<typename Plugin__, typename... Args__>                     \
      static kaleidoscope::EventHandlerResult                             \
        call(Plugin__ &plugin, Args__&&... hook_args) {                   \
         _VALIDATE_EVENT_HANDLER_SIGNATURE(HOOK_NAME, Plugin__)           \
         return plugin.HOOK_NAME(hook_args...);                           \
      }                                                                   \
    };                                                                    \
                                                                          \
   }                                                                          \
                                                                          \
   namespace kaleidoscope {                                               \
                                                                          \
     EventHandlerResult Hooks::HOOK_NAME SIGNATURE {                      \
        return kaleidoscope_internal::EventDispatcher::template           \
        apply<kaleidoscope_internal::EventHandler_ ## HOOK_NAME>          \
             ARGS_LIST;                                                   \
      }                                                                   \
                                                                          \
   }
```

This looks big and scary, but in practice, it isn't all that bad. Nevertheless, this is where the magic happens!

We create two things: `EventHandler_SomeThing` and `Hooks::SomeThing`, the latter being a wrapper around the first, that uses `EventDispatcher::template apply<>` discussed above.

Lets take `onSetup` as an example! For that, the above expands to:

```
namespace kaleidoscope_internal {

struct EventHandler_onSetup {
  static bool shouldAbortOnConsumedEvent() {
    return false;
  }

  template<typename Plugin__, typename... Args__>
```

(continues on next page)

(continued from previous page)

```
  static kaleidoscope::EventHandlerResult
  call(Plugin__ &plugin, Args__&&... hook_args) {
    return plugin.onSetup(hook_args...);
  }
};

}

namespace kaleidoscope {

EventHandlerResult Hooks::onSetup() {
  return kaleidoscope_internal::EventDispatcher::template
         apply<kaleidoscope_internal::EventHandler_onSetup>();
}

}
```

This still looks scary… but please read a bit further, and it will all make sense!

_FOR_EACH_EVENT_HANDLER

This just evaluates its argument for each event handler supported by Kaleidoscope core. Very simple macro expansion, which we will not expand here, because that would take up a lot of space, and they all look the same (see above).

11.9.4 EventDispatcher

```
namespace kaleidoscope_internal {
struct EventDispatcher {
  template<typename EventHandler__, typename... Args__ >
  static kaleidoscope::EventHandlerResult apply(Args__&&... hook_args) {

    kaleidoscope::EventHandlerResult result;
    MAP(_INLINE_EVENT_HANDLER_FOR_PLUGIN, __VA_ARGS__)

    return result;
  }
};
```

This is where the other part of the magic happens, and we need to understand this to be able to make sense of _REGISTER_EVENT_HANDLER above.

_INLINE_EVENT_HANDLER_FOR_PLUGIN

In isolation, this is not very interesting, and is closely tied to EventDispatcher. The definition is here so we can look at it while we learn the details of EventDispatcher below.

```
#define _INLINE_EVENT_HANDLER_FOR_PLUGIN(PLUGIN)                      \
                                                                      \
   result = EventHandler__::call(PLUGIN, hook_args...);              \
                                                                      \
   if (EventHandler__::shouldAbortOnConsumedEvent() &&               \
       result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) { \
     return result;                                                   \
   }
```

Back to EventDispatcher...

The EventDispatcher structure has a single method: apply<>, which requires an event handler as its template argument. The macros calls the event handler given in the template argument for each and every initialised plugin. It's best explained with an example! Let's use two plugins, SomePlugin and ExampleEffect:

```
namespace kaleidoscope_internal {
struct EventDispatcher {
  template<typename EventHandler__, typename... Args__ >
  static kaleidoscope::EventHandlerResult apply(Args__&&... hook_args) {

    kaleidoscope::EventHandlerResult result;

    result = EventHandler__::call(SomePlugin);
    if (EventHandler__::shouldAbortOnConsumedEvent() &&
        result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
        return result;
    }

    result = EventHandler__::call(ExampleEffect);
    if (EventHandler__::shouldAbortOnConsumedEvent() &&
        result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
        return result;
    }

    return result;
  }
};
```

See? It's unrolled! But how do we get from here to - say - calling the onSetup() method of the plugins? Why, by way of EventDispatcher::template apply<EventHandler_onSetup>! Lets see what happens when we do a call like that:

```
namespace kaleidoscope_internal {
struct EventDispatcher {
  template<typename EventHandler_onSetup, typename... Args__ >
  static kaleidoscope::EventHandlerResult apply(Args__&&... hook_args) {

    kaleidoscope::EventHandlerResult result;
```

(continues on next page)

(continued from previous page)

```
    result = EventHandler_onSetup::call(SomePlugin);
    if (EventHandler_onSetup::shouldAbortOnConsumedEvent() &&
        result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
        return result;
    }

    result = EventHandler_onSetup::call(ExampleEffect);
    if (EventHandler_onSetup::shouldAbortOnConsumedEvent() &&
        result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
        return result;
    }

    return result;
  }
};
```

Because we call `EventHandler_onSetup::call` with the plugin as the first argument, and because `call` is also a templated function, where the first argument is templated, we get a method that is polymorphic on its first argument. This means that for each and every plugin, we have a matching `EventHandler_onSetup::call` that is tied to that plugin. *This* is the magic that lets us use non-virtual methods.

11.9.5 Exploring what the compiler does

Because all hooks are called via `kaleidoscope::Hooks::NAME`, let's explore how the compiler will optimize the code for `onSetup`, assuming we use two plugins, `SomePlugin` and `ExampleEffect`.

Our entry point is this:

```
return kaleidoscope::Hooks::onSetup();
```

`_REGISTER_EVENT_HANDLER` created `Hooks::onSetup()` for us:

```
EventHandlerResult kaleidoscope::Hooks::onSetup() {
  return kaleidoscope_internal::EventDispatcher::template
         apply<kaleidoscope_internal::EventHandler_onSetup>();
}
```

If we inline the call to `EventDispatcher::template apply<>`, we end up with the following:

```
EventHandlerResult kaleidoscope::Hooks::onSetup() {
  kaleidoscope::EventHandlerResult result;

  result = EventHandler_onSetup::call(SomePlugin);
  if (EventHandler_onSetup::shouldAbortOnConsumedEvent() &&
    result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
    return result;
  }

  result = EventHandler_onSetup::call(ExampleEffect);
  if (EventHandler_onSetup::shouldAbortOnConsumedEvent() &&
    result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
```

(continues on next page)

(continued from previous page)

```
    return result;
  }

  return result;
}
```

This is starting to look human readable, isn't it? But we can go further, because `EventHandler::onSetup::call` and `EventHandler_onSetup::shouldAbortOnConsumedEvent` are evaluated at compile-time too!

```
EventHandlerResult kaleidoscope::Hooks::onSetup() {
  kaleidoscope::EventHandlerResult result;

  result = SomePlugin.onSetup();
  if (false &&
    result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
    return result;
  }

  result = ExampleEffect.onSetup();
  if (false &&
    result == kaleidoscope::EventHandlerResult::EVENT_CONSUMED) {
    return result;
  }

  return result;
}
```

Which in turn, may be optimized further to something like the following:

```
EventHandlerResult kaleidoscope::Hooks::onSetup() {
  kaleidoscope::EventHandlerResult result;

  result = SomePlugin.onSetup();
  result = ExampleEffect.onSetup();

  return result;
}
```

And this, is the end of the magic. This is roughly how much the code gets transformed *at compile time*, so that at run-time, none of this indirection is present.

11.9.6 Summary

As you can see, there is a lot going on behind the scenes, and a combination of template meta programming and pre-processor macros is used to accomplish our goal. Following the code path as outlined above allows us to see what the compiler sees (more or less), and inlining all the things that are done at compile-time provides us with the final code, which is pretty simple by that point.

11.10 kaleidoscope::driver::bootloader

We rarely have to work with or care about the bootloader from the user firmware, but there's one scenario when we do: if we want to reset the device, and put it into bootloader (programmable) mode, we need to do that in a bootloader-specific manner.

This driver provides a number of helpers that implement the reset functionality for various bootloaders.

11.10.1 Using the driver

To use the driver, we need to include the appropriate header, from the hardware plugin of our keyboard:

```
#include <kaleidoscope/driver/bootloader/avr/Caterina.h>
```

Next, we set up the device `Properties` so that it includes the override for our bootloader:

```
struct OurBoardProps : kaleidoscope::device::BaseProps {
  typedef kaleidoscope::driver::bootloader::avr::Caterina Bootloader;
};
```

The base classes will do all the rest.

11.10.2 Methods provided by all bootloader drivers

`.rebootBootloader()`

Resets the device, and forces it into bootloader (programmable) mode.

11.10.3 List of bootloaders

All of the drivers below live below the `kaleidoscope::driver::bootloader` namespace.

11.10.4 `avr::Caterina:`

Used by many (most?) arduino MCUs. Provided by `kaleidoscope/driver/bootloader/avr/Caterina.h`.

`avr::HalfKay`

Used by the Teensy2. Provided by `kaleidoscope/driver/bootloader/avr/HalfKay.h`.

avr::FLIP

Used by the ATMega32U4 MCUs by default, unless another bootloader has been flashed on them. Provided by kaleidoscope/driver/bootloader/avr/FLIP.h.

For this driver to work, one also needs to define the KALEIDOSCOPE_BOOTLOADER_FLIP_WORKAROUND macro before including the driver header, for technical reasons.

11.11 kaleidoscope::driver::led::WS2812

This driver provides a generic base class for driving WS2812-based LED strips. This is not a plugin, and is not meant to be user-facing. It is meant to be used by developers, and hardware plugins in particular. See the KBDFans KBD4x plugin for a practical, existing example about how to use the driver.

11.11.1 Using the driver

To use the driver, we need to include the header:

```
#include <kaleidoscope/driver/led/WS2812.h>
```

For performance reasons, the driver is templated, and requires three template arguments:

1. pin, the PIN the driver will use to communicate with the LED strip.
2. class Color, the color class that determines the order of the RGB components, and should match the component order the LED strip uses. We provide three orders out of the box, all in the kaleidoscope::driver::led::color namespace: RGB, GRB, and BGR.
3. ledCount is the number of LEDs on the strip this instance of the driver should be able to address.

Armed with this knowledge, instantiating an object is as easy as:

```
using Color = kaleidoscope::driver::led::color::GRB;

kaleidoscope::driver::led::WS2812<PIN_E2, Color, 6> LEDs;
```

11.11.2 Driver methods

The instantiated WS2812 object will have the following methods:

.led_count()

Returns the number of LEDs, the same value as the ledCount template argument.

`.sync()`

Synchronises the internal LED state with the hardware, by sending over all of the LED data.

It is recommended to call this at most once per cycle. Calling it less frequently isn't wrong either.

`.setColorAt(index, color)`

`.setColorAt(index, r, g, b)`

Sets the color at the given `index` to the specified value. He value can either be a `Color` object (the same type as the template argument), or a list of RGB component values.

`.getColorAt(index)`

Returns the color at the given `index`, as a `Color` object.

11.11.3 Further information

To have a better idea how to use the driver in practice, looking at the KBD4x hardware library is recommended.

11.12 Automated Testing

11.12.1 Testing with gtest/gmock

Before feature requests or bug fixes can be merged into master, the folowing steps should be taken:

- Create a new test suite named after the issue, e.g. `Issue840`.
- Add a test case named `Reproduces` that reproduces the bug. It should fail if the bug is present and pass if the bug is fixed.
- Merge the proposed fix into a temporary testing branch.
- The reproduction test should fail.
- Add a test called "HasNotRegresed" that detects a potential regression. It should pass with the fix and fail before the fix.
- Comment out and keep the `Reproduces` test case as documentation.

For an example, see keyboardio:Kaleidoscope/tests/issue_840.

Adding a New Test Case

For general information on writing test with gtest/gmock see gtest docs and gmock docs.

1. Create a new test file, if appropriate.
2. Choose a new test suite name and create a new test fixture, if appropriate.
3. Write your test case.

The final include in any test file should be `#include "testing/setup-googletest.h"` which should be followed by the macro invocation `SETUP_GOOGLETEST()`. This will take care of including headers commonly used in tests in addtion to gtest and gmock headers.

Any test fixtures should inherit from `VirtualDeviceTest` which wraps the test sim APIs and provides common setup and teardown functionality. The appropriate header is already imported by `setup-googletest.h`

Test Infrastructure

If you need to modify or extend test infrastructure to support your use case, it can currently be found under `keyboardio:Kaleidoscope/testing`.

Style

TODO(obra): Fill out this section to your liking.

You can see samples of the desired test style in the *example tests*.

Examples

All existing tests are examples and may be found under `keyboardio:Kaleidoscope/tests`.

11.12.2 Testing with Aglais/Papilio

TODO(obra): Write (or delegate the writing of) this section.

11.13 Release testing

Before a new release of Kaleidoscope, the following test process should be run through on all supported operating systems.

Always run all of the automated tests to verify there are no regressions.

(As of August 2017, this whole thing really applies to Model01-Firmware, but we hope to generalize it to Kaleidoscope)

11.14 Tested operating systems

- The latest stable Ubuntu Linux release running X11. (We *should* eventually be testing both X11 and Wayland)
- The latest stable release of macOS
- An older Mac OS X release TBD. (There were major USB stack changes in 10.9 or so)
- Windows 10
- Windows 7
- The current release of ChromeOS
- A currentish android tablet that supports USB Host
- an iOS device (once we fix the usb connection issue to limit power draw)

11.15 Test process

11.15.1 Basic testing

1. Plug the keyboard in
2. Make sure the host OS doesn't throw an error
3. Make sure the LED in the top left doesn't glow red
4. Make sure the LED in the top-right corner of the left side breathes blue for ~10s
5. Bring up some sort of notepad app or text editor

11.15.2 Basic testing, part 2

1. Test typing of shifted and unshifted letters and numbers with and without key repeat
2. Test typing of fn-shifted characters: []{}|\ with and without key repeat
3. Test that 'Any' key generates a random letter or number and that key repeat works
4. Test fn-hjkl to move the cursor
5. Test Fn-WASD to move the mouse
6. Test Fn-RFV for the three mouse buttons
7. Test Fn-BGTabEsc for mouse warp
8. Test that LeftFn+RightFn + hjkl move the cursor
9. Verify that leftfn+rightfn do not light up the numpad

11.15.3 NKRO

1. Open the platform's native key event viewer (If not available, visit https://www.microsoft.com/appliedsciences/KeyboardGhostingDemo.mspx in a browser)
2. Press as many keys as your fingers will let you
3. Verify that the keymap reports all the keys you're pressing

11.15.4 Test media keys

1. Fn-Any: previous track
2. Fn-Y: next-track
3. Fn-Enter: play/pause
4. Fn-Butterfly: Windows 'menu' key
5. Fn-n: mute
6. Fn-m: volume down
7. Fn-,: volume up

11.15.5 Test numlock

1. Tap “Num”
2. Verify that the numpad lights up red
3. Verify that the num key is breathing blue
4. Verify that numpad keys generate numbers
5. Tap the Num key
6. Verify that the numpad keys stop being lit up 1 Verify that ‘jkl’ don’t generate numbers.

11.15.6 Test LED Effects

1. Tap the LED key
2. Verify that there is a rainbow effect
3. Tap the LED key a few more times and verify that other LED effects show up
4. Verify that you can still type.

11.15.7 Second connection

1. Unplug the keyboard
2. Plug the keyboard back in
3. Make sure you can still type

11.15.8 Programming

1. If the OS has a way to show serial port devices, verify that the keyboard’s serial port shows up.
2. If you can run stty, as you can on linux and macos, make sure you can tickle the serial port at 1200 bps. Linux: stty -F /dev/ttyACM0 1200 Mac:
3. If you tickle the serial port without holding down the prog key, verify that the Prog key does not light up red
4. If you hold down the prog key before tickling the serial port, verify that the Prog key’s LED lights up red.
5. Unplug the keyboard
6. While holding down prog, plug the keyboard in
7. Verify that the prog key is glowing red.
8. Unplug the keyboard
9. Plug the keyboard in
10. Verify that the prog key is not glowing red.

11.16 If the current platform supports the Arduino IDE (Win/Lin/Mac)

1. use the Arduino IDE to reflash the current version of the software.
2. Verify that you can type a few keys
3. Verify that the LED key toggles between LED effects

11.17 Testing Kaleidoscope

This is not yet proper documentation about running or writing tests, just some rough notes.

Kaleidoscope includes a simulator that can pretend (to a certain extent) to be a keyboard for the purpose of testing.

On most UNIX-like systems, you can run Kaleidoscope's simulator tests by running

```
make simulator-tests
```

Our simulator currently has some weird linking issues on macOS, so the easiest way to run tests on macOS is using Docker.

```
make docker-simulator-tests
```

During development, when you may be running your tests very frequently, it's sometimes useful to run a subset of tests.

You can control the directory that Kaleidoscope searches for test suites with the 'TEST_PATH' variable.

To only run tests in subdirectories of the 'tests/hid' directory, you'd write:

```
make simulator-tests TEST_PATH=tests/hid
```

or

```
make docker-simulator-tests TEST_PATH=tests/hid
```

11.18 Kaleidoscope v2.0

Currently at 1.99.8, in development!

See *UPGRADING.md* for more detailed instructions about upgrading from earlier versions (even earlier betas). This is just a list of noteworthy changes.

- *New features*
- *New hardware support*
- *New plugins*
- *Breaking changes*
- *Bugfixes*

11.18.1 New features

ModLayer keys

There is a new type of built-in key that activates both a layer shift and a keyboard modifier simultaneously, and keeps both active until the key is released. Basically, it's a combination of a single modifier key (any one of the standard eight) and a `ShiftToLayer(N)` key (for any layer in the range 0-31).

A ModLayer key `key` will return `true` for the test functions `key.isKeyboardModifier()`, `key.isLayerKey()`, and `key.isLayerShift()`. As such, it can be turned into a OneShot key by either `OneShot.enableAutoModifiers()` or `OneShot.enableAutoLayers()`.

An additional umbrella test function has also been added: `key.isMomentary()`, which returns `true` for any key that is either a keyboard modifier or a layer shift (including ModLayer keys).

Layer changes updated

Layer change key handling has been updated to be more consistent with activation ordering. In most common cases there will be no obvious difference; layer move, lock, and shift keys still generally function the same way. The details now vary by type in the edge cases, however.

Layer lock keys (i.e. `LockLayer(N)`) will change slightly. Whenever a layer lock key toggles on, it will put the target layer on the top of the active layer stack, unless that layer is already at the top of the stack, in which case it will be deactivated. This means that if layers `A`, `B`, and `C` are on the stack, with `A` on top, pressing `LockLayer(B)` will move layer `B` to the top of the stack, above `A`, rather than deactivating it. As a result, if you use layers that have no transparent entries, every press of a layer lock key will result in a user-visible change.

Layer shift keys will now work independently of locked layers. This means that if layers `A`, `B` and `C` are (locked) on the stack, with `A` on top, pressing `ShiftToLayer(B)` will keep a temporary version of layer `B` on top, but when that layer shift key is released, the layer stack will return to the same state it had been in before the layer shift key was pressed, with layers `A`, `B`, and `C` still active, and in the same order. However, since it is assumed that users won't forget about keys that they are holding, pressing a second layer shift key for the same target layer will not result in promoting that shifted layer to the top of the stack.

A consequence of this is that releasing a layer shift key will no longer deactivate a layer that is locked, either from pressing a `LockLayer()` or `MoveToLayer()` key. This allows users to configure keymaps such that layer `N` could be reached by holding `ShiftToLayer(N)`, then kept active and on top by tapping `LockLayer(N)` or `MoveToLayer(N)`, which could be mapped on Layer `N`. That layer would continue to stay active after the release of the `ShiftToLayer(N)` key.

OneShot public functions

The OneShot plugin now allows other plugins to control the OneShot state of individual keys, by calling one of the following:

- `OneShot.setPending(key_addr)`: Put the key at `key_addr` in the "pending" OneShot state. This will make that key act like any other OneShot key until it is cancelled by a subsequent keypress. Once a key is in this state, OneShot will manage it from that point on, including making the key "sticky" if it is double-tapped.
- `OneShot.setSticky(key_addr)`: Put the key at `key_addr` in the "sticky" OneShot state. The key will be released by OneShot when it is tapped again.
- `OneShot.setOneShot(key_addr)`: Put the key at `key_addr` in the "one-shot" state. This is normally the state OneShot key will be in after it has been tapped. Calling `setPending()` is more likely to be useful.
- `OneShot.clear(key_addr)`: Clear the OneShot state of the key at `key_addr`.

Note: Any plugin that calls one of these OneShot methods must either be registered in `KALEIDOSCOPE_INIT_PLUGINS()` after OneShot, or it must add the `INJECTED` bit to the keyswitch state of the event (i.e. `event.state |= INJECTED`) to prevent OneShot from prematurely advancing keys to the next OneShot state.

SpaceCadet "no-delay" mode

SpaceCadet can now be enabled in "no-delay" mode, wherein the primary (modifier) value of the key will be sent to the host immediately when the key is pressed. If the SpaceCadet key is released before the timeout, the modifier is released, and then the alternate (symbol) value is sent. To activate "no-delay" mode, call `SpaceCadet.enableWithoutDelay()`.

New Qukeys features

Tap-repeat

It is now possible to get the "tap" value of a qukey to repeat (as if that key for that character was simply being held down on a normal keyboard) by tapping the qukey, then quickly pressing and holding it down. The result on the OS will be as if the key was pressed and held just once, so that users of macOS apps that use the Cocoa input system can get the menu for characters with diacritics without an extra character in the output.

The maximum interval between the two keypresses that will trigger a tap repeat can be configured via the `Qukeys.setMaxIntervalForTapRepeat(ms)` function, where the argument specifies the number of milliseconds Qukeys will wait after a qukey is tapped for it to be pressed a second time. If it is, but the qukey is released within that same interval from the first tap's release, it will be treated as a double-tap, and both taps will be sent to the OS.

New OneShot features

Auto-OneShot modifiers & layers

OneShot can now treat modifiers and layer-shift keys as automatic OneShot keys. This includes modifiers with other modifier flags applied, so it is now very simple to turn `Key_Meh` or `Key_Hyper` into a OneShot key. The feature is controlled by the following new functions:

- `OneShot.toggleAutoModifiers()`: Turn auto-OneShot modifiers on or off.
- `OneShot.toggleAutoLayers()`: Turn auto-OneShot layer shifts on or off.
- `OneShot.toggleAutoOneShot()`: Both of the above.

There are also `enable` and `disable` versions of these functions.

Note, it is still possible to define a modifier key in the keymap that will not automatically become a OneShot key when pressed, by applying modifier flags to `Key_NoKey` (e.g. `LSHIFT(Key_NoKey)`).

Two new special OneShot keys

OneShot can now also turn *any* key into a sticky key, using either of two special `Key` values that can be inserted in the keymap.

`OneShot_MetaStickyKey`

This is a special OneShot key (it behaves like other OneShot keys), but its effect is to make any key pressed while it is active sticky. Press `OneShot_MetaStickyKey`, then press `X`, and `X` will become sticky. Sticky keys can be deactivated just like other OneShot keys, by pressing them again. This works for any key value, so use it with caution.

`OneShot_ActiveStickyKey`

Like `OneShot_ActiveStickyKey`, this key makes other keys sticky, but rather than affecting a subsequent key, it affects any keys already held when it is pressed. Press `X`, press `OneShot_ActiveStickyKey`, and release `X`, and `X` will be sticky until it is pressed again to deactivate it. Again, it works on any key value, so use with caution.

LED-ActiveModColor highlighting

With the updates to OneShot, LED-ActiveModColor now recognizes and highlights OneShot keys in three different states (along with normal modifiers):

- one-shot (a key that's active after release, but will time out)
- sticky (a key that will stay active indefinitely after release)
- normal (a key that will stay active only while physically held; also applies to normal modifier keys)

The colors of theses three highlights are controlled by the properties `ActiveModColorEffect.oneshot_color`, `ActiveModColorEffect.sticky_color`, and `ActiveModColorEffect.highlight_color`, respectively.

Better protection against unintended modifiers from Qukeys

Qukeys has two new configuration options for preventing unintended modifiers in the output, particularly when typing fast:

- `Qukeys.setMinimumHoldTime(ms)` sets the minimum duration of a qukey press required for it to be eligible to take on its alternate (modifier) value.
- `Qukeys.setMinimumPriorInterval(ms)` sets the minimum interval between the previous printable (letters, numbers, and punctuation) key press and the press of the qukey required to make the qukey eligible to take on its alternate (modifier) value.

KALEIDOSCOPE_API_VERSION bump

`KALEIDOSCOPE_API_VERSION` has been bumped to **2** due to the plugin API changes mentioned below. It does not mean that version two of the API is final, though. The bump is there so plugins can check it, and make compile-time decisions based on it. Such as whether to compile for the version one, or for the version two API.

The API version will remain the same, even if we introduce breaking changes - until a stable release is made from the v2 branch. From that point onwards, the API version will change with further breaking changes.

New device API

A new hardware device API was introduced in November 2019, replacing the old system. It was designed to be more composable, more future proof than the old system. All hardware plugins under Keyboardio control have been updated to use the new APIs.

See UPGRADING.md for more information.

New plugin API

A new plugin API was introduced in May 2018, which replaces the old system. The new system is hopefully easier to use and develop for:

- It does not require one to register / use hooks anymore. Implementing the interface provided by `kaleidoscope::Plugin` is all that is required.
- The new system has more hook points, and the method names are much more clear now.

Plugins under Keyboardio control have all been updated to use the new API, and they no longer support the older one.

See UPGRADING.md for more information.

Transition to a monorepo

We heard a lot of complaints that while the plugin architecture of Kaleidoscope is great, having so many plugins scattered around in dozens of repositories is a huge barrier of entry for potential contributors, and a significant pain point for end-users to update. For these reasons and more, we merged almost all plugins into the Kaleidoscope repository.

While at first it may seem that this is a move towards a monolithic architecture, rest assured, it is not. The plugin APIs are still a core part of Kaleidoscope, it isn't going anywhere. We merely rearranged the sources, is all. Nothing else changes.

Some headers and names did change, however, see UPGRADING.md for more information.

Bidirectional communication for plugins

The bi-directional communication protocol formerly implemented by `Kaleidoscope-Focus` has been partially pulled into core, using the new plugin system mentioned above. The new system makes it a lot easier for both end-users and developers to use the feature.

See UPGRADING.md for more information.

Consistent timing

Numerous plugins use timers, most of them directly calling `millis()`. This has the disadvantage that calls within a main loop cycle will be inconsistent, which makes timing synchronization across plugins hard. The newly introduced `Kaleidoscope.millisAtCycleStart()` function helps dealing with this issue.

See UPGRADING.md for more information.

USB detach / attach

It is now possible to detach, and re-attach the USB link from/to the host, without resetting the device. The intent of this feature (as implemented by the `Kaleidoscope.detachFromHost()` and `Kaleidoscope.attachToHost()` methods) is to allow configuration changes without rebooting.

See the Kaleidoscope-USB-Quirks plugin for a use-case.

Finer stickability controls for OneShot

The *OneShot plugin* gained finer stickability controls, one can now control whether the double-tap stickiness is enabled on a per-key basis. See UPGRADING.md for more information.

A way to slow down Unicode input

In certain cases we need to delay the unicode input sequence, otherwise the host is unable to process the input properly. For this reason, the *Unicode* gained an `.input_delay()` method that lets us do just that. It still defaults to no delay.

Better support for modifiers in the Cycle plugin

The *Cycle* plugin has much better support for cycling through keys with modifiers applied to them, such as `LSHIFT(Key_A)`. Please see the documentation and the updated example for more information.

More control over when to send reports during Macro playback

There are situations where one would like to disable sending a report after each and every step of a macro, and rather have direct control over when reports are sent. The new `WITH_EXPLICIT_REPORT`, `WITH_IMPLICIT_REPORT` and `SEND_REPORT` steps help with that. Please see the *Macros* documentation for more information.

LED-ActiveModColor can be asked to not highlight normal modifiers

The plugin was intended to work with OneShot primarily, and that's where it is most useful. To make it less surprising, and more suitable to include it in default-like firmware, we made it possible to ask it not to highlight normal modifiers. Please see the *LED-ActiveModColor* documentation for more information.

Events now trigger on layer changes

Changing layers now triggers the `onLayerChange` event - but only if there was real change (thus, calling `Layer.on(SOME_LAYER)` multiple times in a row will only trigger one event). This event was introduced to help plugins that depend on layer state schedule their work better.

Hyper and Meh keys

To make it easier to create custom shortcuts, that do not interfere with system ones, an old trick is to use many modifiers. To make this easier, `Ctrl+Shift+Alt` is commonly abbreviated as `Meh`, while `Ctrl+Shift+Alt+Gui` is often called `Hyper`. To support this, we offer the `Key_Meh` and `Key_Hyper` aliases, along with `MEH(k)` and `HYPER(k)` to go with them.

11.18.2 `keymap` internals are now a one dimensional array

Historically, Kaleidoscope used the dimensional array `keymaps` to map between logical key position and hardware key position. `keymaps` has been replaced with `keymaps_linear`, which moves the keymap to a simple array. This makes it easier to support new features in Kaleidoscope and simplifies some code

11.18.3 `PER_KEY_DATA` macros

New `PER_KEY_DATA` and `PER_KEY_DATA_STACKED` macros are available (when defined by a hardware implementation). These macros make it easier to build features like `KEYMAPS` that track some data about each key on a keyboard.

11.18.4 New hardware support

Kaleidoscope has been ported to the following devices:

- Atreus: All known variants of the original Atreus are supported. From the Legacy Teensy variant, through the pre-2016 PCB with an A* MCU, the post-2016 PCB, and FalbaTech's handwired one too. Apart from the legacy Teensy variant, the other support both the A* or a Teensy as an MCU.
- ErgoDox: Originally developed to support the ErgoDox EZ, but all other compatible hardware is supported, such as the original ErgoDox and anything else wired like it, like some Dactyls.
- Planck: AVR-based Plancks, and anything else wired similarly should be supported, as long as they use a Teensy.
- Splitography: Initial support for the Splitography Steno keyboard.

For more information, please see the hardware plugins' documentation.

To make it easier to port Kaleidoscope, we introduced the `ATMegaKeyboard` base class. For any board that's based on the ATMega MCU and a simple matrix, this might be a good foundation to develop the hardware plugin upon.

11.18.5 New plugins

CharShift

The *CharShift* plugin allows independent assignment of symbols to keys depending on whether or not a `shift` key is held.

AutoShift

The *AutoShift* plugin provides an alternative way to get shifted symbols, by long-pressing keys instead of using a separate `shift` key.

DynamicMacros

The *DynamicMacros* plugin provides a way to use and update macros via the Focus API, through Chrysalis.

IdleLEDs

The *IdleLEDs* plugin is a simple, yet, useful one: it will turn the keyboard LEDs off after a period of inactivity, and back on upon the next key event.

LEDActiveLayerColor

The [LEDActiveLayerColor][plugins/Kaleidoscope-LEDActiveLayerColor.md] plugin makes it possible to set the color of all LEDs to the same color, depending on which layer is active topmost.

LED-Wavepool

We integrated the *LEDWavepool* plugin by ToyKeeper, with a few updates and new features added.

Turbo

The *Turbo* plugin provides a way to send keystrokes in very quick succession while holding down a key.

WinKeyToggle

The *WinKeyToggle* plugin assists with toggling the Windows key on and off - a little something for those of us who game under Windows and are tired of accidentally popping up the start menu.

FirmwareDump

The *FirmwareDump* plugin makes it possible to dump one's firmware over Focus.

11.18.6 Breaking changes

Implementation of type Key internally changed from C++ union to class

Type `Key` was originally implemented as a C++ union. For technical reasons it had to be converted to a C++ class. This implies that the double usage of the original union, holding either raw data (member `raw`) or key code/key flags data (members `keyCode` and `flags`) is no more possible.

Direct use of member `raw` will emit a diagnostic compiler message but will cause the firmware linking process to fail. For a deprecation periode `keyCode` and `flags` keep on being supported but will cause deprecation warnings during compile.

Please see the relevant upgrade notes for information about how to upgrade legacy code.

`LEDControl.paused` has been deprecated

The `.paused` property of `LEDControl` has been deprecated in favour of the new `LEDControl.disable()` and `LEDControl.enable()` methods. These two will turn off or refresh the LEDs, respectively, along with disabling or re-enabling future updates and syncs.

The `NumPad` plugin no longer toggles `NumLock`

The `NumPad` plugin used to toggle `NumLock` when switching to the NumPad layer. This caused issues on OSX where `NumLock` is interpreted as `Clear`. For this reason, the plugin no longer does this. As a consequence, everyone's encouraged to update their keymap so that the numpad layer uses normal number keys instead of the keypad numbers. See Model01-Firmware#79 for an example about how to do this.

The `RxCy` macros and peeking into the keyswitch state

The `RxCy` macros changed from being indexes into a per-hand bitmap to being an index across the whole keyboard. This mostly affected the *MagicCombo* plugin.

Please see the relevant upgrade notes for more information.

The `Redial` plugin had a breaking API change

The *Redial* plugin was simplified, one no longer needs to define `Key_Redial` on their own, the plugin defines it itself. See the upgrade notes for more information about how to upgrade.

Color palette storage has changed

The *LED-Palette-Theme* had to be changed to store the palette colors in reverse. This change had to be made in order to not default to a bright white palette, that would draw so much power that most operating systems would disconnect the keyboard due to excessive power usage. With inverting the colors, we now default to a black palette instead. This sadly breaks existing palettes, and you will have to re-set the colors.

We also changed when we reserve space for the palette in EEPROM: we used to do it as soon as possible, but that made it impossible to go from a firmware that does not use the plugin to one that does, and still have a compatible EEPROM layout. We now reserve space as late as possible. This breaks existing EEPROM layouts however.

EEPROM-Keymap changed Focus commands

The *EEPROMKeymap* plugin was changed to treat built-in (default) and EEPROM-stored (custom) layers separately, because that's less surprising, and easier to work with from Chrysalis. The old `keymap.map` and `keymap.roLayers` commands are gone, the new `keymap.default` and `keymap.custom` commands should be used instead.

EEPROMSettings' version() setter has been deprecated

We're repurposing the `version` setting: instead of it being something end-users can set, we'll be using it internally to track changes made to *EEPROMSettings* itself, with the goal of allowing external tools to aid in migrations. The setting wasn't widely used - if at all -, which is why we chose to repurpose it instead of adding a new field.

Key masking has been deprecated

Key masking was a band-aid introduced to avoid accidentally sending unintended keys when key mapping changes between a key being pressed and released. Since the introduction of keymap caching, this is no longer necessary, as long as we can keep the mapping consistent. Users of key masking are encouraged to find ways to use the caching mechanism instead.

11.18.7 Bugfixes

We fixed way too many issues to list here, so we're going to narrow it down to the most important, most visible ones.

Support for BIOSes, EFI, login prompts, etc

Keyboards report keys pressed to the host via either of two protocols: the boot protocol, or the report protocol. The boot protocol is the simpler, and it is what older BIOSes, EFI, and certain OS login prompts (or hard disk password prompts and the like) require. Until recently, the firmware wasn't able to provide this protocol, only the more advanced report one, which is required for N-key roll-over.

We now support the boot protocol, and on operating systems that fully conform to the USB specification, this works automatically. For all others, one can implement a way to force one mode or the other. See the factory firmware for an example how to achieve this.

As the firmware evolves, there are - and will be - APIs that we deprecate, and eventually remove. We are constantly adding new features and plugins too.

This document lists noteworthy new features for the *current* release, with examples of use. Another section provides a short guide for upgrading from deprecated APIs. For deprecations, their planned removal date is also listed.

If any of this does not make sense to you, or you have trouble updating your .ino sketch or custom plugins, do not hesitate to write us at help@keyboard.io, we can help you fix it.

11.19 Upgrade notes

As a matter of policy, we try hard to give you at least 60 days notice before we permanently remove or break any API we've included in a release. Typically, this means that any code that uses the old API will emit a warning when compiled with a newer version of Kaleidoscope. In all cases, this document should explain how to update your code to use the new API.

11.19.1 New features

New event handler

One more `KeyEvent` handler has been added: `afterReportingState(const KeyEvent &event)`. This handler gets called after HID reports are sent for an event, providing a point for plugins to act after an event has been fully processed by `Runtime.handleKeyEvent()`.

Event-driven main loop

Kaleidoscope's main loop has been rewritten. It now responds to key toggle-on and toggle-off events, dealing with one event at a time (and possibly more than one in a given cycle). Instead of sending a keyboard HID report at the end of every scan cycle (and letting the HID module suppress duplicates), it now only sends HID reports in response to input events.

Furthermore, there are now two functions for initiating the processing of key events:

- `Runtime.handleKeyswitchEvent()` is the starting point for events that represent physical keyswitches toggling on or off.
- `Runtime.handleKeyEvent()` is the starting point for "artificial" key events. It is also called at the end of `handleKeyswitchEvent()`. In general, if a plugin needs to generate a key event, it should call `handleKeyEvent()`, not `handleKeyswitchEvent()`.

Each of the above functions calls its own set of plugin event handlers. When those event handlers are all done, event processing continues as `handleKeyEvent()` prepares a new keyboard HID report, then sends it:

- `Runtime.prepareKeyboardReport()` first clears the HID report, then populates it based on the contents of the `live_keys[]` array. Note that the HID report is not cleared until *after* the new plugin event handlers have been called.
- `Runtime.sendKeyboardReport()` handles generating extra HID reports required for keys with keyboard modifier flags to avoid certain bugs, then calls a new plugin event handler before finally sending the new HID report. These functions should rarely, if ever, need to be called by plugins.

The `KeyEvent` data type

There is a new `KeyEvent` type that encapsulates all the data relevant to a new key event, and it is used as the parameter for the new event-handling functions.

- `event.addr` contains the `KeyAddr` associated with the event.
- `event.state` contains the state bitfield (`uint8_t`), which can be tested with `keyToggledOn()`/`keyToggledOff()`.
- `event.key` contains a `Key` value, usually looked up from the keymap.
- `event.id` contains a pseudo-unique ID number of type `KeyEventId` (an 8-bit integer), used by certain plugins (see `onKeyswitchEvent()` below).

New plugin event handlers

`onKeyswitchEvent(KeyEvent &event)`

`onKeyEvent(KeyEvent &event)`

`onAddToReport(Key key)`

`beforeReportingState(const KeyEvent &event)`

For end-users

Existing sketches should be mostly backwards-compatible, but some updates will be needed for sketches that use custom code. In particular, users of the Macros plugin are likely to need to make adjustments to the code in the user-defined `macroAction()` function, including that function's signature, the new version of which takes a `KeyEvent` parameter instead of just an event state value. In most cases, this will make the resulting code more straightforward without any loss of functionality.

In addition to Macros, these changes might also affect user-defined code executed by the TapDance, Leader, and Syster plugins. Please see the documentation and examples for the affected plugins for details.

Keyboard State array

The keymap cache (`Layer_::live_composite_keymap_[]`) has been replaced by a keyboard state array (`kaleidoscope::live_keys[]`). The top-level functions that handle keyswitch events have been updated to treat this new array as a representation of the current state of the keyboard, with corresponding `Key` values for any keys that are active (physically held or activated by a plugin).

For end-users

There should be no user-visible changes for anyone who simply uses core plugins. A few functions have been deprecated (`Layer.eventHandler()` & `Layer.updateLiveCompositeKeymap()`), but there are straightforward replacements for both.

For developers

The major changes are to the `handleKeyswitchEvent()` function, which has been reorganized in order to update the new keyboard state array with correct values at the appropriate times. In addition to that, two new facilities are available:

EventHandlerResult::ABORT

This is a new return value available to plugin event handlers, which is similar to EVENT_CONSUMED in that it causes the calling hook function to return early (stopping any subsequent handlers from seeing the event), but is treated differently by handleKeyswitchEvent(). If a handler returns EVENT_CONSUMED, the keyboard state array will still be updated by handleKeyswitchEvent(), but if it returns ABORT, it will not. In both cases, no further event processing will be done by the built-in event handler.

live_keys[key_addr]

This is the new facility for checking the value of an entry in the keyboard state array. It is indexed directly by KeyAddr values, without the need to convert them to integers first. For example, it could be used in a range-based for loop to check for values of interest:

```
for (KeyAddr key_addr : KeyAddr::all()) {
  Key key = live_keys[key_addr];
  if (key == Key_LeftShift || key == Key_RightShift) {
    // do something special...
  }
}
```

Additionally, if the KeyAddr values are not needed, one can use the iterator from the new KeyMap class like so:

```
for (Key key : live_keys.all()) {
  if (key == Key_X) {
    // do something special...
  }
}
```

The live_keys object's subscript operator can also be used to set values in the keyboard state array:

```
live_keys[key_addr] = Key_X;
```

It also comes with several convenience functions which can be used to make the intention of the code clear:

```
// Set a value in the keyboard state array to a specified Key value:
live_keys.activate(key_addr, Key_X);

// Set a value to Key_Inactive, deactivating the key:
live_keys.clear(key_addr);

// Set all values in the array to Key_Inactive:
live_keys.clear();)

// Set a value to Key_Masked, masking the key until its next release event:
live_keys.mask(key_addr);
```

In most cases, it won't be necessary for plugins or user sketches to call any of these functions directly, as the built-in event handler functions will manage the keyboard state array automatically.

New build system

In this release, we replace kaleidoscope-builder with a new Makefile based build system that uses `arduino-cli` instead of of the full Arduino IDE. This means that you can now check out development copies of Kaleidoscope into any directory, using the `KALEIDOSCOPE_DIR` environment variable to point to your installation.

New device API

We are introducing - or rather, replacing - the older hardware plugins, with a system that's much more composable, more extensible, and will allow us to better support new devices, different MCUs, and so on.

For end-users

For end users, this doesn't come with any breaking changes. A few things have been deprecated (`ROWS`, `COLS`, `LED_COUNT`, `KeyboardHardware`), but they still function for the time being.

For developers

For those wishing to port Kaleidoscope to devices it doesn't support yet, the new API should make most things considerably easier. Please see the documentation in *device-apis.md*.

The old symbols and APIs are no longer available.

New plugin API

For end-users

With the next version of `Kaleidoscope`, we are introducing a new plugin API. It's more efficient, smaller, and uses less resources than the previous one, while being more extensible, and a lot easier to use as well. But all of this matters little when one's not all that interested in writing plugins. However, the new plugin API comes with breaking changes, and one will need to update their own sketch too.

To achieve all of the above, we had to change how plugins are initialized. Instead of using `Kaleidoscope.use()` in the `setup()` method of one's sketch, the plugins must now be initialized with `KALEIDOSCOPE_INIT_PLUGINS()`, outside of the `setup()` method. While `use()` was expecting pointers (`&Plugin`), `_INIT_PLUGINS()` expects references (`Plugin`).

The conversion should be simple, and all of the official plugins have been updated already to use the new API, so they're safe to use this way. Some third-party plugins may still use the older API, they will need to be updated.

To make things clear, here's an example of how to migrate from the old way to the new:

```
// Old way
void setup() {
  Kaleidoscope.use(&LEDControl,
                   &Macros,
                   &OneShot,
                   &MouseKeys,
                   &LEDOff,
                   &LEDRainbowEffect);
  Kaleidoscope.setup();
```

```
// New way
KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          Macros,
                          OneShot,
                          MouseKeys,
                          LEDOff,
                          LEDRainbowEffect);
void setup() {
  Kaleidoscope.setup();
}
```

One thing to keep in mind is that with the old interface, plugins were able to automatically pull in their dependencies. This is not possible with the new interface, and one is required to initialize dependencies manually. Please consult the documentation of the plugins you use, to discover their dependencies - if any - and add them to the list if need be. You only need to add each dependency once.

For developers

Developing plugins should be considerably simpler now, there is no need to register hooks, just implement the parts of the `kaleidoscope::Plugin` interface that make sense for a particular plugin.

In practice, this boils down to implementing one or more of the following hook points:

- `onSetup()`: Called once during device bootup, at the end of the `setup()` method. It takes no arguments, and must return `kaleidoscope::EventHandlerResult::OK`.
- `beforeEachCycle()`: Called once, at the beginning of each cycle of the main loop. This is similar to the old "loop hook" with its `post_clear` argument set to false. Takes no arguments, must return `kaleidoscope::EventHandlerResult::OK`.
- `onKeyswitchEvent`: Called for every non-idle key event. This replaces the old "event handler hook". It takes a key reference, a key address, and a key state. The key reference can be updated to change the key being processed, so that any plugin that processes it further, will see the updated key. Can return `kaleidoscope::EventHandlerResult::OK` to let other plugins process the event further, or `kaleidoscope::EventHandlerResult::EVENT_CONSUMED` to stop processing.
- `onFocusEvent`: Used to implement *bi-directional communication*. This is called whenever the firmware receives a command from the host. The only argument is the command name. Can return `kaleidoscope::EventHandlerResult::OK` to let other plugins process the event further, or `kaleidoscope::EventHandlerResult::EVENT_CONSUMED` to stop processing.
- `onNameQuery`: Used by the *Focus* plugin, when replying to a `plugins` command. Should either send the plugin name, or not be implemented at all, if the host knowing about the plugin isn't important.
- `beforeReportingState`: Called without arguments, just before sending the keyboard and mouse reports to the host. Must return `kaleidoscope::EventHandlerResult::OK`.
- `afterEachCycle`: Called without arguments at the very end of each cycle. This is the replacement for the "loop hook" with its `post_clear` argument set.

Bidirectional communication for plugins

For end-users

Whereas one would have used `Focus.addHook()` to register a new focus command, with the new architecture, one needs to add the object implementing the command to their list of plugins in `KALEIDOSCOPE_INIT_PLUGINS()`. A number of plugins that used to provide optional Focus commands now provide them by default. Some still provide optional ones, and we must transition to the new way.

For example, where one would have written the following before:

```
Focus.addHook(FOCUS_HOOK_LEDCONTROL);
```

. . . we need to add the appropriate object to the plugin list:

```
KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          FocusLEDCommand)
```

For developers

Upgrading from `Focus` to `onFocusEvent` and `FocusSerial` is a reasonably simple process, the interface is quite similar. Nevertheless, we present a step-by-step guide here, covering two use cases: one where we wish to always provide a Focus command when both our plugin and `FocusSerial` are enabled; and another where we only wish to provide the command when explicitly asked to do so.

The most trivial example

The biggest difference between `Focus` and `onFocusEvent` is that the former required explicit registering of hooks, while the latter does it automatically: every plugin that implements the `onFocusEvent` method will be part of the system. As a consequence, only plugins are able to supply new commands: there is no explicit registration, thus, no way to inject a command that isn't part of a plugin. This also means that these functions now return `kaleidoscope::EventHandlerResult` instead of `bool`. Furthermore, with `FocusSerial`, all communication is expected to go through it, instead of using `Serial` directly. Lets see a trivial example!

Focus

```
bool exampleFocusHook(const char *command) {
  if (strcmp_P(command, PSTR("example")) != 0)
    return false;

  Serial.println(F("This is an example response. Hello world!"));

  return true;
}

KALEIDOSCOPE_INIT_PLUGINS(Focus)

void setup() {
  Serial.begin(9600);
  Kaleidoscope.setup();
```

(continues on next page)

(continued from previous page)

```
  Focus.addHook(FOCUS_HOOK(exampleFocusHook, "example"));
}
```

onFocusEvent

```
namespace kaleidoscope {
class FocusExampleCommand : public Plugin {
 public:
  FocusExampleCommand() {}

  EventHandlerResult onNameQuery() {
    return ::Focus.sendName(F("FocusExampleCommand"));
  }

  EventHandlerResult onFocusEvent(const char *input) {
    const char *cmd = PSTR("example");

    if (::Focus.inputMatchesHelp(input))
      return ::Focus.printHelp(cmd);

    if (!::Focus.inputMatchesCommand(input, cmd))
      return EventHandlerResult::OK;

    ::Focus.send(F("This is an example response. Hello world!"));
    return EventHandlerResult::EVENT_CONSUMED;
  }
};
}

kaleidoscope::FocusExampleCommand FocusExampleCommand;

KALEIDOSCOPE_INIT_PLUGINS(Focus, FocusExampleCommand);

void setup() {
  Kaleidoscope.setup();
}
```

Summary

The new version is slightly more verbose for the trivial use case, because we have to wrap it up in an object. But other than that, the changes are minimal, and we don't need to explicitly register it!

Observe that the return values changed: with `Focus`, if we wanted other hooks to have a chance at processing the same command, the hook returned `false`; if we wanted to stop processing, and consider it consumed, it returned `true`. With the new system, this is more descriptive with the `EventHandlerResult::OK` and `EventHandlerResult::EVENT_CONSUMED` return values.

A stateful example

Perhaps a better example that shows the quality of life improvements the new system brings is the case where the command needs access to either plugin state, or plugin methods. With the former system, the focus hooks needed to be static methods, and needed to be public. This is not necessarily the case now, because `onFocusEvent` is a non-static object method. It has full access to plugin internals!

Focus

```
namespace kaleidoscope {
class ExamplePlugin : public Plugin {
 public:
  ExamplePlugin();

  static bool exampleToggle() {
    example_toggle_ = !example_toggle_;
    return example_toggle_;
  }

  static bool focusHook(const char *command) {
    if (strcmp_P(command, PSTR("example.toggle")) != 0)
      return false;

    ::Focus.printBool(exampleToggle());
    return true;
  }

 private:
  static bool example_toggle_;
};
}

kaleidoscope::ExamplePlugin ExamplePlugin;

KALEIDOSCOPE_PLUGIN_INIT(Focus, ExamplePlugin)

void setup() {
  Serial.begin(9600);
  Kaleidoscope.setup();

  Focus.addHook(FOCUS_HOOK(ExamplePlugin.focusHook, "example.toggle"));
}
```

onFocusEvent

```
namespace kaleidoscope {
class ExamplePlugin : public Plugin {
 public:
  ExamplePlugin();

  EventHandlerResult onFocusEvent(const char *input) {
    if (!::Focus.inputMatchesCommand(input, PSTR("example.toggle")))
      return EventHandlerResult::OK;

    example_toggle_ = !example_toggle_;
    ::Focus.send(example_toggle_);

    return EventHandlerResult::EVENT_CONSUMED;
  }

 private:
  static bool example_toggle_;
};
}

kaleidoscope::ExamplePlugin ExamplePlugin;

KALEIDOSCOPE_PLUGIN_INIT(Focus, ExamplePlugin)

void setup() {
  Kaleidoscope.setup();
}
```

Summary

It's just another plugin, with just another event handler method implemented, nothing more. No need to explicitly register the focus hook, no need to provide access to private variables - we can just keep them private.

Optional commands

Optional commands are something that were perhaps easier with the `Focus` method: one just didn't register them. With `onFocusEvent`, we need to do a bit more than that, and move the command to a separate plugin, if we do not wish to enable it in every case. This adds a bit of overhead, but still less than `Focus` did.

Focus

```
bool exampleOptionalHook(const char *command) {
  if (strcmp_P(command, PSTR("optional")) != 0)
    return false;

  Serial.println(Layer.getLayerState(), BIN);
  return true;
}

KALEIDOSCOPE_INIT_PLUGINS(Focus)

void setup() {
  Kaleidoscope.setup();
}
```

Do note that we do not register the `exampleOptionalHook` here! As such, because it is unused code, it will get optimized out during compilation. While this is a simplistic example, the optional hook might have been part of a class, that provides other hooks.

onFocusEvent

```
namespace kaleidoscope {
class ExampleOptionalCommand : public Plugin {
 public:
  ExampleOptionalCommand() {}

  EventHandlerResult onFocusEvent(const char *input) {
    if (!::Focus.inputMatchesCommand(input, PSTR("optional")))
      return EventHandlerResult::OK;

    ::Focus.send(Layer.getLayerState());
    return EventHandlerResult::EVENT_CONSUMED;
  }
};
}

KALEIDOSCOPE_INIT_PLUGINS(Focus)

void setup() {
  Kaleidoscope.setup();
}
```

Summary

The trick here is to move optional commands out into a separate plugin. It's a bit more boilerplate, but not by much.

Consistent timing

As an end-user, there's nothing one needs to do. Consistent timing helpers are primarily for plugin use.

As a developer, one can continue using `millis()`, but migrating to `Kaleidoscope.millisAtCycleStart()` is recommended. The new method will return the same value for the duration of the main loop cycle, making time-based synchronization between plugins a lot easier.

11.19.2 Breaking changes

Sketch preprocssing system

We used to support the ability to amend all compiled sketches by adding code to `src/kaleidoscope_internal/sketch_preprocessing/sketch_header.h` and `src/kaleidoscope_internal/sketch_preprocessing/sketch_footer.h`. The functionality was never used by Kaleidoscope itself and frequently pulled the (empty) header files from the wrong copy of Kaleidoscope. If you need this functionality, please open a GitHub issue.

Macros

This is a guide to upgrading existing Macros code to use the new version of Kaleidoscope and the Macros plugin.

New `macroAction()` function

There is a new version of the `macroAction()` function, which is the entry point for user-defined Macros code. The old version takes two integer parameters, with the following call signature:

```
const macro_t* macroAction(uint8_t macro_id, uint8_t key_state)
```

If your sketch has this function, with a `key_state` bitfield parameter, it might still work as expected, but depending on the specifics of the code that gets called from it, your macros might not work as expected. Either way, you should update that function to the new version, which takes a `KeyEvent` reference as its second parameter:

```
const macro_t* macroAction(uint8_t macro_id, KeyEvent &event)
```

For simple macros, it is a simple matter of replacing `key_state` in the body of the `macroAction()` code with `event.state`. This covers most cases where all that's done is a call to `Macros.type()`, or a `MACRO()` or `MACRODOWN()` sequence is returned.

Using MACRO() and MACRODOWN()

The preprocessor macro `MACRODOWN()` has been deprecated, because the event handler for Macros is no longer called every cycle, but only when a key is either pressed or released. Instead of using `return MACRODOWN()`, you should test for a toggle-on event in `macroAction()` and use `MACRO()` instead. If you previously had something like the following in your `macroAction()` function:

```
switch(macro_id) {
case MY_MACRO:
  return MACRODOWN(T(X), T(Y), T(Z));
}
```

. . . you should replace that with:

```
switch(macro_id) {
case MY_MACRO:
  if (keyToggledOn(event.state))
    return MACRO(T(X), T(Y), T(Z));
}
```

. . . or, for a group of macros that should only fire on keypress:

```
if (keyToggledOn(event.state)) {
  switch(macro_id) {
  case MY_MACRO:
    return MACRO(T(X), T(Y), T(Z));
  case MY_OTHER_MACRO:
    return MACRO(T(A), T(B), T(C));
  }
}
```

Releasing keys with Macros.release() or U()/Ur()/Uc()

Macros now operates by manipulating keys on a small supplemental virtual keyboard when using `Macros.press()` and `Macros.release()` (which are called by `D()` and `U()`, *et al*, respectively). This means that it has no built-in facility for releasing other keys that are held on the keyboard. For example, if you had a Macro that removed `shift` keycodes from the HID report in the past, it won't work. For example:

```
  case KEY_COMMA:
    if (keyToggledOn(event.state)) {
      if (Kaleidoscope.hid().keyboard().wasModifierKeyActive(Key_LeftShift)) {
        return MACRO(U(LeftShift), T(Comma), D(LeftShift));
      } else {
        return MACRO(T(M));
      }
    }
```

In this case, holding a physical `Key_LeftShift` and pressing `M(KEY_COMMA)` will not cause the held `shift` to be released, and you'll get a < instead of the intended `,` (depending on the OS keymap). To accomplish this, you'll need a small plugin like the following in your sketch:

```
namespace kaleidoscope {
namespace plugin {

// When activated, this plugin will suppress any `shift` key (including modifier
// combos with `shift` a flag) before it's added to the HID report.
class ShiftBlocker : public Plugin {

 public:
  EventHandlerResult onAddToReport(Key key) {
    if (active_ && key.isKeyboardShift())
      return EventHandlerResult::ABORT;
    return EventHandlerResult::OK;
  }

  void enable() {
    active_ = true;
  }
  void disable() {
    active_ = false;
  }

 private:
  bool active_{false};

};

} // namespace plugin
} // namespace kaleidoscope

kaleidoscope::plugin::ShiftBlocker ShiftBlocker;
```

You may also need to define a function to test for held `shift` keys:

```
bool isShiftKeyHeld() {
  for (Key key : kaleidoscope::live_keys.all()) {
    if (key.isKeyboardShift())
      return true;
  }
  return false;
}
```

Then, in your `macroAction()` function:

```
  if (keyToggledOn(event.state)) {
    switch (macro_id) {
    case MY_MACRO:
      if (isShiftKeyHeld()) {
        ShiftBlocker.enable();
        Macros.tap(Key_Comma);
        ShiftBlocker.disable();
      } else {
        Macros.tap(Key_M);
      }
```

(continues on next page)

(continued from previous page)

```
      return MACRO_NONE;
    }
  }
```

In many simple cases, such as the above example, an even better solution is to use the CharShift plugin instead of Macros.

Code that calls `handleKeyswitchEvent()` or `pressKey()`

It is very likely that if you have custom code that calls `handleKeyswitchEvent()` or `pressKey()` directly, it will no longer function properly after upgrading. To adapt this code to the new `KeyEvent` system requires a deeper understanding of the changes to Kaleidoscope, but likely results in much simpler Macros code.

The first thing that is important to understand is that the `macroAction()` function will now only be called when a Macros `Key` toggles on or off, not once per cycle while the key is held. This is because the new event handling code in Kaleidoscope only calls plugin handlers in those cases, dealing with one event at a time, in a single pass through the plugin event handlers (rather than one pass per active key)–and only sends a keyboard HID report in response to those events, not once per scan cycle.

This means that any Macros code that is meant to keep keycodes in the keyboard HID report while the Macros key is held needs to be changed. For example, if a macro contained the following code:

```
if (keyIsPressed(key_state)) {
  Runtime.hid().keyboard().pressKey(Key_LeftShift);
}
```

…that wouldn't work quite as expected, because as soon as the next key is pressed, a new report would be generated without ever calling `macroAction()`, and therefore that change to the HID report would not take place, effectively turning off the `shift` modifier immediately before sending the report with the keycode that it was intended to modify.

Furthermore, that `shift` modifier would never even get sent in the first place, because the HID report no longer gets cleared at the beginning of every cycle. Now it doesn't get cleared until *after* the plugin event handlers get called (in the case of Macros, that's `onKeyEvent()`, which calls the user-defined `macroAction()` function), so any changes made to the HID report from that function will be discarded before it's sent.

Instead of the above, there are two new mechanisms for keeping keys active while a Macros key is pressed:

Alter the `event.key` value

If your macro only needs to keep a single `Key` value active after running some code, and doesn't need to run any custom code when the key is released, the simplest thing to do is to override the event's `Key` value:

```
if (keyToggledOn(event.state)) {
  // do some macro action(s)
  event.key = Key_LeftShift;
}
```

This will (temporarily) replace the Macros key with the value assigned (in this case, `Key_LeftShift`), starting immediately after the `macroAction()` function returns, and lasting until the key is released. This key value can include modifier flags, or it can be a layer-shift, or any other valid `Key` value (though it won't get processed by plugins that are initialized before Macros in `KALEIDOSCOPE_INIT_PLUGINS()`, and Macros itself won't act on the value, if it gets replaced by a different Macros key).

Use the supplemental Macros `Key` array

The Macros plugin now contains a small array of `Key` values that will be included when building HID reports triggered by subsequent, non-Macros events. To use it, just call one (or more) of the following methods:

```
Macros.press(key);
Macros.release(key);
Macros.tap(key)
```

Each one of these functions generates a new artificial key event, and processes it (including sending a HID report, if necessary). For `press()` and `release()`, it also stores the specified key's value in the Macros supplemental `Key` array. In the case of the `tap()` function, it generates matching press and release events, but skips storing them, assuming that no plugin will generate an intervening event. All of the events generated by these functions will be marked `INJECTED`, which will cause Macros itself (and many other plugins) to ignore them.

This will allow you to keep multiple `Key` values active while a Macros key is held, while leaving the Macros key itself active, enabling more custom code to be called on its release. Note that whenever a Macros key is released, the supplemental key array is cleared to minimize the chances of keycodes getting "stuck". It is still possible to write a macro that will cause values to persist in this array, however, by combining both a sequence that uses key presses without matched releases *and* replacing `event.key` (see above) in the same macro.

Borrow an idle key (not recommended)

It's also possible to "borrow" one (or more) idle keys on the keyboard by searching the `live_keys[]` array for an empty entry, and generating a new event with the address of that key. This is not recommended because surprising things can happen if that key is then pressed and released, but it's still an option for people who like to live dangerously.

Code that calls `sendReport()`

Calling `sendReport()` directly from a macro is now almost always unnecessary. Instead, a call to `Runtime.handleKeyEvent()` will result in a keyboard HID report being sent in response to the generated event without needing to make it explicit.

Code that uses `Macros.key_addr`

This variable is deprecated. Instead, using the new `macroAction(id, event)` function, the address of the Macros key is available via the `event.addr` variable.

Working with other plugins

Plugin-specific `Key` values

When the the Macros plugin generates events, it marks the event state as `INJECTED` in order to prevent unbounded recursion (Macros ignores injected events). This causes most other plugins to ignore the event, as well. Therefore, including a plugin-specific key (e.g. a OneShot modifier such as `OSM(LeftAlt)`) will most likely be ignored by the target plugin, and will therefore not have the desired effect. This applies to any calls to `Macros.play()` (including returning `MACRO()` from `macroAction()`), `Macros.tap()`, `Macros.press()`, and `Macros.release()`.

Physical event plugins

Macros cannot usefully produce events handled by plugins that implement the `onKeyswitchEvent()` handler, such as Qukeys, TapDance, and Leader. To make those plugins work with Macros, it's necessary to have the other plugin produce a Macros key, not the other way around. A `macroAction()` function must not call `Runtime.handleKeyswitchEvent()`.

OneShot

This is one plugin that you might specifically want to use with a macro, generally at the end of a sequence. For example, a macro for ending one sentence and beginning the next one might print a period followed by a space (`. `), then a OneShot shift key tap, so that the next character will be automatically capitalized. The problem, as mentioned before is that the following won't work:

```
MACRO(Tc(Period), Tc(Spacebar), Tr(OSM(LeftShift)))
```

…because OneShot will ignore the `INJECTED` event. One solution is to change the value of `event.key`, turning the pressed Macros key into a OneShot modifier. This will only work if Macros is registered before OneShot in `KALEIDOSCOPE_INIT_PLUGINS()`:

```
const macro_t* macroNewSentence(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    event.key = OSM(LeftShift);
    return MACRO(Tc(Period), Tc(Spacebar));
  }
  return MACRO_NONE;
}
```

A more robust solution is to explicitly call `Runtime.handleKeyEvent()`, but this is more complex, because you'll need to prevent the Macros key from clobbering the OneShot key in the `live_keys[]` array:

```
void macroNewSentence(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    Macros.tap(Key_Period);
    Macros.tap(Key_Spacebar);
    event.key = OSM(LeftShift);
    kaleidoscope::Runtime.handleKeyEvent(event);
    // Last, we invalidate the current event's key address to prevent the Macros
    // key value from clobbering the OneShot shift.
    event.key = Key_NoKey;
    event.addr.clear();
  }
}
```

Removed `kaleidoscope-builder`

`kaleidoscope-builder` has been removed.

We replaced it with a new Makefile based build system that uses `arduino-cli` instead of of the full Arduino IDE. This means that you can now check out development copies of Kaleidoscope into any directory, using the `KALEIDOSCOPE_DIR` environment variable to point to your installation.

OneShot meta keys

The special OneShot keys `OneShot_MetaStickyKey` & `OneShot_ActiveStickyKey` are no longer handled by the OneShot plugin directly, but instead by a separate OneShotMetaKeys plugin. If you use these keys in your sketch, you will need to add the new plugin, and register it after OneShot in `KALEIDOSCOPE_INIT_PLUGINS()` for those keys to work properly.

Repository rearchitecture

To improve build times and to better highlight Kaleidoscope's many plugins, plugins have been move into directories inside the Kaleidoscope directory.

The "breaking change" part of this is that git checkouts of Kaleidoscope are no longer directly compatible with the Arduino IDE, since plugins aren't in a directory the IDE looks in. They are, of course, visible to tools using our commandline build infrastructure / Makefiles.

When we build releases, those plugins are moved into directories inside the arduino platform packages for each architecture to make them visible to the Arduino IDE.

Layer system switched to activation order

The layer system used to be index-ordered, meaning that we'd look keys up on layers based on the *index* of active layers. Kaleidoscope now uses activation order, which looks up keys based on the order of layer activation.

The following functions have been removed as of **2021-01-01**:

- `Layer.top()`, which used to return the topmost layer index. Use `Layer.mostRecent()` instead, which returns the most recently activated layer. Until removed, the old function will return the most recent layer.
- `Layer.deactivateTop()`, which used to return the topmost layer index. Use `Layer.deactivateMostRecent()` instead. The old function will deactivate the most recent layer.
- `Layer.getLayerState()`, which used to return a bitmap of the active layers. With activation-order, a simple bitmap is not enough. For now, we still return the bitmap, but without the ordering, it is almost useless. Use `Layer.forEachActiveLayer()` to walk the active layers in order (from least recent to most).

For end-users

This is a breaking change only if your code accesses the member `raw` of type `Key` directly, for instance in a construct like

```
Key k;
k.raw = Key_A.raw;
```

This can easily be fixed by replacing read access to `Key::raw` with `Key::getRaw()` and write access with `Key::setRaw(...)`.

```
Key k;
k.setRaw(Key_A.getRaw());
```

Moreover, the compiler will still emit warnings in places of the code where members `keyCode` and `flags` of the original type `Key` are used, like e.g.

```
Key k;
k.keyCode = Key_A.keyCode;
k.flags = Key_A.flags;
```

These warnings can be also resolved by using the appropriate accessor methods `Key::getKeyCode()`/`Key::setKeyCode()` and `Key::getFlags()`/`Key::setKlags()` instead.

```
Key k;
k.setKeyCode(Key_A.getKeyCode());
k.setFlags(Key_A.getFlags());
```

The `RxCy` macros and peeking into the keyswitch state

The `RxCy` macros changed from being indexes into a per-hand bitmap to being an index across the whole keyboard. This means they can no longer be `or`-ed together to check against the keyswitch state of a given hand. Instead, the `kaleidoscope::hid::getKeyswitchStateAtPosition()` method can be used to check the state of a keyswitch at a given row and column; or at a given index.

HostOS

Prior versions of `HostOS` used to include a way to auto-detect the host operating system. This code was brittle, unreliable, and rather big too. For these reasons, this functionality was removed. Furthermore, `HostOS` now depends on `Kaleidoscope-EEPROM-Settings`, that plugin should be initialized first.

MagicCombo

To make `MagicCombo` more portable, and easier to use, we had to break the API previously provided, there was no way to maintain backwards compatibility. This document is an attempt at guiding you through the process of migrating from the earlier API to the current one.

Migration should be a straightforward process, but if you get stuck, please feel free to open an issue, or start a thread on the forums, and we'll help you with it.

The old API

```
void magicComboActions(uint8_t combo_index, uint32_t left_hand, uint32_t right_hand) {
  switch (combo_index) {
  case 0:
    Macros.type(PSTR("It's a kind of magic!"));
    break;
  }
}

static const kaleidoscope::MagicCombo::combo_t magic_combos[] PROGMEM = {
```

(continues on next page)

(continued from previous page)

```
  {
    R3C6,  // left palm key
    R3C9   // right palm key
  },
  {0, 0}
};

void setup() {
  Kaleidoscope.setup();

  MagicCombo.magic_combos = magic_combos;
}
```

Previously, we used a global, overrideable function (`magicComboActions`) to run the actions of all magic combos, similar to how macros are set up to work. Unlike macros, magic combos can't be defined in the keymap, due to technical reasons, so we had to use a separate list - `magic_combos` in our example. We also needed to tell `MagicCombo` to use this list, which is what we've done in `setup()`.

The new API

```
void kindOfMagic(uint8_t combo_index) {
  Macros.type(PSTR("It's a kind of magic!"));
}

USE_MAGIC_COMBOS({
  .action = kindOfMagic,
  .keys = {R3C6, R3C9} // Left Fn + Right Fn
});
```

The new API is much shorter, and is inspired by the way the *Leader* plugin works: instead of having a list, and a dispatching function like `magicComboActions`, we include the action method in the list too!

We also don't make a difference between left- and right-hand anymore, you can just list keys for either in the same list. This will be very handy for non-split keyboards.

Migration

First of all, we'll need to split up `magicComboActions` into separate functions. Each function should have a unique name, but their shape is always the same:

```
void someFunction(uint8_t combo_index) {
 // Do some action here
}
```

Copy the body of each `case` statement of `magicComboActions`, and copy them one by one into appropriately named functions of the above shape. You can name your functions anything you want, the only constraint is that they need to be valid C++ function names. The plugin itself does nothing with the name, we'll reference them later in the `USE_MAGIC_COMBOS` helper macro.

Once `magicComboActions` is split up, we need to migrate the `magic_combos` list to the new format. That list had to be terminated by a `{0, 0}` entry, the new method does not require such a sentinel at the end.

For each entry in magic_combos, add an entry to USE_MAGIC_COMBOS, with the following structure:

```
{.action = theActionFunction,
 .keys = { /* list of keys */ }}
```

The list of keys are the same RxCy constants you used for magic_combos, with the left- and right hands combined. The action, theActionFunction, is the function you extracted the magic combo action to. It's the function that has the same body as the case statement in magicComboActions had.

And this is all there is to it.

If your actions made use of the left_hand or right_hand arguments of magicComboActions, the same information is still available. But that's a bit more involved to get to, out of scope for this simple migration guide. Please open an issue, or ask for help on the forums, and we'll help you.

OneShot

Older versions of the plugin were based on Key values; OneShot is now based on KeyAddr coordinates instead, in order to improve reliability and functionality.

Qukeys

Older versions of the plugin used row and col indexing for defining Qukey objects. This has since been replaced with a single KeyAddr parameter in the constructor.

Older versions of the plugin used a single timeout, configured via a setTimeout() method. For clarity, that method has been renamed to setHoldTimeout().

Older versions of the plugin used a configurable "release delay" value to give the user control over how Qukeys determined which value to assign to a qukey involved in rollover, via the setReleaseDelay() method. That release delay has been replaced with a better "overlap percentage" strategy, which makes the decision based on the percentage of the subsequent keypress's duration overlaps with the qukey's press. The configuration method is now setOverlapThreshold(), which accepts a value between 0 and 100 (interpreted as a percentage). User who used higher values for setReleaseDelay() will want a lower values for setOverlapThreshold().

These functions have been removed as of **2020-12-31**:

- Qukeys.setTimeout(millis)
- Qukeys.setReleaseDelay(millis)
- Qukey(layer, row, col, alternate_key)

TypingBreaks

Older versions of the plugin used to provide EEPROM storage for the settings only optionally, when it was explicitly enabled via the TypingBreaks.enableEEPROM() method. Similarly, the Focus hooks were optional too.

Storing the settable settings in EEPROM makes it depend on Kaleidoscope-EEPROM-Settings, which should be initialized before this plugin is.

Redial

Older versions of the plugin required one to set up `Key_Redial` manually, and let the plugin know about it via `Redial.key`. This is no longer required, as the plugin sets up the redial key itself. As such, `Redial.key` was removed, and `Key_Redial` is defined by the plugin itself. To upgrade, simply remove your definition of `Key_Redial` and the `Redial.key` assignment from your sketch.

Key masking has been removed

Key masking was a band-aid introduced to avoid accidentally sending unintended keys when key mapping changes between a key being pressed and released. Since the introduction of keymap caching, this is no longer necessary, as long as we can keep the mapping consistent. Users of key masking are encouraged to find ways to use the caching mechanism instead.

As an example, if you had a key event handler that in some cases masked a key, it should now map it to `Key_NoKey` instead, until released.

The masking API has been removed on **2021-01-01**

11.19.3 Deprecated APIs and their replacements

Leader plugin

The `Leader.inject()` function is deprecated. Please call `Runtime.handleKeyEvent()` directly instead.

Direct access to the `Leader.time_out` configuration variable is deprecated. Please use the `Leader.setTimeout(ms)` function instead.

Source code and namespace rearrangement

With the move towards a monorepo-based source, some headers have moved to a new location, and plenty of plugins moved to a new namespace (`kaleidoscope::plugin`). This means that the old headers, and some old names are deprecated. The old names no longer work.

The following headers and names have changed:

- `layers.h`, `key_defs_keymaps.h` and `macro_helpers.h` are obsolete, and should not be included in the first place, as `Kaleidoscope.h` will pull them in. In the rare case that one needs them, prefixing them with `kaleidoscope/` is the way to go. Of the various headers provided under the `kaleidoscope/` space, only `kaleidoscope/macro_helpers.h` should be included directly, and only by hardware plugins that can't pull `Kaleidoscope.h` in due to circular dependencies.
- `LED-Off.h`, provided by *LEDControl* is obsolete, the `LEDOff` LED mode is automatically provided by `Kaleidoscope-LEDControl.h`. The `LED-Off.h` includes can be safely removed.
- `LEDUtils.h` is automatically pulled in by `Kaleiodscope-LEDControl.h`, too, and there's no need to directly include it anymore.
- Plugins that implement LED modes should subclass `kaleidoscope::plugin::LEDMode` instead of `kaleidoscope::LEDMode`.
- *GhostInTheFirmware* had the `kaleidoscope::GhostInTheFirmware::GhostKey` type replaced by `kaleidoscope::plugin::GhostInTheFirmware::GhostKey`.
- *HostOS* no longer provides the `Kaleidoscope/HostOS-select.h` header, and there is no backwards compatibility header either.

- *Leader* had the `kaleidoscope::Leader::dictionary_t` type replaced by `kaleidoscope::plugin::Leader::dictionary_t`.
- *LED-AlphaSquare* used to provide extra symbol graphics in the `kaleidoscope::alpha_square::symbols` namespace. This is now replaced by `kaleidoscope::plugin::alpha_square::symbols`.
- *LEDEffect-SolidColor* replaced the base class - `kaleidoscope::LEDSolidColor` - with `kaleidoscope::plugin::LEDSolidColor`.
- *Qukeys* had the `kaleidoscope::Qukey` type replaced by `kaleidoscope::plugin::Qukey`.
- *ShapeShifter* had the `kaleidoscope::ShapeShifter::dictionary_t` type replaced by `kaleidoscope::plugin::ShapeShifter::dictionary_t`.
- *SpaceCadet* had the `kaleidoscope::SpaceCadet::KeyBinding` type replaced by `kaleidoscope::plugin::SpaceCadet::KeyBinding`.
- *Syster* had the `kaleidoscope::Syster::action_t` type replaced by `kaleidoscope::plugin::Syster::action_t`.
- *TapDance* had the `kaleidoscope::TapDance::ActionType` type replaced by `kaleidoscope::plugin::TapDance::ActionType`.

11.20 Removed APIs

11.20.1 Removed on 2023-11-13

FocusLEDCommand

The brightness functionality of this API lives on in the LEDBrightnessConfig plugin.

11.20.2 Removed on 2022-03-03

Pre-`KeyEvent` event handler hooks

The old event handler `onKeyswitchEvent(Key &key, KeyAddr addr, uint8_t state)` was removed on **2022-03-03**. It has been replaced with the new `onKeyEvent(KeyEvent &event)` handler (and, in some special cases the `onKeyswitchEvent(KeyEvent &event)` handler). Plugins using the deprecated handler will need to be rewritten to use the new one(s).

The old event handler `beforeReportingState()` was removed on **2022-03-03**. It has been replaced with the new `beforeReportingState(KeyEvent &event)` handler. However, the new handler will be called only when a report is being sent (generally in response to a key event), not every cycle, like the old one. It was common practice in the past for plugins to rely on `beforeReportingState()` being called every cycle, so when adapting to the `KeyEvent` API, it's important to check for code that should be moved to `afterEachCycle()` instead.

`::handleKeyswitchEvent(Key key, KeyAddr key_addr, uint8_t state)`

The old master function for processing key "events" was removed on **2022-03-03**. Functions that were calling this function should be rewritten to call `kaleidoscope::Runtime.handleKeyEvent(KeyEvent event)` instead.

`Keyboard::pressKey(Key key, bool toggled_on)`

This deprecated function was removed on **2022-03-03**. Its purpose was to handle rollover events for keys that include modifier flags, and that handling is now done elsewhere. Any code that called it should now simply call `Keyboard::pressKey(Key key)` instead, dropping the second argument.

Old layer key event handler functions

The deprecated `Layer.handleKeymapKeyswitchEvent()` function was removed on **2022-03-03**. Any code that called it should now call `Layer.handleLayerKeyEvent()` instead, with `event.addr` set to the appropriate `KeyAddr` value if possible, and `KeyAddr::none()` otherwise.

The deprecated `Layer.eventHandler(key, addr, state)` function was removed on **2022-03-03**. Any code that refers to it should now call call `handleLayerKeyEvent(KeyEvent(addr, state, key))` instead.

Keymap cache functions

The deprecated `Layer.updateLiveCompositeKeymap()` function was removed on **2022-03-03**. Plugin and user code probably shouldn't have been calling this directly, so there's no direct replacement for it. If a plugin needs to make changes to the `live_keys` structure (equivalent in some circumstances to the old "live composite keymap"), it can call `live_keys.activate(addr, key)`, but there are probably better ways to accomplish this goal (e.g. simply changing the value of `event.key` from an `onKeyEvent(event)` handler).

The deprecated `Layer.lookup(addr)` function was removed on **2022-03-03**. Please use `Runtime.lookupKey(addr)` instead in most circumstances. Alternatively, if you need information about the current state of the keymap regardless of any currently active keys (which may have values that override the keymap), use `Layer.lookupOnActiveLayer(addr)` instead.

`LEDControl.syncDelay` configuration variable

Direct access to this configuration variable was removed on **2022-03-03**. Please use `LEDControl.setInterval()` to set the interval between LED updates instead.

Obsolete active macros array removed

The deprecated `Macros.active_macro_count` variable was removed on **2022-03-03**. Any references to it are obsolete, and can simply be removed.

The deprecated `Macros.active_macros[]` array was removed on **2022-03-03**. Any references to it are obsolete, and can simply be removed.

The deprecated `Macros.addActiveMacroKey()` function was removed on **2022-03-03**. Any references to it are obsolete, and can simply be removed.

Pre-`KeyEvent` Macros API

This is a brief summary of specific elements that were removed. There is a more comprehensive guide to upgrading existing Macros user code in the *Breaking Changes* section, under *Macros*.

Support for deprecated form of the `macroAction(uint8_t macro_id, uint8_t key_state)` function was removed on **2022-03-03**. This old form must be replaced with the new `macroAction(uint8_t macro_id, KeyEvent &event)` for macros to continue working.

The `Macros.key_addr` public variable was removed on **2022-03-03**. To get access to the key address of a Macros key event, simply refer to `event.addr` from within the new `macroAction(macro_id, event)` function.

The deprecated `MACRODOWN()` preprocessor macro was removed on **2022-03-03**. Since most macros are meant to be triggered only by keypress events (not key release), and because `macroAction()` does not get called every cycle for held keys, it's better to simply do one test for `keyToggledOn(event.state)` first, then use `MACRO()` instead.

ActiveModColor public variables

The following deprecated `ActiveModColorEffect` public variables were removed on **2022-03-03**. Please use the following methods instead:

- For `ActiveModColor.highlight_color`, use `ActiveModColor.setHighlightColor(color)`
- For `ActiveModColor.oneshot_color`, use `ActiveModColor.setOneShotColor(color)`
- For `ActiveModColor.sticky_color`, use `ActiveModColor.setStickyColor(color)`

OneShot public variables

The following deprecated `OneShot` public variables were removed on **2022-03-03**. Please use the following methods instead:

- For `OneShot.time_out`, use `OneShot.setTimeout(ms)`
- For `OneShot.hold_time_out`, use `OneShot.setHoldTimeout(ms)`
- For `OneShot.double_tap_time_out`, use `OneShot.setDoubleTapTimeout(ms)`

Deprecated OneShot API functions

OneShot was completely rewritten in early 2021, and now is based on `KeyAddr` values (as if it keeps physical keys pressed) rather than `Key` values (with no corresponding physical key location). This allows it to operate on any `Key` value, not just modifiers and layer shifts.

The deprecated `OneShot.inject(key, key_state)` function was removed on **2022-03-03**. Its use was very strongly discouraged, and is now unavailable. See below for alternatives.

The deprecated `OneShot.isActive(key)` function was removed on **2022-03-03**. There is a somewhat equivalent `OneShot.isActive(KeyAddr addr)` function to use when the address of a key that might be currently held active by OneShot is known. Any code that needs information about active keys is better served by not querying OneShot specifically.

The deprecated `OneShot.isSticky(key)` function was removed on **2022-03-03**. There is a somewhat equivalent `OneShot.isStick(KeyAddr addr)` function to use when the address of a key that may be in the one-shot sticky state is known.

The deprecated `OneShot.isPressed()` function was removed on **2022-03-03**. It was already devoid of functionality, and references to it can be safely removed.

The deprecated `OneShot.isModifierActive(key)` function was removed on **2022-03-03**. OneShot modifiers are now indistinguishable from other modifier keys, so it is better for client code to do a more general search of `live_keys` or to use another mechanism for tracking this state.

`HostPowerManagement.enableWakeup()`

This deprecated function was removed on **2022-03-03**. The firmware now supports wakeup by default, so any references to it can be safely removed.

`EEPROMSettings.version(uint8_t version)`

This deprecated function was removed on **2022-03-03**. The information stored is not longer intended for user code to set, but instead is used internally.

Model01-TestMode plugin

This deprecated plugin was removed on **2022-03-03**. Please use the more generic HardwareTestMode plugin instead.

11.20.3 Removed on 2020-10-10

Deprecation of the HID facade

With the new Device APIs it became possible to replace the HID facade (the `kaleidoscope::hid` family of functions) with a driver. As such, the old APIs are deprecated, and was removed on 2020-10-10. Please use `Kaleidoscope.hid()` instead.

Implementation of type Key internally changed from C++ union to class

The deprecated functions were removed on 2020-10-10.

11.20.4 Removed on 2020-06-16

The old device API

After the introduction of the new device API, the old APIs (`ROWS`, `COLS`, `LED_COUNT`, `KeyboardHardware`, the old `Hardware` base class, etc) were removed on **2020-06-16**.

`LEDControl.mode_add()`

Since March of 2019, this method has been deprecated, and turned into a no-op. While no removal date was posted at the time, after more than a year of deprecation, it has been removed on **2020-06-16**.

`LEDControl.paused`

Wherever we used `LEDControl.paused`, we'll need to use one of `LEDControl.disable()`, `LEDControl.enable()`, or `LEDControl.isEnabled()` instead. `LEDControl.paused` has been removed on **2020-06-16**.

Keep in mind that `.enable()` and `.disable()` do more than what `paused` did: they will refresh and turn off LEDs too, respectively.

A few examples to show how to transition to the new APIs follow, old use first, new second.

```
if (someCondition) {
  LEDControl.set_all_leds_to({0, 0, 0});
  LEDControl.syncLeds();
  LEDControl.paused = true;
} else if (someOtherCondition) {
  LEDControl.paused = false;
  LEDControl.refreshAll();
}

if (LEDControl.paused) {
 // do things...
}
```

```
if (someCondition) {
  LEDControl.disable();
} else if (someOtherCondition) {
  LEDControl.enable();
}
if (!LEDControl.isEnabled()) {
  // do things...
}
```

Class/global instance Kaleidoscope_/Kaleidoscope renamed to kaleidoscope::Runtime_/kaleidoscope::Runtime

After the renaming, Kaleidoscope core should be using `kaleidoscope::Runtime`. The former `Kaleidoscope` global symbol is to be used by sketches only - and only to not diverge too much from the Arduino naming style.

The deprecated `Kaleidoscope_` class has been removed on **2020-06-16**.

Transition to linear indexing

Row/col based indexing was replaced by linear indexing throughout the whole firmware. A compatibility layer of functions was introduced that allows the firmware to remain backwards compatible, however, these functions are deprecated and will be removed in future versions of the firmware.

Also a new version of the onKeyswitchEvent-handler has been introduced.

The deprecated row/col based indexing APIs have been removed on **2020-06-16**.

11.20.5 Removed on 2020-01-06

EEPROMKeymap mode

The *EEPROM-Keymap* plugin had its `setup()` method changed, the formerly optional `method` argument is now obsolete and unused. It can be safely removed.

keymaps array and KEYMAPS and KEYMAPS_STACKED macros

The `keymaps` array has been replaced with a `keymaps_linear` array. This new array treats each layer as a simple one dimensional array of keys, rather than a two dimensional array of arrays of rows. At the same time, the `KEYMAPS` and `KEYMAPS_STACKED` macros that were previously defined in each hardware implmentation class have been replaced with `PER_KEY_DATA` and `PER_KEY_DATA_STACKED` macros in each hardware class. This change should be invisible to users, but will require changes by any plugin that accessed the 'keymaps' variable directly.

Code like `key.raw = pgm_read_word(&(keymaps[layer][row][col])); return key;` should be changed to look like this: `return keyFromKeymap(layer, row, col);`

11.20.6 Removed on 2019-01-18

Removal of Layer.defaultLayer

The `Layer.defaultLayer()` method has been deprecated, because it wasn't widely used, nor tested well, and needlessly complicated the layering logic. If one wants to set a default layer, which the keyboard switches to when booting up, `EEPROMSettings.default_layer()` may be of use.

`Layer.defaultLayer` has since been removed.

More clarity in Layer method names

A number of methods on the `Layer` object have been renamed, to make their intent clearer:

- `Layer.on()` and `Layer.off()` became `Layer.activate()` and `Layer.decativate()`, repsectively.
- `Layer.next()` and `Layer.previous()` became `Layer.activateNext()` and `Layer.deactivateTop()`.
- `Layer.isOn` became `Layer.isActive()`.

The goal was to have a method name that is a verb, because these are actions we do. The old names have since been removed.

11.20.7 Removed on 2019-01-17

Compat headers following the source code and namespace rearrangement

With the move towards a monorepo-based source, some headers have moved to a new location, and plenty of plugins moved to a new namespace (`kaleidoscope::plugin`). This means that the old headers, and some old names are deprecated. The old names no longer work.

HostOS.autoDetect()

The `autoDetect()` method has been formerly deprecated, and is now removed.

The old MagicCombo API

We've changed the API of the MagicCombo plugin, and while it provided a helpful error message for a while when trying to use the old API, it no longer does so, the error message has been removed.

TypingBreaks.enableEEPROM()

`TypingBreaks.enableEEPROM()` has been previously deprecated, and turned into a no-op, and is now removed.

`OneShot.double_tap_sticky` and `OneShot.double_tap_layer_sticky`

These were deprecated in favour of a better, finer grained API, and are now removed.

11.20.8 Removed on 2018-08-20

We aim at making a new release by mid-July, and APIs we deprecate now, will be removed shortly after the major release, before the next point release. We may deprecate further APIs during the next month (until mid-June), and those deprecations will share the same removal date. We will try our best to minimize deprecations, and do them as soon as possible, to give everyone at least a month to prepare and update.

Kaleidoscope.use()

Deprecated in May 2018, this method is part of the old plugin API, replaced by `KALEIDOSCOPE_INIT_PLUGINS`. To upgrade, you need to modify your .ino sketch file, and replace the text `Kaleidoscope.use` with `KALEIDOSCOPE_INIT_PLUGINS`, then remove the `&` from all of the plugins inside it, and finally, move it outside of `setup()`.

If your current sketch looks like this:

```
void setup() {
  Kaleidoscope.use(&Plugin1, &Plugin2);
  Kaleidoscope.setup();
}
```

You should change it so that it looks like this instead:

```
KALEIDOSCOPE_INIT_PLUGINS(Plugin1, Plugin2);

void setup() {
  Kaleidoscope.setup();
}
```

The old-style (v1) plugin API

This includes using `KaleidoscopePlugin`, `Kaleidoscope.useEventHandlerHook`, `Kaleidoscope.replaceEventHandlerHook`, `Kaleidoscope.appendEventHandlerHook`, `Kaleidoscope.useLoopHook`, `Kaleidoscope.replaceLoopHook`, `Kaleidoscope.appendLoopHook`. They were deprecated in May 2017.

Their replacement is the new plugin API:

```
namespace kaleidoscope {

enum class EventHandlerResult {
  OK,
  EVENT_CONSUMED,
  ERROR,
};

class Plugin {
public:
  EventHandlerResult onSetup();
  EventHandlerResult beforeEachCycle();
  EventHandlerResult onKeyswitchEvent(Key &mapped_key, KeyAddr key_addr, uint8_t key_
↪state);
  EventHandlerResult beforeReportingState();
  EventHandlerResult afterEachCycle();
};

}
```

Plugins are supposed to implement this new API, and then be initialised via `KALEIDOSCOPE_INIT_PLUGINS`.

Consumer_SNapshot

A key with a typo in its name, which was left in place after fixing the typo, so as to not break any code that may be using it already, however unlikely.

11.20.9 Removed on 2018-06-10 (originally scheduled for 2018-05-27)

These APIs and functions have been deprecated for a long time, and as far as we can tell, aren't used by any third party or user code anymore. They were removed as of the June 10th, 2018.

Kaleidoscope.setup(KEYMAP_SIZE)

The `Kaleidoscope.setup()` method is still around, and is **not** deprecated, but the variant of it that takes a keymap size is, and has been since October 2017.

Instead, one should use the argument-less `Kaleidoscope.setup()`, and the new `KEYMAP()` macros to define a keymap.

event_handler_hook_use, loop_hook_use, and USE_PLUGINS

Deprecated in October 2017, these are old aliases that should no longer be in use. They were replaced by `Kaleidoscope.useEventHandlerHook`, `Kaleidoscope.useLoopHook`, and `Kaleidoscope.use`, respectively.

The replacements themselves are also deprecated - see below -, but their removal will come at a later date.

MOMENTARY_OFFSET

Deprecated in October 2017, replaced by `LAYER_SHIFT_OFFSET`.

This symbol was meant to be used by plugins, not user code, and as far as we know, no third party plugin ever used it.

key_was_pressed, key_is_pressed, key_toggled_on, key_toggled_off

Deprecated in July 2017, replaced by `keyWasPressed`, `keyIsPressed`, `keyToggledOn`, and `keyToggledOff`, respectively.

11.21 Contributor Covenant Code of Conduct

11.21.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

11.21.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

11.21.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

11.21.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

11.21.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at jesse@keyboard.io. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

11.21.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

11.22 All example sketches

11.22.1 Basic/Basic.ino

```
,/* -*- mode: c++ -*-
 * Basic -- A very basic Kaleidoscope example
 * Copyright (C) 2018  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
```

(continues on next page)

(continued from previous page)

```
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include "Kaleidoscope.h"

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
   Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_NoKey,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_NoKey
  ),
)
// clang-format on

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.2 Devices/EZ/ErgoDox/ErgoDox.ino

```
,/* -*- mode: c++ -*-
 * ErgoDox -- Chrysalis-enabled Sketch for ErgoDox-compatible boards
 * Copyright (C) 2019-2022  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

(continues on next page)

(continued from previous page)

```
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

/*
 * This is based on the QMK factory firmware the ErgoDox EZ ships with. Modeled
 * after the layout in
 * https://configure.ergodox-ez.com/layouts/default/latest/0, as of 2019-01-04.
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-DynamicMacros.h"
#include "Kaleidoscope-Escape-OneShot.h"
#include "Kaleidoscope-EEPROM-Settings.h"
#include "Kaleidoscope-EEPROM-Keymap.h"
#include "Kaleidoscope-FirmwareVersion.h"
#include "Kaleidoscope-FocusSerial.h"
#include "Kaleidoscope-MouseKeys.h"
#include "Kaleidoscope-OneShot.h"
#include "Kaleidoscope-Qukeys.h"
#include "Kaleidoscope-SpaceCadet.h"

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
      // left hand
      Key_Equals,      Key_1,     Key_2,                Key_3,         Key_4,         ␣
↪Key_5,   Key_LeftArrow,
      Key_Delete,      Key_Q,     Key_W,                Key_E,         Key_R,         ␣
↪Key_T,   LockLayer(1),
      Key_Backspace,   Key_A,     Key_S,                Key_D,         Key_F,         ␣
↪Key_G,
      Key_LeftShift,   Key_Z,     Key_X,                Key_C,         Key_V,         ␣
↪Key_B,   Key_Hyper,
      LT(1, Backtick), Key_Quote, LALT(Key_LeftShift), Key_LeftArrow, Key_RightArrow,

                     MT(LeftAlt, PcApplication), Key_LeftGui,
                                                 Key_Home,
      Key_Space,     Key_Backspace,              Key_End,

      // right hand
      Key_RightArrow, Key_6, Key_7,       Key_8,        Key_9,           Key_0,         ␣
↪   Key_Minus,
      LockLayer(1),   Key_Y, Key_U,       Key_I,        Key_O,           Key_P,         ␣
↪   Key_Backslash,
                      Key_H, Key_J,       Key_K,        Key_L,           LT(2,␣
↪Semicolon), MT(LeftGui, Quote),
      Key_Meh,        Key_N, Key_M,       Key_Comma,    Key_Period,      Key_Slash,     ␣
```

(continues on next page)

(continued from previous page)

```
↪   Key_RightShift,
                              Key_UpArrow, Key_DownArrow, Key_LeftBracket, Key_
↪RightBracket, ShiftToLayer(1),

    Key_LeftAlt,  MT(LeftControl, Esc),
    Key_PageUp,
    Key_PageDown, Key_Tab, Key_Enter
),
[1] = KEYMAP_STACKED
(
    // left hand
    Key_Esc, Key_F1,       Key_F2,       Key_F3,                Key_F4,          ␣
↪      Key_F5,              XXX,
    XXX,     LSHIFT(Key_1), LSHIFT(Key_2), LSHIFT(Key_LeftBracket), LSHIFT(Key_
↪RightBracket), LSHIFT(Key_Backslash), ___,
    XXX,     LSHIFT(Key_3), LSHIFT(Key_4), LSHIFT(Key_9),         LSHIFT(Key_0),    ␣
↪      Key_Backtick,
    XXX,     LSHIFT(Key_5), LSHIFT(Key_6), Key_LeftBracket,        Key_RightBracket, ␣
↪      LSHIFT(Key_Backtick),  XXX,
    ___,     XXX,          XXX,          XXX,                   XXX,

         XXX, XXX,
              XXX,
    XXX, XXX, XXX,

    // right hand
    XXX, Key_F6,        Key_F7, Key_F8,     Key_F9, Key_F10,       Key_F11,
    ___, Key_UpArrow,   Key_7,  Key_8,      Key_9,  LSHIFT(Key_8), Key_F12,
         Key_DownArrow, Key_4,  Key_5,      Key_6,  XXX,           XXX,
    XXX, LSHIFT(Key_7), Key_1,  Key_2,      Key_3,  Key_Backslash, XXX,
                        XXX,    Key_Period, Key_0,  Key_Equals,    ___,

    XXX, XXX,
    XXX,
    XXX, XXX, XXX
),
[2] = KEYMAP_STACKED
(
    // left hand
    XXX, XXX, XXX,       XXX,          XXX,          XXX, XXX,
    XXX, XXX, XXX,       Key_mouseUp,  XXX,          XXX, XXX,
    XXX, XXX, Key_mouseL, Key_mouseDn,  Key_mouseR,   XXX,
    XXX, XXX, XXX,       XXX,          XXX,          XXX,  XXX,
    XXX, XXX, XXX,       Key_mouseBtnL, Key_mouseBtnR,

         XXX, XXX,
              XXX,
    XXX, XXX, XXX,

    // right hand
    XXX, XXX, XXX,                    XXX,                   XXX,              ␣
↪    XXX, XXX,
```

(continues on next page)

(continued from previous page)

```
     XXX, XXX, XXX,                    XXX,                      XXX,              ␣
↪     XXX, XXX,
         XXX, XXX,                     XXX,                      XXX,              ␣
↪     ___, Consumer_PlaySlashPause,
     XXX, XXX, XXX,                    Consumer_ScanPreviousTrack, Consumer_
↪ScanNextTrack, XXX, XXX,
              Consumer_VolumeIncrement, Consumer_VolumeDecrement,   Consumer_Mute,    ␣
↪     XXX, XXX,

     XXX, XXX,
     XXX,
     XXX, XXX, XXX
  ),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(
  EEPROMSettings,
  EEPROMKeymap,
  Focus,
  FocusEEPROMCommand,
  FocusSettingsCommand,
  Qukeys,
  SpaceCadet,
  OneShot,
  EscapeOneShot,
  EscapeOneShotConfig,
  DynamicMacros,
  MouseKeys,
  FirmwareVersion);

void blinkAllStatusLEDs() {
  for (auto i = 0; i <= 3; i++) {
    Kaleidoscope.device().setStatusLED(i, false);
  }

  for (auto i = 1; i <= 3; i++) {
    Kaleidoscope.device().setStatusLED(i, true);
    _delay_ms(50);
  }

  for (auto i = 1; i <= 3; i++) {
    Kaleidoscope.device().setStatusLED(i, false);
    _delay_ms(50);
  }
}

void setup() {
  Kaleidoscope.setup();

  EEPROMKeymap.setup(5);
  SpaceCadet.disable();
```

(continues on next page)

(continued from previous page)

```
  DynamicMacros.reserve_storage(256);

  blinkAllStatusLEDs();

  Layer.move(EEPROMSettings.default_layer());
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.3 Devices/KBDFans/KBD4x/KBD4x.ino

```
,/* -*- mode: c++ -*-
 * KBD4x -- A very basic Kaleidoscope example for the KBDFans KBD4x keyboard
 * Copyright (C) 2019  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTabILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-Macros.h"

enum {
  _QWERTY,
  _FN
};

enum {
  RESET
};

#define MO(n) ShiftToLayer(n)

// clang-format off
KEYMAPS(

/* Qwerty
```

(continues on next page)

(continued from previous page)

```
 * ,-------------------------------------------------------------------------.
 * | Esc  |  Q  |  W  |  E  |  R  |  T  |  Y  |  U  |  I  |  O  |  P  | Bksp |
 * |------+-----+-----+-----+-----+-----------+-----+-----+-----+-----+------|
 * | Tab  |  A  |  S  |  D  |  F  |  G  |  H  |  J  |  K  |  L  |  ;  |  "   |
 * |------+-----+-----+-----+-----+-----|-----+-----+-----+-----+-----+------|
 * | Shift|  Z  |  X  |  C  |  V  |  B  |  N  |  M  |  ,  |  .  | Up  |Enter |
 * |------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+------|
 * | Ctrl | GUI |  1  |  2  |  3  |   Space   |  FN |  /  | Lft | Dn  |Right |
 * `-------------------------------------------------------------------------'
 */

[_QWERTY] = KEYMAP(
   Key_Escape      ,Key_Q        ,Key_W ,Key_E ,Key_R ,Key_T ,Key_Y ,Key_U   ,Key_I     ,
↪Key_O         ,Key_P         ,Key_Backspace
  ,Key_Tab         ,Key_A        ,Key_S ,Key_D ,Key_F ,Key_G ,Key_H ,Key_J   ,Key_K     ,
↪Key_L         ,Key_Semicolon ,Key_Quote
  ,Key_LeftShift   ,Key_Z        ,Key_X ,Key_C ,Key_V ,Key_B ,Key_N ,Key_M   ,Key_Comma ,
↪Key_Period    ,Key_UpArrow   ,Key_Enter
  ,Key_LeftControl ,Key_LeftGui ,Key_1 ,Key_2 ,Key_3   ,Key_Space  ,MO(_FN) ,Key_Slash ,
↪Key_LeftArrow ,Key_DownArrow ,Key_RightArrow
),

/* Fn
 * ,-----------------------------------------------------------------------------------.
 * |   `  |   1  |   2  |   3  |   4  |   5  |   6  |   7  |   8  |   9  |   0  |      |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * |      |      |      |      |      |      |      |      |      |      |      | RST  |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * |      |      |      |      |      |      |      |      |      |      |      |      |
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * |      |      |      |      |      |             |      |      |      |      |      |
 * `-----------------------------------------------------------------------------------'
 */
[_FN] = KEYMAP(
   Key_Backtick ,Key_1 ,Key_2 ,Key_3 ,Key_4 ,Key_5 ,Key_6 ,Key_7 ,Key_8 ,Key_9 ,Key_0 ,__
↪_
  ,___          ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,
↪M(RESET)
  ,___          ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,___   ,__
↪_
  ,___          ,___   ,___   ,___   ,___       ,___       ,___   ,___   ,___   ,___   ,__
↪_
)
);
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(Macros);

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {
  case RESET:
    if (keyToggledOn(event.state))
```

(continues on next page)

(continued from previous page)

```
      Kaleidoscope.rebootBootloader();
    break;
  default:
    break;
  }

  return MACRO_NONE;
}

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.4 Devices/Keyboardio/Atreus/Atreus.ino

```
,/* -*- mode: c++ -*-
 * Atreus -- Chrysalis-enabled Sketch for the Keyboardio Atreus
 * Copyright (C) 2018-2022  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

#ifndef BUILD_INFORMATION
#define BUILD_INFORMATION "locally built on " __DATE__ " at " __TIME__
#endif

#include "Kaleidoscope.h"
#include "Kaleidoscope-EEPROM-Settings.h"
#include "Kaleidoscope-EEPROM-Keymap.h"
#include "Kaleidoscope-Escape-OneShot.h"
#include "Kaleidoscope-FirmwareVersion.h"
#include "Kaleidoscope-FocusSerial.h"
#include "Kaleidoscope-Macros.h"
#include "Kaleidoscope-MouseKeys.h"
```

(continues on next page)

(continued from previous page)

```
#include "Kaleidoscope-OneShot.h"
#include "Kaleidoscope-Qukeys.h"
#include "Kaleidoscope-SpaceCadet.h"
#include "Kaleidoscope-DynamicMacros.h"
#include "Kaleidoscope-LayerNames.h"

#define MO(n) ShiftToLayer(n)
#define TG(n) LockLayer(n)

enum {
  MACRO_QWERTY,
  MACRO_VERSION_INFO
};

#define Key_Exclamation LSHIFT(Key_1)
#define Key_At          LSHIFT(Key_2)
#define Key_Hash        LSHIFT(Key_3)
#define Key_Dollar      LSHIFT(Key_4)
#define Key_Percent     LSHIFT(Key_5)
#define Key_Caret       LSHIFT(Key_6)
#define Key_And         LSHIFT(Key_7)
#define Key_Star        LSHIFT(Key_8)
#define Key_Plus        LSHIFT(Key_Equals)

enum {
  QWERTY,
  FUN,
  UPPER
};

// clang-format off
KEYMAPS(
  [QWERTY] = KEYMAP_STACKED
  (
       Key_Q   ,Key_W   ,Key_E       ,Key_R         ,Key_T
      ,Key_A   ,Key_S   ,Key_D       ,Key_F         ,Key_G
      ,Key_Z   ,Key_X   ,Key_C       ,Key_V         ,Key_B, Key_Backtick
      ,Key_Esc ,Key_Tab ,Key_LeftGui ,Key_LeftShift ,Key_Backspace ,Key_LeftControl

                     ,Key_Y     ,Key_U      ,Key_I     ,Key_O      ,Key_P
                     ,Key_H     ,Key_J      ,Key_K     ,Key_L      ,Key_Semicolon
       ,Key_Backslash,Key_N     ,Key_M      ,Key_Comma ,Key_Period ,Key_Slash
       ,Key_LeftAlt  ,Key_Space ,MO(FUN)    ,Key_Minus ,Key_Quote  ,Key_Enter
  ),

  [FUN] = KEYMAP_STACKED
  (
       Key_Exclamation ,Key_At           ,Key_UpArrow   ,Key_Dollar           ,Key_
↪Percent
      ,Key_LeftParen   ,Key_LeftArrow    ,Key_DownArrow ,Key_RightArrow       ,Key_
↪RightParen
      ,Key_LeftBracket ,Key_RightBracket ,Key_Hash      ,Key_LeftCurlyBracket ,Key_
```

(continues on next page)

(continued from previous page)

```
↪RightCurlyBracket ,Key_Caret
      ,TG(UPPER)       ,Key_Insert       ,Key_LeftGui   ,Key_LeftShift        ,Key_
↪Delete         ,Key_LeftControl

                    ,Key_PageUp   ,Key_7 ,Key_8      ,Key_9 ,Key_Backspace
                    ,Key_PageDown ,Key_4 ,Key_5      ,Key_6 ,___
      ,Key_And      ,Key_Star     ,Key_1 ,Key_2      ,Key_3 ,Key_Plus
      ,Key_LeftAlt ,Key_Space    ,___   ,Key_Period ,Key_0 ,Key_Equals
   ),

  [UPPER] = KEYMAP_STACKED
  (
       Key_Insert            ,Key_Home                 ,Key_UpArrow   ,Key_End        ,
↪Key_PageUp
      ,Key_Delete            ,Key_LeftArrow            ,Key_DownArrow ,Key_RightArrow ,
↪Key_PageDown
      ,M(MACRO_VERSION_INFO) ,Consumer_VolumeIncrement ,XXX           ,XXX            ,__
↪_ ,___
      ,MoveToLayer(QWERTY)   ,Consumer_VolumeDecrement ,___           ,___            ,__
↪_ ,___

                 ,Key_UpArrow   ,Key_F7              ,Key_F8          ,Key_F9         ,
↪Key_F10
                 ,Key_DownArrow ,Key_F4              ,Key_F5          ,Key_F6         ,
↪Key_F11
      ,___       ,XXX           ,Key_F1              ,Key_F2          ,Key_F3         ,
↪Key_F12
      ,___       ,___           ,MoveToLayer(QWERTY) ,Key_PrintScreen ,Key_ScrollLock ,
↪Consumer_PlaySlashPause
   )
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(
  // ----------------------------------------------------------------------
  // Chrysalis plugins

  // The EEPROMSettings & EEPROMKeymap plugins make it possible to have an
  // editable keymap in EEPROM.
  EEPROMSettings,
  EEPROMKeymap,

  // Focus allows bi-directional communication with the host, and is the
  // interface through which the keymap in EEPROM can be edited.
  Focus,

  // FocusSettingsCommand adds a few Focus commands, intended to aid in
  // changing some settings of the keyboard, such as the default layer (via the
  // `settings.defaultLayer` command)
  FocusSettingsCommand,

  // FocusEEPROMCommand adds a set of Focus commands, which are very helpful in
```

(continues on next page)

(continued from previous page)

```
  // both debugging, and in backing up one's EEPROM contents.
  FocusEEPROMCommand,

  // The FirmwareVersion plugin lets Chrysalis query the version of the firmware
  // programmatically.
  FirmwareVersion,

  // The LayerNames plugin allows Chrysalis to display - and edit - custom layer
  // names, to be shown instead of the default indexes.
  LayerNames,

  // ---------------------------------------------------------------------
  // Keystroke-handling plugins

  // The Qukeys plugin enables the "Secondary action" functionality in
  // Chrysalis. Keys with secondary actions will have their primary action
  // performed when tapped, but the secondary action when held.
  Qukeys,

  // SpaceCadet can turn your shifts into parens on tap, while keeping them as
  // Shifts when held. SpaceCadetConfig lets Chrysalis configure some aspects of
  // the plugin.
  SpaceCadet,
  SpaceCadetConfig,

  // Enables the "Sticky" behavior for modifiers, and the "Layer shift when
  // held" functionality for layer keys.
  OneShot,
  OneShotConfig,
  EscapeOneShot,
  EscapeOneShotConfig,

  // The macros plugin adds support for macros
  Macros,

  // Enables dynamic, Chrysalis-editable macros.
  DynamicMacros,

  // The MouseKeys plugin lets you add keys to your keymap which move the mouse.
  MouseKeys,
  MouseKeysConfig  //,

  // The MagicCombo plugin lets you use key combinations to trigger custom
  // actions - a bit like Macros, but triggered by pressing multiple keys at the
  // same time.
  // MagicCombo,

  // Enables the GeminiPR Stenography protocol. Unused by default, but with the
  // plugin enabled, it becomes configurable - and then usable - via Chrysalis.
  // GeminiPR,
);
```

(continues on next page)

(continued from previous page)

```
const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    switch (macro_id) {
    case MACRO_QWERTY:
      // This macro is currently unused, but is kept around for compatibility
      // reasons. We used to use it in place of `MoveToLayer(QWERTY)`, but no
      // longer do. We keep it so that if someone still has the old layout with
      // the macro in EEPROM, it will keep working after a firmware update.
      Layer.move(QWERTY);
      break;
    case MACRO_VERSION_INFO:
      Macros.type(PSTR("Keyboardio Atreus - Kaleidoscope "));
      Macros.type(PSTR(BUILD_INFORMATION));
      break;
    default:
      break;
    }
  }
  return MACRO_NONE;
}

void setup() {
  Kaleidoscope.setup();
  EEPROMKeymap.setup(9);

  DynamicMacros.reserve_storage(48);

  LayerNames.reserve_storage(63);

  Layer.move(EEPROMSettings.default_layer());

  // To avoid any surprises, SpaceCadet is turned off by default. However, it
  // can be permanently enabled via Chrysalis, so we should only disable it if
  // no configuration exists.
  SpaceCadetConfig.disableSpaceCadetIfUnconfigured();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.5 Devices/Keyboardio/Imago/Imago.ino

```
,/* -*- mode: c++ -*-
 * Imago.ino -- Example sketch for the Keyboardio Imago
 * Copyright (C) 2018, 2019, 2020  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License,        or
```

(continues on next page)

(continued from previous page)

```
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTabILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not,        write to the Free Software Foundation,        Inc.,
 * 51 Franklin Street,        Fifth Floor,        Boston,        MA 02110-1301 USA.
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-Macros.h"

// Support for controlling the keyboard's LEDs
#include "Kaleidoscope-LEDControl.h"

// Support for the "Boot greeting" effect, which pulses the 'LED' button for 10s
// when the keyboard is connected to a computer (or that computer is powered on)
#include "Kaleidoscope-LEDEffect-BootGreeting.h"

// Support for LED modes that set all LEDs to a single color
#include "Kaleidoscope-LEDEffect-SolidColor.h"

// Support for an LED mode that makes all the LEDs 'breathe'
#include "Kaleidoscope-LEDEffect-Breathe.h"


#include "Kaleidoscope-LEDEffect-Chase.h"

// Support for LED modes that pulse the keyboard's LED in a rainbow pattern
#include "Kaleidoscope-LEDEffect-Rainbow.h"


// Support for host power management (suspend & wakeup)
#include "Kaleidoscope-HostPowerManagement.h"

// Support for magic combos (key chords that trigger an action)
#include "Kaleidoscope-MagicCombo.h"

// Support for USB quirks, like changing the key state report protocol
#include "Kaleidoscope-USB-Quirks.h"


enum {
  _QWERTY,
};


// clang-format off
KEYMAPS(
```

(continues on next page)

(continued from previous page)

```
[_QWERTY] = KEYMAP(

Key_F1, Key_Escape,        Key_Backtick,        Key_1,               Key_2,        ␣
↪        Key_3,               Key_4,          Key_5, Key_6,       Key_7,
↪                Key_8,                 Key_9,            Key_0,          Key_Minus,
↪        Key_Equals,        Key_Backspace,
Key_F2, Key_Tab,        Key_Q,                Key_W,                Key_E,        ␣
↪        Key_R,               Key_T,                                    Key_Y,␣
↪                Key_U,                 Key_I,            Key_O,      Key_P,        ␣
↪        Key_LeftBracket, Key_RightBracket, Key_Backslash,
Key_F3, Key_Escape,        Key_A,                 Key_S,                 Key_D,        ␣
↪        Key_F,               Key_G,                                    Key_H,␣
↪                Key_J,                 Key_K,              Key_L,            Key_
↪Semicolon,        Key_Quote, Key_Enter,
Key_F4, Key_LeftShift,        Key_Z,                 Key_X,                Key_C,
↪                 Key_V,                Key_B,            Key_UpArrow,           Key_N,␣
↪                Key_M,                 Key_Comma,    Key_Period,        Key_Slash,
↪        Key_RightShift,        Key_LEDEffectNext,
Key_F5,        Key_LeftControl,Key_LeftAlt,        Key_LeftGui,          Key_Backspace,  ␣
↪  Key_LeftArrow,                     Key_DownArrow,        Key_RightArrow, Key_Space,␣
↪        Key_RightAlt, Key_Menu, Key_RightControl, Key_LEDEffectNext


));
// clang-format on


// These 'solid' color effect definitions define a rainbow of
// LED color modes calibrated to draw 500mA or less on the
// Keyboardio Model 01.


static kaleidoscope::plugin::LEDSolidColor solidRed(160, 0, 0);
static kaleidoscope::plugin::LEDSolidColor solidOrange(140, 70, 0);
static kaleidoscope::plugin::LEDSolidColor solidYellow(130, 100, 0);
static kaleidoscope::plugin::LEDSolidColor solidGreen(0, 160, 0);
static kaleidoscope::plugin::LEDSolidColor solidBlue(0, 70, 130);
static kaleidoscope::plugin::LEDSolidColor solidIndigo(0, 0, 170);
static kaleidoscope::plugin::LEDSolidColor solidViolet(130, 0, 120);


/** toggleLedsOnSuspendResume toggles the LEDs off when the host goes to sleep,
 * and turns them back on when it wakes up.
 */
void toggleLedsOnSuspendResume(kaleidoscope::plugin::HostPowerManagement::Event event) {
  switch (event) {
  case kaleidoscope::plugin::HostPowerManagement::Suspend:
    LEDControl.disable();
    break;
  case kaleidoscope::plugin::HostPowerManagement::Resume:
    LEDControl.enable();
```

(continues on next page)

(continued from previous page)

```
    break;
  case kaleidoscope::plugin::HostPowerManagement::Sleep:
    break;
  }
}

/** hostPowerManagementEventHandler dispatches power management events (suspend,
 * resume, and sleep) to other functions that perform action based on these
 * events.
 */
void hostPowerManagementEventHandler(kaleidoscope::plugin::HostPowerManagement::Event␣
↪event) {
  toggleLedsOnSuspendResume(event);
}

/** This 'enum' is a list of all the magic combos used by the Model 01's
 * firmware The names aren't particularly important. What is important is that
 * each is unique.
 *
 * These are the names of your magic combos. They will be used by the
 * `USE_MAGIC_COMBOS` call below.
 */
enum {
  // Toggle between Boot (6-key rollover; for BIOSes and early boot) and NKRO
  // mode.
  COMBO_TOGGLE_NKRO_MODE
};

/** A tiny wrapper, to be used by MagicCombo.
 * This simply toggles the keyboard protocol via USBQuirks, and wraps it within
 * a function with an unused argument, to match what MagicCombo expects.
 */
static void toggleKeyboardProtocol(uint8_t combo_index) {
  USBQuirks.toggleKeyboardProtocol();
}

/** Magic combo list, a list of key combo and action pairs the firmware should
 * recognise.
 */
USE_MAGIC_COMBOS(
  {.action = toggleKeyboardProtocol,
   // Left Fn + Esc + Shift
   .keys = {R3C6, R2C6, R3C7}});


KALEIDOSCOPE_INIT_PLUGINS(
  Macros,

  // LEDControl provides support for other LED modes
  LEDControl,
```

(continues on next page)

(continued from previous page)

```
  // The rainbow effect changes the color of all of the keyboard's keys at the same time
  // running through all the colors of the rainbow.
  LEDRainbowEffect,

  // The rainbow wave effect lights up your keyboard with all the colors of a rainbow
  // and slowly moves the rainbow across your keyboard
  LEDRainbowWaveEffect,

  // The chase effect follows the adventure of a blue pixel which chases a red pixel␣
↪across
  // your keyboard. Spoiler: the blue pixel never catches the red pixel
  LEDChaseEffect,

  // These static effects turn your keyboard's LEDs a variety of colors
  solidRed,
  solidOrange,
  solidYellow,
  solidGreen,
  solidBlue,
  solidIndigo,
  solidViolet,

  // The breathe effect slowly pulses all of the LEDs on your keyboard
  LEDBreatheEffect,

  // The HostPowerManagement plugin allows us to turn LEDs off when then host
  // goes to sleep, and resume them when it wakes up.
  HostPowerManagement,

  // The MagicCombo plugin lets you use key combinations to trigger custom
  // actions - a bit like Macros, but triggered by pressing multiple keys at the
  // same time.
  MagicCombo,

  // The USBQuirks plugin lets you do some things with USB that we aren't
  // comfortable - or able - to do automatically, but can be useful
  // nevertheless. Such as toggling the key report protocol between Boot (used
  // by BIOSes) and Report (NKRO).
  USBQuirks);

void setup() {
  Kaleidoscope.setup();
  Serial.begin(9600);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.6 Devices/Keyboardio/Model01/Model01.ino

```
,// -*- mode: c++ -*-
// Copyright 2016 Keyboardio, inc. <jesse@keyboard.io>
// See "LICENSE" for license details

#ifndef BUILD_INFORMATION
#define BUILD_INFORMATION "locally built on " __DATE__ " at " __TIME__
#endif


/**
 * These #include directives pull in the Kaleidoscope firmware core,
 * as well as the Kaleidoscope plugins we use in the Model 01's firmware
 */


// The Kaleidoscope core
#include "Kaleidoscope.h"

// Support for storing the keymap in EEPROM
#include "Kaleidoscope-EEPROM-Settings.h"
#include "Kaleidoscope-EEPROM-Keymap.h"

// Support for communicating with the host via a simple Serial protocol
#include "Kaleidoscope-FocusSerial.h"

// Support for querying the firmware version via Focus
#include "Kaleidoscope-FirmwareVersion.h"

// Support for setting the names of layers via Chrysalis
// #include "Kaleidoscope-LayerNames.h"

// Support for keys that move the mouse
#include "Kaleidoscope-MouseKeys.h"

// Support for macros & dynamic macros
#include "Kaleidoscope-Macros.h"
#include "Kaleidoscope-DynamicMacros.h"

// Support for controlling the keyboard's LEDs
#include "Kaleidoscope-LEDControl.h"

// Support for setting and saving the default LED mode
// #include "Kaleidoscope-DefaultLEDModeConfig.h"

// Support for "Numpad" mode, which is mostly just the Numpad specific LED mode
#include "Kaleidoscope-NumPad.h"

// Support for the "Boot greeting" effect, which pulses the 'LED' button for 10s
// when the keyboard is connected to a computer (or that computer is powered on)
#include "Kaleidoscope-LEDEffect-BootGreeting.h"
```

(continues on next page)

(continued from previous page)

```
// Support for LED modes that set all LEDs to a single color
#include "Kaleidoscope-LEDEffect-SolidColor.h"

// Support for an LED mode that makes all the LEDs 'breathe'
#include "Kaleidoscope-LEDEffect-Breathe.h"

// Support for an LED mode that makes a red pixel chase a blue pixel across the keyboard
#include "Kaleidoscope-LEDEffect-Chase.h"

// Support for LED modes that pulse the keyboard's LED in a rainbow pattern
#include "Kaleidoscope-LEDEffect-Rainbow.h"

// Support for shared palettes for other plugins, like Colormap below
#include "Kaleidoscope-LED-Palette-Theme.h"

// Support for an LED mode that lets one configure per-layer color maps
#include "Kaleidoscope-Colormap.h"

// Support for host power management (suspend & wakeup)
#include "Kaleidoscope-HostPowerManagement.h"

// Support for magic combos (key chords that trigger an action)
#include "Kaleidoscope-MagicCombo.h"

// Support for secondary actions (one action when tapped, another when held)
#include "Kaleidoscope-Qukeys.h"

// Support for SpaceCadet keys
// #include "Kaleidoscope-SpaceCadet.h"

// Support for one-shot modifiers and layer keys
// #include "Kaleidoscope-OneShot.h"
// #include "Kaleidoscope-Escape-OneShot.h"

// Support for USB quirks, like changing the key state report protocol
#include "Kaleidoscope-USB-Quirks.h"

/** This 'enum' is a list of all the macros used by the Model 01's firmware
  * The names aren't particularly important. What is important is that each
  * is unique.
  *
  * These are the names of your macros. They'll be used in two places.
  * The first is in your keymap definitions. There, you'll use the syntax
  * `M(MACRO_NAME)` to mark a specific keymap position as triggering `MACRO_NAME`
  *
  * The second usage is in the 'switch' statement in the `macroAction` function.
  * That switch statement actually runs the code associated with a macro when
  * a macro key is pressed.
  */

enum { MACRO_VERSION_INFO,
       MACRO_ANY
```

(continues on next page)

(continued from previous page)

```
};


/** The Model 01's key layouts are defined as 'keymaps'. By default, there are three
  * keymaps: The standard QWERTY keymap, the "Function layer" keymap and the "Numpad"
  * keymap.
  *
  * Each keymap is defined as a list using the 'KEYMAP_STACKED' macro, built
  * of first the left hand's layout, followed by the right hand's layout.
  *
  * Keymaps typically consist mostly of `Key_` definitions. There are many, many keys
  * defined as part of the USB HID Keyboard specification. You can find the names
  * (if not yet the explanations) for all the standard `Key_` defintions offered by
  * Kaleidoscope in these files:
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs_
↪keyboard.h
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs_
↪consumerctl.h
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs_
↪sysctl.h
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs_
↪keymaps.h
  *
  * Additional things that should be documented here include
  *   using ___ to let keypresses fall through to the previously active layer
  *   using XXX to mark a keyswitch as 'blocked' on this layer
  *   using ShiftToLayer() and LockLayer() keys to change the active keymap.
  *   keeping NUM and FN consistent and accessible on all layers
  *
  * The PROG key is special, since it is how you indicate to the board that you
  * want to flash the firmware. However, it can be remapped to a regular key.
  * When the keyboard boots, it first looks to see whether the PROG key is held
  * down; if it is, it simply awaits further flashing instructions. If it is
  * not, it continues loading the rest of the firmware and the keyboard
  * functions normally, with whatever binding you have set to PROG. More detail
  * here: https://community.keyboard.io/t/how-the-prog-key-gets-you-into-the-bootloader/
↪506/8
  *
  * The "keymaps" data structure is a list of the keymaps compiled into the firmware.
  * The order of keymaps in the list is important, as the ShiftToLayer(#) and LockLayer(
↪#)
  * macros switch to key layers based on this list.
  *
  *

  * A key defined as 'ShiftToLayer(FUNCTION)' will switch to FUNCTION while held.
  * Similarly, a key defined as 'LockLayer(NUMPAD)' will switch to NUMPAD when tapped.
  */

/**
  * Layers are "0-indexed" -- That is the first one is layer 0. The second one is layer␣
↪1.
```

(continues on next page)

(continued from previous page)

```
  * The third one is layer 2.
  * This 'enum' lets us use names like QWERTY, FUNCTION, and NUMPAD in place of
  * the numbers 0, 1 and 2.
  *
  */

enum { PRIMARY,
       NUMPAD,
       FUNCTION };  // layers


/**
  * To change your keyboard's layout from QWERTY to DVORAK or COLEMAK, comment out the␣
↪line
  *
  * #define PRIMARY_KEYMAP_QWERTY
  *
  * by changing it to
  *
  * // #define PRIMARY_KEYMAP_QWERTY
  *
  * Then uncomment the line corresponding to the layout you want to use.
  *
  */

#define PRIMARY_KEYMAP_QWERTY
// #define PRIMARY_KEYMAP_DVORAK
// #define PRIMARY_KEYMAP_COLEMAK
// #define PRIMARY_KEYMAP_CUSTOM


/* This comment temporarily turns off astyle's indent enforcement
 *   so we can make the keymaps actually resemble the physical key layout better
 */
// clang-format off

KEYMAPS(

#if defined (PRIMARY_KEYMAP_QWERTY)
  [PRIMARY] = KEYMAP_STACKED
  (___,          Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),  Key_6, Key_7, Key_8,     Key_9,         Key_0,         ␣
↪LockLayer(NUMPAD),
   Key_Enter,     Key_Y, Key_U, Key_I,     Key_O,         Key_P,         Key_Equals,
                  Key_H, Key_J, Key_K,     Key_L,         Key_Semicolon, Key_Quote,
   Key_RightAlt,  Key_N, Key_M, Key_Comma, Key_Period,    Key_Slash,     Key_Minus,
```

(continues on next page)

(continued from previous page)

```
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#elif defined (PRIMARY_KEYMAP_DVORAK)

  [PRIMARY] = KEYMAP_STACKED
  (___,           Key_1,         Key_2,     Key_3,      Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Quote,     Key_Comma, Key_Period, Key_P, Key_Y, Key_Tab,
   Key_PageUp,   Key_A,         Key_O,     Key_E,      Key_U, Key_I,
   Key_PageDown, Key_Semicolon, Key_Q,     Key_J,      Key_K, Key_X, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),   Key_6, Key_7, Key_8, Key_9, Key_0, LockLayer(NUMPAD),
   Key_Enter,      Key_F, Key_G, Key_C, Key_R, Key_L, Key_Slash,
                   Key_D, Key_H, Key_T, Key_N, Key_S, Key_Minus,
   Key_RightAlt,   Key_B, Key_M, Key_W, Key_V, Key_Z, Key_Equals,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#elif defined (PRIMARY_KEYMAP_COLEMAK)

  [PRIMARY] = KEYMAP_STACKED
  (___,           Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Q, Key_W, Key_F, Key_P, Key_G, Key_Tab,
   Key_PageUp,   Key_A, Key_R, Key_S, Key_T, Key_D,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),  Key_6, Key_7, Key_8,     Key_9,         Key_0,         ␣
↪LockLayer(NUMPAD),
   Key_Enter,     Key_J, Key_L, Key_U,     Key_Y,         Key_Semicolon, Key_Equals,
                  Key_H, Key_N, Key_E,     Key_I,         Key_O,         Key_Quote,
   Key_RightAlt,  Key_K, Key_M, Key_Comma, Key_Period,    Key_Slash,     Key_Minus,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#elif defined (PRIMARY_KEYMAP_CUSTOM)
  // Edit this keymap to make a custom layout
  [PRIMARY] = KEYMAP_STACKED
  (___,           Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),  Key_6, Key_7, Key_8,     Key_9,         Key_0,         ␣
↪LockLayer(NUMPAD),
   Key_Enter,     Key_Y, Key_U, Key_I,     Key_O,         Key_P,         Key_Equals,
                  Key_H, Key_J, Key_K,     Key_L,         Key_Semicolon, Key_Quote,
```

(continues on next page)

(continued from previous page)

```
   Key_RightAlt,  Key_N, Key_M, Key_Comma, Key_Period,    Key_Slash,     Key_Minus,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#else

#error "No default keymap defined. You should make sure that you have a line like '
↪#define PRIMARY_KEYMAP_QWERTY' in your sketch"

#endif



  [NUMPAD] =  KEYMAP_STACKED
  (___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___,
   ___,

   M(MACRO_VERSION_INFO),  ___, Key_7, Key_8,      Key_9,              Key_
↪KeypadSubtract, ___,
   ___,                    ___, Key_4, Key_5,      Key_6,              Key_KeypadAdd,   ␣
↪  ___,
                           ___, Key_1, Key_2,      Key_3,              Key_Equals,      ␣
↪  ___,
   ___,                    ___, Key_0, Key_Period, Key_KeypadMultiply, Key_KeypadDivide,␣
↪  Key_Enter,
   ___, ___, ___, ___,
   ___),

  [FUNCTION] =  KEYMAP_STACKED
  (___,      Key_F1,           Key_F2,      Key_F3,     Key_F4,        Key_F5,           ␣
↪Key_CapsLock,
   Key_Tab,  ___,              Key_mouseUp, ___,        Key_mouseBtnR, Key_mouseWarpEnd,␣
↪Key_mouseWarpNE,
   Key_Home, Key_mouseL,       Key_mouseDn, Key_mouseR, Key_mouseBtnL, Key_mouseWarpNW,
   Key_End,  Key_PrintScreen,  Key_Insert,  ___,        Key_mouseBtnM, Key_mouseWarpSW, ␣
↪Key_mouseWarpSE,
   ___, Key_Delete, ___, ___,
   ___,

   Consumer_ScanPreviousTrack, Key_F6,                 Key_F7,                    Key_F8,␣
↪                 Key_F9,          Key_F10,          Key_F11,
   Consumer_PlaySlashPause,    Consumer_ScanNextTrack, Key_LeftCurlyBracket,      Key_
↪RightCurlyBracket,    Key_LeftBracket, Key_RightBracket, Key_F12,
                               Key_LeftArrow,          Key_DownArrow,             Key_
↪UpArrow,              Key_RightArrow,  ___,              ___,
   Key_PcApplication,          Consumer_Mute,          Consumer_VolumeDecrement,␣
↪Consumer_VolumeIncrement, ___,             Key_Backslash,    Key_Pipe,
   ___, ___, Key_Enter, ___,
```

(continues on next page)

(continued from previous page)

```
   ___)
) // KEYMAPS(

/* Re-enable astyle's indent enforcement */
// clang-format on

/** versionInfoMacro handles the 'firmware version info' macro
 *  When a key bound to the macro is pressed, this macro
 *  prints out the firmware build information as virtual keystrokes
 */

static void versionInfoMacro(uint8_t key_state) {
  if (keyToggledOn(key_state)) {
    Macros.type(PSTR("Keyboardio Model 01 - Kaleidoscope "));
    Macros.type(PSTR(BUILD_INFORMATION));
  }
}

/** anyKeyMacro is used to provide the functionality of the 'Any' key.
 *
 * When the 'any key' macro is toggled on, a random alphanumeric key is
 * selected. While the key is held, the function generates a synthetic
 * keypress event repeating that randomly selected key.
 *
 */

static void anyKeyMacro(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    event.key.setKeyCode(Key_A.getKeyCode() + (uint8_t)(millis() % 36));
    event.key.setFlags(0);
  }
}


/** macroAction dispatches keymap events that are tied to a macro
    to that macro. It takes two uint8_t parameters.

    The first is the macro being called (the entry in the 'enum' earlier in this file).
    The second is the state of the keyswitch. You can use the keyswitch state to figure␣
↪out
    if the key has just been toggled on, is currently pressed or if it's just been␣
↪released.

    The 'switch' statement should have a 'case' for each entry of the macro enum.
    Each 'case' statement should call out to a function to handle the macro in question.

 */

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {

  case MACRO_VERSION_INFO:
```

(continues on next page)

(continued from previous page)

```
    versionInfoMacro(event.state);
    break;

  case MACRO_ANY:
    anyKeyMacro(event);
    break;
  }
  return MACRO_NONE;
}


// These 'solid' color effect definitions define a rainbow of
// LED color modes calibrated to draw 500mA or less on the
// Keyboardio Model 01.


static kaleidoscope::plugin::LEDSolidColor solidRed(160, 0, 0);
static kaleidoscope::plugin::LEDSolidColor solidOrange(140, 70, 0);
static kaleidoscope::plugin::LEDSolidColor solidYellow(130, 100, 0);
static kaleidoscope::plugin::LEDSolidColor solidGreen(0, 160, 0);
static kaleidoscope::plugin::LEDSolidColor solidBlue(0, 70, 130);
static kaleidoscope::plugin::LEDSolidColor solidIndigo(0, 0, 170);
static kaleidoscope::plugin::LEDSolidColor solidViolet(130, 0, 120);

/** toggleLedsOnSuspendResume toggles the LEDs off when the host goes to sleep,
 * and turns them back on when it wakes up.
 */
void toggleLedsOnSuspendResume(kaleidoscope::plugin::HostPowerManagement::Event event) {
  switch (event) {
  case kaleidoscope::plugin::HostPowerManagement::Suspend:
  case kaleidoscope::plugin::HostPowerManagement::Sleep:
    LEDControl.disable();
    break;
  case kaleidoscope::plugin::HostPowerManagement::Resume:
    LEDControl.enable();
    break;
  }
}

/** hostPowerManagementEventHandler dispatches power management events (suspend,
 * resume, and sleep) to other functions that perform action based on these
 * events.
 */
void hostPowerManagementEventHandler(kaleidoscope::plugin::HostPowerManagement::Event␣
↪event) {
  toggleLedsOnSuspendResume(event);
}

/** This 'enum' is a list of all the magic combos used by the Model 01's
 * firmware The names aren't particularly important. What is important is that
 * each is unique.
 *
```

(continues on next page)

(continued from previous page)

```
 * These are the names of your magic combos. They will be used by the
 * `USE_MAGIC_COMBOS` call below.
 */
enum {
  // Toggle between Boot (6-key rollover; for BIOSes and early boot) and NKRO
  // mode.
  COMBO_TOGGLE_NKRO_MODE,
  // Enter test mode
  COMBO_ENTER_TEST_MODE
};

/** Wrappers, to be used by MagicCombo. **/

/**
 * This simply toggles the keyboard protocol via USBQuirks, and wraps it within
 * a function with an unused argument, to match what MagicCombo expects.
 */
static void toggleKeyboardProtocol(uint8_t combo_index) {
  USBQuirks.toggleKeyboardProtocol();
}

/** Magic combo list, a list of key combo and action pairs the firmware should
 * recognise.
 */
USE_MAGIC_COMBOS({.action = toggleKeyboardProtocol,
                  // Left Fn + Esc + Shift
                  .keys = {R3C6, R2C6, R3C7}});

// First, tell Kaleidoscope which plugins you want to use.
// The order can be important. For example, LED effects are
// added in the order they're listed here.
KALEIDOSCOPE_INIT_PLUGINS(
  // ----------------------------------------------------------------------
  // Chrysalis plugins

  // The EEPROMSettings & EEPROMKeymap plugins make it possible to have an
  // editable keymap in EEPROM.
  EEPROMSettings,
  EEPROMKeymap,

  // Focus allows bi-directional communication with the host, and is the
  // interface through which the keymap in EEPROM can be edited.
  Focus,

  // FocusSettingsCommand adds a few Focus commands, intended to aid in
  // changing some settings of the keyboard, such as the default layer (via the
  // `settings.defaultLayer` command)
  FocusSettingsCommand,

  // FocusEEPROMCommand adds a set of Focus commands, which are very helpful in
  // both debugging, and in backing up one's EEPROM contents.
  FocusEEPROMCommand,
```

(continues on next page)

(continued from previous page)

```
  // The FirmwareVersion plugin lets Chrysalis query the version of the firmware
  // programmatically.
  FirmwareVersion,

  // The LayerNames plugin allows Chrysalis to display - and edit - custom layer
  // names, to be shown instead of the default indexes.
  // LayerNames,

  // Enables setting, saving (via Chrysalis), and restoring (on boot) the
  // default LED mode.
  // DefaultLEDModeConfig,

  // ----------------------------------------------------------------------
  // Keystroke-handling plugins

  // The Qukeys plugin enables the "Secondary action" functionality in
  // Chrysalis. Keys with secondary actions will have their primary action
  // performed when tapped, but the secondary action when held.
  Qukeys,

  // SpaceCadet can turn your shifts into parens on tap, while keeping them as
  // Shifts when held. SpaceCadetConfig lets Chrysalis configure some aspects of
  // the plugin.
  // SpaceCadet,
  // SpaceCadetConfig,

  // Enables the "Sticky" behavior for modifiers, and the "Layer shift when
  // held" functionality for layer keys.
  // OneShot,
  // OneShotConfig,
  // EscapeOneShot,
  // EscapeOneShotConfig,

  // The macros plugin adds support for macros
  Macros,

  // Enables dynamic, Chrysalis-editable macros.
  DynamicMacros,

  // The MouseKeys plugin lets you add keys to your keymap which move the mouse.
  MouseKeys,
  // MouseKeysConfig,

  // The MagicCombo plugin lets you use key combinations to trigger custom
  // actions - a bit like Macros, but triggered by pressing multiple keys at the
  // same time.
  MagicCombo,

  // Enables the GeminiPR Stenography protocol. Unused by default, but with the
  // plugin enabled, it becomes configurable - and then usable - via Chrysalis.
  // GeminiPR,
```

(continues on next page)

(continued from previous page)

```
// ----------------------------------------------------------------------
// LED mode plugins

// The boot greeting effect pulses the LED button for 10 seconds after the
// keyboard is first connected
BootGreetingEffect,

// LEDControl provides support for other LED modes
LEDControl,

// We start with the LED effect that turns off all the LEDs.
LEDOff,

// The rainbow effect changes the color of all of the keyboard's keys at the same time
// running through all the colors of the rainbow.
LEDRainbowEffect,

// The rainbow wave effect lights up your keyboard with all the colors of a rainbow
// and slowly moves the rainbow across your keyboard
LEDRainbowWaveEffect,

// The chase effect follows the adventure of a blue pixel which chases a red pixel␣
↪across
// your keyboard. Spoiler: the blue pixel never catches the red pixel
LEDChaseEffect,

// These static effects turn your keyboard's LEDs a variety of colors
solidRed,
solidOrange,
solidYellow,
solidGreen,
solidBlue,
solidIndigo,
solidViolet,

// The breathe effect slowly pulses all of the LEDs on your keyboard
LEDBreatheEffect,

// The AlphaSquare effect prints each character you type, using your
// keyboard's LEDs as a display
// AlphaSquareEffect,

// The stalker effect lights up the keys you've pressed recently
// StalkerEffect,

// The LED Palette Theme plugin provides a shared palette for other plugins,
// like Colormap below
LEDPaletteTheme,

// The Colormap effect makes it possible to set up per-layer colormaps
ColormapEffect,
```

(continues on next page)

(continued from previous page)

```
  // The numpad plugin is responsible for lighting up the 'numpad' mode
  // with a custom LED effect
  NumPad,

  // The HostPowerManagement plugin allows us to turn LEDs off when then host
  // goes to sleep, and resume them when it wakes up.
  HostPowerManagement,

  // Turns LEDs off after a configurable amount of idle time.
  // IdleLEDs,
  // PersistentIdleLEDs,

  // ----------------------------------------------------------------------
  // Miscellaneous plugins

  // The USBQuirks plugin lets you do some things with USB that we aren't
  // comfortable - or able - to do automatically, but can be useful
  // nevertheless. Such as toggling the key report protocol between Boot (used
  // by BIOSes) and Report (NKRO).
  USBQuirks);

/** The 'setup' function is one of the two standard Arduino sketch functions.
 * It's called when your keyboard first powers up. This is where you set up
 * Kaleidoscope and any plugins.
 */
void setup() {
  // First, call Kaleidoscope's internal setup function
  Kaleidoscope.setup();

  // While we hope to improve this in the future, the NumPad plugin
  // needs to be explicitly told which keymap layer is your numpad layer
  NumPad.numPadLayer = NUMPAD;

  // We set the brightness of the rainbow effects to 150 (on a scale of 0-255)
  // This draws more than 500mA, but looks much nicer than a dimmer effect
  LEDRainbowEffect.brightness(150);
  LEDRainbowWaveEffect.brightness(150);

  // We want to make sure that the firmware starts with LED effects off
  // This avoids over-taxing devices that don't have a lot of power to share
  // with USB devices
  LEDOff.activate();

  // To make the keymap editable without flashing new firmware, we store
  // additional layers in EEPROM. For now, we reserve space for five layers. If
  // one wants to use these layers, just set the default layer to one in EEPROM,
  // by using the `settings.defaultLayer` Focus command, or by using the
  // `keymap.onlyCustom` command to use EEPROM layers only.
  EEPROMKeymap.setup(5);

  // We need to tell the Colormap plugin how many layers we want to have custom
```

(continues on next page)

(continued from previous page)

```
  // maps for. To make things simple, we set it to five layers, which is how
  // many editable layers we have (see above).
  ColormapEffect.max_layers(5);

  // For Dynamic Macros, we need to reserve storage space for the editable
  // macros.
  DynamicMacros.reserve_storage(128);

  // If there's a default layer set in EEPROM, we should set that as the default
  // here.
  Layer.move(EEPROMSettings.default_layer());
}

/** loop is the second of the standard Arduino sketch functions.
  * As you might expect, it runs in a loop, never exiting.
  *
  * For Kaleidoscope-based keyboard firmware, you usually just want to
  * call Kaleidoscope.loop(); and not do anything custom here.
  */

void loop() {
  Kaleidoscope.loop();
}
```

11.22.7 Devices/Keyboardio/Model100/Model100.ino

```
,// -*- mode: c++ -*-
// Copyright 2016-2022 Keyboardio, inc. <jesse@keyboard.io>
// See "LICENSE" for license details

/**
 * These #include directives pull in the Kaleidoscope firmware core,
 * as well as the Kaleidoscope plugins we use in the Model 100's firmware
 */

// The Kaleidoscope core
#include "Kaleidoscope.h"

// Support for storing the keymap in EEPROM
#include "Kaleidoscope-EEPROM-Settings.h"
#include "Kaleidoscope-EEPROM-Keymap.h"

// Support for communicating with the host via a simple Serial protocol
#include "Kaleidoscope-FocusSerial.h"

// Support for querying the firmware version via Focus
#include "Kaleidoscope-FirmwareVersion.h"

// Support for keys that move the mouse
#include "Kaleidoscope-MouseKeys.h"
```

(continues on next page)

(continued from previous page)

```
// Support for macros
#include "Kaleidoscope-Macros.h"

// Support for controlling the keyboard's LEDs
#include "Kaleidoscope-LEDControl.h"

// Support for "Numpad" mode, which is mostly just the Numpad specific LED mode
#include "Kaleidoscope-NumPad.h"

// Support for the "Boot greeting" effect, which pulses the 'LED' button for 10s
// when the keyboard is connected to a computer (or that computer is powered on)
#include "Kaleidoscope-LEDEffect-BootGreeting.h"

// Support for LED modes that set all LEDs to a single color
#include "Kaleidoscope-LEDEffect-SolidColor.h"

// Support for an LED mode that makes all the LEDs 'breathe'
#include "Kaleidoscope-LEDEffect-Breathe.h"

// Support for an LED mode that makes a red pixel chase a blue pixel across the keyboard
#include "Kaleidoscope-LEDEffect-Chase.h"

// Support for LED modes that pulse the keyboard's LED in a rainbow pattern
#include "Kaleidoscope-LEDEffect-Rainbow.h"

// Support for an LED mode that lights up the keys as you press them
#include "Kaleidoscope-LED-Stalker.h"

// Support for an LED mode that prints the keys you press in letters 4px high
#include "Kaleidoscope-LED-AlphaSquare.h"

// Support for shared palettes for other plugins, like Colormap below
#include "Kaleidoscope-LED-Palette-Theme.h"

// Support for an LED mode that lets one configure per-layer color maps
#include "Kaleidoscope-Colormap.h"

// Support for turning the LEDs off after a certain amount of time
#include "Kaleidoscope-IdleLEDs.h"

// Support for setting and saving the default LED mode
#include "Kaleidoscope-DefaultLEDModeConfig.h"

// Support for changing the brightness of the LEDs
#include "Kaleidoscope-LEDBrightnessConfig.h"

// Support for Keyboardio's internal keyboard testing mode
#include "Kaleidoscope-HardwareTestMode.h"

// Support for host power management (suspend & wakeup)
#include "Kaleidoscope-HostPowerManagement.h"
```

(continues on next page)

(continued from previous page)

```
// Support for magic combos (key chords that trigger an action)
#include "Kaleidoscope-MagicCombo.h"

// Support for USB quirks, like changing the key state report protocol
#include "Kaleidoscope-USB-Quirks.h"

// Support for secondary actions on keys
#include "Kaleidoscope-Qukeys.h"

// Support for one-shot modifiers and layer keys
#include "Kaleidoscope-OneShot.h"
#include "Kaleidoscope-Escape-OneShot.h"

// Support for dynamic, Chrysalis-editable macros
#include "Kaleidoscope-DynamicMacros.h"

// Support for SpaceCadet keys
#include "Kaleidoscope-SpaceCadet.h"

// Support for editable layer names
#include "Kaleidoscope-LayerNames.h"

// Support for the GeminiPR Stenography protocol
#include "Kaleidoscope-Steno.h"

/** This 'enum' is a list of all the macros used by the Model 100's firmware
  * The names aren't particularly important. What is important is that each
  * is unique.
  *
  * These are the names of your macros. They'll be used in two places.
  * The first is in your keymap definitions. There, you'll use the syntax
  * `M(MACRO_NAME)` to mark a specific keymap position as triggering `MACRO_NAME`
  *
  * The second usage is in the 'switch' statement in the `macroAction` function.
  * That switch statement actually runs the code associated with a macro when
  * a macro key is pressed.
  */

enum {
  MACRO_VERSION_INFO,
  MACRO_ANY,
};


/** The Model 100's key layouts are defined as 'keymaps'. By default, there are three
  * keymaps: The standard QWERTY keymap, the "Function layer" keymap and the "Numpad"
  * keymap.
  *
  * Each keymap is defined as a list using the 'KEYMAP_STACKED' macro, built
  * of first the left hand's layout, followed by the right hand's layout.
  *
```

(continues on next page)

(continued from previous page)

```
  * Keymaps typically consist mostly of `Key_` definitions. There are many, many keys
  * defined as part of the USB HID Keyboard specification. You can find the names
  * (if not yet the explanations) for all the standard `Key_` defintions offered by
  * Kaleidoscope in these files:
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs/
↪keyboard.h
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs/
↪consumerctl.h
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs/
↪sysctl.h
  *    https://github.com/keyboardio/Kaleidoscope/blob/master/src/kaleidoscope/key_defs/
↪keymaps.h
  *
  * Additional things that should be documented here include
  *   using ___ to let keypresses fall through to the previously active layer
  *   using XXX to mark a keyswitch as 'blocked' on this layer
  *   using ShiftToLayer() and LockLayer() keys to change the active keymap.
  *   keeping NUM and FN consistent and accessible on all layers
  *
  * The PROG key is special, since it is how you indicate to the board that you
  * want to flash the firmware. However, it can be remapped to a regular key.
  * When the keyboard boots, it first looks to see whether the PROG key is held
  * down; if it is, it simply awaits further flashing instructions. If it is
  * not, it continues loading the rest of the firmware and the keyboard
  * functions normally, with whatever binding you have set to PROG. More detail
  * here: https://community.keyboard.io/t/how-the-prog-key-gets-you-into-the-bootloader/
↪506/8
  *
  * The "keymaps" data structure is a list of the keymaps compiled into the firmware.
  * The order of keymaps in the list is important, as the ShiftToLayer(#) and LockLayer(
↪#)
  * macros switch to key layers based on this list.
  *
  *


  * A key defined as 'ShiftToLayer(FUNCTION)' will switch to FUNCTION while held.
  * Similarly, a key defined as 'LockLayer(NUMPAD)' will switch to NUMPAD when tapped.
  */

/**
  * Layers are "0-indexed" -- That is the first one is layer 0. The second one is layer␣
↪1.
  * The third one is layer 2.
  * This 'enum' lets us use names like QWERTY, FUNCTION, and NUMPAD in place of
  * the numbers 0, 1 and 2.
  *
  */

enum {
  PRIMARY,
  NUMPAD,
  FUNCTION,
```

(continues on next page)

(continued from previous page)

```
};  // layers


/**
  * To change your keyboard's layout from QWERTY to DVORAK or COLEMAK, comment out the␣
↪line
  *
  * #define PRIMARY_KEYMAP_QWERTY
  *
  * by changing it to
  *
  * // #define PRIMARY_KEYMAP_QWERTY
  *
  * Then uncomment the line corresponding to the layout you want to use.
  *
  */

#define PRIMARY_KEYMAP_QWERTY
// #define PRIMARY_KEYMAP_DVORAK
// #define PRIMARY_KEYMAP_COLEMAK
// #define PRIMARY_KEYMAP_CUSTOM


/* This comment temporarily turns off astyle's indent enforcement
 *   so we can make the keymaps actually resemble the physical key layout better
 */
// clang-format off

KEYMAPS(

#if defined (PRIMARY_KEYMAP_QWERTY)
  [PRIMARY] = KEYMAP_STACKED
  (___,          Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),  Key_6, Key_7, Key_8,     Key_9,         Key_0,         ␣
↪LockLayer(NUMPAD),
   Key_Enter,     Key_Y, Key_U, Key_I,     Key_O,         Key_P,         Key_Equals,
                  Key_H, Key_J, Key_K,     Key_L,         Key_Semicolon, Key_Quote,
   Key_RightAlt,  Key_N, Key_M, Key_Comma, Key_Period,    Key_Slash,     Key_Minus,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#elif defined (PRIMARY_KEYMAP_DVORAK)

  [PRIMARY] = KEYMAP_STACKED
  (___,          Key_1,         Key_2,     Key_3,      Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Quote,     Key_Comma, Key_Period, Key_P, Key_Y, Key_Tab,
```

(continues on next page)

(continued from previous page)

```
   Key_PageUp,   Key_A,         Key_O,     Key_E,      Key_U, Key_I,
   Key_PageDown, Key_Semicolon, Key_Q,     Key_J,      Key_K, Key_X, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),   Key_6, Key_7, Key_8, Key_9, Key_0, LockLayer(NUMPAD),
   Key_Enter,      Key_F, Key_G, Key_C, Key_R, Key_L, Key_Slash,
                   Key_D, Key_H, Key_T, Key_N, Key_S, Key_Minus,
   Key_RightAlt,   Key_B, Key_M, Key_W, Key_V, Key_Z, Key_Equals,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#elif defined (PRIMARY_KEYMAP_COLEMAK)

  [PRIMARY] = KEYMAP_STACKED
  (___,          Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Q, Key_W, Key_F, Key_P, Key_B, Key_Tab,
   Key_PageUp,   Key_A, Key_R, Key_S, Key_T, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_D, Key_V, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),  Key_6, Key_7, Key_8,     Key_9,         Key_0,         ␣
↪LockLayer(NUMPAD),
   Key_Enter,     Key_J, Key_L, Key_U,     Key_Y,         Key_Semicolon, Key_Equals,
                  Key_M, Key_N, Key_E,     Key_I,         Key_O,         Key_Quote,
   Key_RightAlt,  Key_K, Key_H, Key_Comma, Key_Period,    Key_Slash,     Key_Minus,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#elif defined (PRIMARY_KEYMAP_CUSTOM)
  // Edit this keymap to make a custom layout
  [PRIMARY] = KEYMAP_STACKED
  (___,          Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   M(MACRO_ANY),  Key_6, Key_7, Key_8,     Key_9,         Key_0,         ␣
↪LockLayer(NUMPAD),
   Key_Enter,     Key_Y, Key_U, Key_I,     Key_O,         Key_P,         Key_Equals,
                  Key_H, Key_J, Key_K,     Key_L,         Key_Semicolon, Key_Quote,
   Key_RightAlt,  Key_N, Key_M, Key_Comma, Key_Period,    Key_Slash,     Key_Minus,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),

#else

#error "No default keymap defined. You should make sure that you have a line like '
↪#define PRIMARY_KEYMAP_QWERTY' in your sketch"
```

(continues on next page)

(continued from previous page)

```
#endif



  [NUMPAD] =  KEYMAP_STACKED
  (___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___,
   ___,

   M(MACRO_VERSION_INFO),  ___, Key_7, Key_8,      Key_9,              Key_
↪KeypadSubtract, ___,
   ___,                    ___, Key_4, Key_5,      Key_6,              Key_KeypadAdd,   ␣
↪  ___,
                           ___, Key_1, Key_2,      Key_3,              Key_Equals,      ␣
↪  ___,
   ___,                    ___, Key_0, Key_Period, Key_KeypadMultiply, Key_KeypadDivide,␣
↪  Key_Enter,
   ___, ___, ___, ___,
   ___),

  [FUNCTION] =  KEYMAP_STACKED
  (___,      Key_F1,           Key_F2,      Key_F3,     Key_F4,        Key_F5,           ␣
↪Key_CapsLock,
   Key_Tab,  ___,              Key_mouseUp, ___,        Key_mouseBtnR, Key_mouseWarpEnd,␣
↪Key_mouseWarpNE,
   Key_Home, Key_mouseL,       Key_mouseDn, Key_mouseR, Key_mouseBtnL, Key_mouseWarpNW,
   Key_End,  Key_PrintScreen,  Key_Insert,  ___,        Key_mouseBtnM, Key_mouseWarpSW, ␣
↪Key_mouseWarpSE,
   ___, Key_Delete, ___, ___,
   ___,

   Consumer_ScanPreviousTrack, Key_F6,                 Key_F7,                   Key_F8,␣
↪                 Key_F9,          Key_F10,          Key_F11,
   Consumer_PlaySlashPause,    Consumer_ScanNextTrack, Key_LeftCurlyBracket,     Key_
↪RightCurlyBracket,    Key_LeftBracket, Key_RightBracket, Key_F12,
                               Key_LeftArrow,          Key_DownArrow,            Key_
↪UpArrow,              Key_RightArrow,  ___,              ___,
   Key_PcApplication,          Consumer_Mute,          Consumer_VolumeDecrement,␣
↪Consumer_VolumeIncrement, ___,             Key_Backslash,    Key_Pipe,
   ___, ___, Key_Enter, ___,
   ___)
) // KEYMAPS(

/* Re-enable astyle's indent enforcement */
// clang-format on

/** versionInfoMacro handles the 'firmware version info' macro
 *  When a key bound to the macro is pressed, this macro
```

(continues on next page)

(continued from previous page)

```
 *  prints out the firmware build information as virtual keystrokes
 */

static void versionInfoMacro(uint8_t key_state) {
  if (keyToggledOn(key_state)) {
    Macros.type(PSTR("Keyboardio Model 100 - Firmware version "));
    Macros.type(PSTR(KALEIDOSCOPE_FIRMWARE_VERSION));
  }
}

/** anyKeyMacro is used to provide the functionality of the 'Any' key.
 *
 * When the 'any key' macro is toggled on, a random alphanumeric key is
 * selected. While the key is held, the function generates a synthetic
 * keypress event repeating that randomly selected key.
 *
 */

static void anyKeyMacro(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    event.key.setKeyCode(Key_A.getKeyCode() + (uint8_t)(millis() % 36));
    event.key.setFlags(0);
  }
}


/** macroAction dispatches keymap events that are tied to a macro
    to that macro. It takes two uint8_t parameters.

    The first is the macro being called (the entry in the 'enum' earlier in this file).
    The second is the state of the keyswitch. You can use the keyswitch state to figure␣
↪out
    if the key has just been toggled on, is currently pressed or if it's just been␣
↪released.

    The 'switch' statement should have a 'case' for each entry of the macro enum.
    Each 'case' statement should call out to a function to handle the macro in question.

 */

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {

  case MACRO_VERSION_INFO:
    versionInfoMacro(event.state);
    break;

  case MACRO_ANY:
    anyKeyMacro(event);
    break;
  }
  return MACRO_NONE;
```

(continues on next page)

(continued from previous page)

```
}


// These 'solid' color effect definitions define a rainbow of
// LED color modes calibrated to draw 500mA or less on the
// Keyboardio Model 100.


static kaleidoscope::plugin::LEDSolidColor solidRed(160, 0, 0);
static kaleidoscope::plugin::LEDSolidColor solidOrange(140, 70, 0);
static kaleidoscope::plugin::LEDSolidColor solidYellow(130, 100, 0);
static kaleidoscope::plugin::LEDSolidColor solidGreen(0, 160, 0);
static kaleidoscope::plugin::LEDSolidColor solidBlue(0, 70, 130);
static kaleidoscope::plugin::LEDSolidColor solidIndigo(0, 0, 170);
static kaleidoscope::plugin::LEDSolidColor solidViolet(130, 0, 120);

/** toggleLedsOnSuspendResume toggles the LEDs off when the host goes to sleep,
 * and turns them back on when it wakes up.
 */
void toggleLedsOnSuspendResume(kaleidoscope::plugin::HostPowerManagement::Event event) {
  switch (event) {
  case kaleidoscope::plugin::HostPowerManagement::Suspend:
  case kaleidoscope::plugin::HostPowerManagement::Sleep:
    LEDControl.disable();
    break;
  case kaleidoscope::plugin::HostPowerManagement::Resume:
    LEDControl.enable();
    break;
  }
}

/** hostPowerManagementEventHandler dispatches power management events (suspend,
 * resume, and sleep) to other functions that perform action based on these
 * events.
 */
void hostPowerManagementEventHandler(kaleidoscope::plugin::HostPowerManagement::Event␣
↪event) {
  toggleLedsOnSuspendResume(event);
}

/** This 'enum' is a list of all the magic combos used by the Model 100's
 * firmware The names aren't particularly important. What is important is that
 * each is unique.
 *
 * These are the names of your magic combos. They will be used by the
 * `USE_MAGIC_COMBOS` call below.
 */
enum {
  // Toggle between Boot (6-key rollover; for BIOSes and early boot) and NKRO
  // mode.
  COMBO_TOGGLE_NKRO_MODE,
  // Enter test mode
```

(continues on next page)

(continued from previous page)

```
  COMBO_ENTER_TEST_MODE
};

/** Wrappers, to be used by MagicCombo. **/

/**
 * This simply toggles the keyboard protocol via USBQuirks, and wraps it within
 * a function with an unused argument, to match what MagicCombo expects.
 */
static void toggleKeyboardProtocol(uint8_t combo_index) {
  USBQuirks.toggleKeyboardProtocol();
}

/**
 * Toggles between using the built-in keymap, and the EEPROM-stored one.
 */
static void toggleKeymapSource(uint8_t combo_index) {
  if (Layer.getKey == Layer.getKeyFromPROGMEM) {
    Layer.getKey = EEPROMKeymap.getKey;
  } else {
    Layer.getKey = Layer.getKeyFromPROGMEM;
  }
}

/**
 *  This enters the hardware test mode
 */
static void enterHardwareTestMode(uint8_t combo_index) {
  HardwareTestMode.runTests();
}


/** Magic combo list, a list of key combo and action pairs the firmware should
 * recognise.
 */
USE_MAGIC_COMBOS({.action = toggleKeyboardProtocol,
                  // Left Fn + Esc + Shift
                  .keys = {R3C6, R2C6, R3C7}},
                 {.action = enterHardwareTestMode,
                  // Left Fn + Prog + LED
                  .keys = {R3C6, R0C0, R0C6}},
                 {.action = toggleKeymapSource,
                  // Left Fn + Prog + Shift
                  .keys = {R3C6, R0C0, R3C7}});

// First, tell Kaleidoscope which plugins you want to use.
// The order can be important. For example, LED effects are
// added in the order they're listed here.
KALEIDOSCOPE_INIT_PLUGINS(
  // ----------------------------------------------------------------------
  // Chrysalis plugins
```

(continues on next page)

(continued from previous page)

```
  // The EEPROMSettings & EEPROMKeymap plugins make it possible to have an
  // editable keymap in EEPROM.
  EEPROMSettings,
  EEPROMKeymap,

  // Focus allows bi-directional communication with the host, and is the
  // interface through which the keymap in EEPROM can be edited.
  Focus,

  // FocusSettingsCommand adds a few Focus commands, intended to aid in
  // changing some settings of the keyboard, such as the default layer (via the
  // `settings.defaultLayer` command)
  FocusSettingsCommand,

  // FocusEEPROMCommand adds a set of Focus commands, which are very helpful in
  // both debugging, and in backing up one's EEPROM contents.
  FocusEEPROMCommand,

  // The FirmwareVersion plugin lets Chrysalis query the version of the firmware
  // programmatically.
  FirmwareVersion,

  // The LayerNames plugin allows Chrysalis to display - and edit - custom layer
  // names, to be shown instead of the default indexes.
  LayerNames,

  // Enables setting, saving (via Chrysalis), and restoring (on boot) the
  // default LED mode.
  DefaultLEDModeConfig,

  // Enables controlling (and saving) the brightness of the LEDs via Focus.
  LEDBrightnessConfig,

  // ----------------------------------------------------------------------
  // Keystroke-handling plugins

  // The Qukeys plugin enables the "Secondary action" functionality in
  // Chrysalis. Keys with secondary actions will have their primary action
  // performed when tapped, but the secondary action when held.
  Qukeys,

  // SpaceCadet can turn your shifts into parens on tap, while keeping them as
  // Shifts when held. SpaceCadetConfig lets Chrysalis configure some aspects of
  // the plugin.
  SpaceCadet,
  SpaceCadetConfig,

  // Enables the "Sticky" behavior for modifiers, and the "Layer shift when
  // held" functionality for layer keys.
  OneShot,
  OneShotConfig,
  EscapeOneShot,
```

(continues on next page)

(continued from previous page)

```
EscapeOneShotConfig,

// The macros plugin adds support for macros
Macros,

// Enables dynamic, Chrysalis-editable macros.
DynamicMacros,

// The MouseKeys plugin lets you add keys to your keymap which move the mouse.
MouseKeys,
MouseKeysConfig,

// The MagicCombo plugin lets you use key combinations to trigger custom
// actions - a bit like Macros, but triggered by pressing multiple keys at the
// same time.
MagicCombo,

// Enables the GeminiPR Stenography protocol. Unused by default, but with the
// plugin enabled, it becomes configurable - and then usable - via Chrysalis.
GeminiPR,

// ---------------------------------------------------------------------
// LED mode plugins

// The boot greeting effect pulses the LED button for 10 seconds after the
// keyboard is first connected
BootGreetingEffect,

// LEDControl provides support for other LED modes
LEDControl,

// We start with the LED effect that turns off all the LEDs.
LEDOff,

// The rainbow effect changes the color of all of the keyboard's keys at the same time
// running through all the colors of the rainbow.
LEDRainbowEffect,

// The rainbow wave effect lights up your keyboard with all the colors of a rainbow
// and slowly moves the rainbow across your keyboard
LEDRainbowWaveEffect,

// The chase effect follows the adventure of a blue pixel which chases a red pixel␣
↪across
// your keyboard. Spoiler: the blue pixel never catches the red pixel
LEDChaseEffect,

// These static effects turn your keyboard's LEDs a variety of colors
solidRed,
solidOrange,
solidYellow,
solidGreen,
```

(continues on next page)

(continued from previous page)

```
  solidBlue,
  solidIndigo,
  solidViolet,

  // The breathe effect slowly pulses all of the LEDs on your keyboard
  LEDBreatheEffect,

  // The AlphaSquare effect prints each character you type, using your
  // keyboard's LEDs as a display
  AlphaSquareEffect,

  // The stalker effect lights up the keys you've pressed recently
  StalkerEffect,

  // The LED Palette Theme plugin provides a shared palette for other plugins,
  // like Colormap below
  LEDPaletteTheme,

  // The Colormap effect makes it possible to set up per-layer colormaps
  ColormapEffect,

  // The numpad plugin is responsible for lighting up the 'numpad' mode
  // with a custom LED effect
  NumPad,

  // The HostPowerManagement plugin allows us to turn LEDs off when then host
  // goes to sleep, and resume them when it wakes up.
  HostPowerManagement,

  // Turns LEDs off after a configurable amount of idle time.
  IdleLEDs,
  PersistentIdleLEDs,

  // ---------------------------------------------------------------------
  // Miscellaneous plugins

  // The USBQuirks plugin lets you do some things with USB that we aren't
  // comfortable - or able - to do automatically, but can be useful
  // nevertheless. Such as toggling the key report protocol between Boot (used
  // by BIOSes) and Report (NKRO).
  USBQuirks,

  // The hardware test mode, which can be invoked by tapping Prog, LED and the
  // left Fn button at the same time.
  HardwareTestMode  //,
);

/** The 'setup' function is one of the two standard Arduino sketch functions.
 * It's called when your keyboard first powers up. This is where you set up
 * Kaleidoscope and any plugins.
 */
void setup() {
```

(continues on next page)

(continued from previous page)

```
  // First, call Kaleidoscope's internal setup function
  Kaleidoscope.setup();

  // Set the hue of the boot greeting effect to something that will result in a
  // nice green color.
  BootGreetingEffect.hue = 85;

  // While we hope to improve this in the future, the NumPad plugin
  // needs to be explicitly told which keymap layer is your numpad layer
  NumPad.numPadLayer = NUMPAD;

  // We configure the AlphaSquare effect to use RED letters
  AlphaSquare.color = CRGB(255, 0, 0);

  // Set the rainbow effects to be reasonably bright, but low enough
  // to mitigate audible noise in some environments.
  LEDRainbowEffect.brightness(170);
  LEDRainbowWaveEffect.brightness(160);

  // Set the action key the test mode should listen for to Left Fn
  HardwareTestMode.setActionKey(R3C6);

  // The LED Stalker mode has a few effects. The one we like is called
  // 'BlazingTrail'. For details on other options, see
  // https://github.com/keyboardio/Kaleidoscope/blob/master/docs/plugins/LED-Stalker.md
  StalkerEffect.variant = STALKER(BlazingTrail);

  // To make the keymap editable without flashing new firmware, we store
  // additional layers in EEPROM. For now, we reserve space for eight layers. If
  // one wants to use these layers, just set the default layer to one in EEPROM,
  // by using the `settings.defaultLayer` Focus command, or by using the
  // `keymap.onlyCustom` command to use EEPROM layers only.
  EEPROMKeymap.setup(8);

  // We need to tell the Colormap plugin how many layers we want to have custom
  // maps for. To make things simple, we set it to eight layers, which is how
  // many editable layers we have (see above).
  ColormapEffect.max_layers(8);

  // For Dynamic Macros, we need to reserve storage space for the editable
  // macros. A kilobyte is a reasonable default.
  DynamicMacros.reserve_storage(1024);

  // If there's a default layer set in EEPROM, we should set that as the default
  // here.
  Layer.move(EEPROMSettings.default_layer());

  // To avoid any surprises, SpaceCadet is turned off by default. However, it
  // can be permanently enabled via Chrysalis, so we should only disable it if
  // no configuration exists.
  SpaceCadetConfig.disableSpaceCadetIfUnconfigured();
```

(continues on next page)

(continued from previous page)

```
  // Editable layer names are stored in EEPROM too, and we reserve 16 bytes per
  // layer for them. We need one extra byte per layer for bookkeeping, so we
  // reserve 17 / layer in total.
  LayerNames.reserve_storage(17 * 8);

  // Unless configured otherwise with Chrysalis, we want to make sure that the
  // firmware starts with LED effects off. This avoids over-taxing devices that
  // don't have a lot of power to share with USB devices
  DefaultLEDModeConfig.activateLEDModeIfUnconfigured(&LEDOff);
}

/** loop is the second of the standard Arduino sketch functions.
  * As you might expect, it runs in a loop, never exiting.
  *
  * For Kaleidoscope-based keyboard firmware, you usually just want to
  * call Kaleidoscope.loop(); and not do anything custom here.
  */

void loop() {
  Kaleidoscope.loop();
}
```

11.22.8 Devices/OLKB/Planck/Planck.ino

```
,/* -*- mode: c++ -*-
 * Planck -- A very basic Kaleidoscope example for the OLKB Planck keyboard
 * Copyright (C) 2018  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTabILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-Macros.h"

enum {
  _QWERTY,
  _COLEMAK,
  _DVORAK,
```

(continues on next page)

(continued from previous page)

```
  _LOWER,
  _RAISE,
  _PLOVER,
  _ADJUST
};


// clang-format off
KEYMAPS(

/* Qwerty
 * ,-----------------------------------------------------------------------------------.
 * | Tab  |   Q  |   W  |   E  |   R  |   T  |   Y  |   U  |   I  |   O  |   P  | Bksp |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * | Esc  |   A  |   S  |   D  |   F  |   G  |   H  |   J  |   K  |   L  |   ;  |  "   |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * | Shift|   Z  |   X  |   C  |   V  |   B  |   N  |   M  |   ,  |   .  |   /  |Enter |
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * | Brite| Ctrl | Alt  | GUI  |Lower |    Space    |Raise | Left | Down |  Up  |Right |
 * `-----------------------------------------------------------------------------------'
 */


[_QWERTY] = KEYMAP(
    Key_Tab,  Key_Q,    Key_W,    Key_E,    Key_R,    Key_T,    Key_Y,    Key_U,    Key_
↪I,    Key_O,    Key_P,    Key_Backspace,
    Key_Escape,  Key_A,    Key_S,    Key_D,    Key_F,    Key_G,    Key_H,    Key_J,   ␣
↪Key_K,    Key_L,    Key_Semicolon, Key_Quote,
    Key_LeftShift, Key_Z,    Key_X,    Key_C,    Key_V,    Key_B,    Key_N,    Key_M,   ␣
↪Key_Comma, Key_Period,  Key_Slash, Key_Enter ,
    ___, Key_LeftControl, Key_LeftAlt, Key_LeftGui, LockLayer(_LOWER),   Key_Space,  Key_
↪Space,  LockLayer(_RAISE),   Key_LeftArrow, Key_DownArrow, Key_UpArrow,   Key_
↪RightArrow

),
/* Colemak
 * ,-----------------------------------------------------------------------------------.
 * | Tab  |   Q  |   W  |   F  |   P  |   G  |   J  |   L  |   U  |   Y  |   ;  | Bksp |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * | Esc  |   A  |   R  |   S  |   T  |   D  |   H  |   N  |   E  |   I  |   O  |  "   |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * | Shift|   Z  |   X  |   C  |   V  |   B  |   K  |   M  |   ,  |   .  |   /  |Enter |
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * | Brite| Ctrl | Alt  | GUI  |Lower |    Space    |Raise | Left | Down |  Up  |Right |
 * `-----------------------------------------------------------------------------------'
 */
[_COLEMAK] = KEYMAP(
    Key_Tab,  Key_Q,    Key_W,    Key_F,    Key_P,    Key_G,    Key_J,    Key_L,    Key_
↪U,    Key_Y,    Key_Semicolon, Key_Backspace,
    Key_Escape,  Key_A,    Key_R,    Key_S,    Key_T,    Key_D,    Key_H,    Key_N,   ␣
↪Key_E,    Key_I,    Key_O,    Key_Quote,
    Key_LeftShift, Key_Z,    Key_X,    Key_C,    Key_V,    Key_B,    Key_K,    Key_M,   ␣
```

(continues on next page)

(continued from previous page)

```
↪Key_Comma, Key_Period,  Key_Slash, Key_Enter ,
    ___, Key_LeftControl, Key_LeftAlt, Key_LeftGui, ShiftToLayer(_LOWER),   Key_Space, ␣
↪Key_Space,  ShiftToLayer(_RAISE),   Key_LeftArrow, Key_DownArrow, Key_UpArrow,   Key_
↪RightArrow
),

/* Dvorak
 * ,-----------------------------------------------------------------------------------.
 * | Tab  |   "  |   ,  |   .  |   P  |   Y  |   F  |   G  |   C  |   R  |   L  | Bksp |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * | Esc  |   A  |   O  |   E  |   U  |   I  |   D  |   H  |   T  |   N  |   S  |  /   |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * | Shift|   ;  |   Q  |   J  |   K  |   X  |   B  |   M  |   W  |   V  |   Z  |Enter |
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * | Brite| Ctrl | Alt  | GUI  |Lower |    Space    |Raise | Left | Down |  Up  |Right |
 * `-----------------------------------------------------------------------------------'
 */
[_DVORAK] = KEYMAP(
    Key_Tab,  Key_Quote, Key_Comma, Key_Period,  Key_P,    Key_Y,    Key_F,    Key_G,   ␣
↪Key_C,    Key_R,    Key_L,    Key_Backspace,
    Key_Escape,  Key_A,    Key_O,    Key_E,    Key_U,    Key_I,    Key_D,    Key_H,   ␣
↪Key_T,    Key_N,    Key_S,    Key_Slash,
    Key_LeftShift, Key_Semicolon, Key_Q,    Key_J,    Key_K,    Key_X,    Key_B,    Key_
↪M,    Key_W,    Key_V,    Key_Z,    Key_Enter ,
    ___, Key_LeftControl, Key_LeftAlt, Key_LeftGui, LockLayer(_LOWER),   Key_Space,  Key_
↪Space,  LockLayer(_RAISE),   Key_LeftArrow, Key_DownArrow, Key_UpArrow,   Key_
↪RightArrow
),
/* Lower
 * ,-----------------------------------------------------------------------------------.
 * |   ~  |   !  |   @  |   #  |   $  |   %  |   ^  |   &  |   *  |   (  |   )  | Bksp |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * | Del  |  F1  |  F2  |  F3  |  F4  |  F5  |  F6  |   _  |   +  |   {  |   }  |  |   |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * |      |  F7  |  F8  |  F9  |  F10 |  F11 |  F12 |ISO ~ |ISO | | Home | End  |      |
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * |      |      |      |      |      |             |      | Next | Vol- | Vol+ | Play |
 * `-----------------------------------------------------------------------------------'
 */
[_LOWER] = KEYMAP(
    LSHIFT(Key_Backtick),  LSHIFT(Key_1),    LSHIFT(Key_2),    LSHIFT(Key_3),    ␣
↪LSHIFT(Key_4),    LSHIFT(Key_5),    LSHIFT(Key_6),    LSHIFT(Key_7),    LSHIFT(Key_8),␣
↪    LSHIFT(Key_9),    LSHIFT(Key_0), Key_Backspace,
    Key_Delete,  Key_F1,    Key_F2,    Key_F3,    Key_F4,    Key_F5,    Key_F6,    LSHIFT(Key_
↪Minus),    LSHIFT(Key_Equals),    Key_LeftBracket, Key_RightBracket, Key_Pipe,
    ___, Key_F7,    Key_F8,    Key_F9,    Key_F10,   Key_F11,   Key_F12,  LSHIFT(Key_
↪NonUsPound), LSHIFT(Key_NonUsBackslashAndPipe), Key_Home, Key_End,  ___,
    ___, ___, ___, ___, ___, ___, ___, ___,     Consumer_ScanNextTrack,    Consumer_
↪VolumeDecrement, Consumer_VolumeIncrement, Consumer_PlaySlashPause
),

/* Raise
```

(continues on next page)

(continued from previous page)

```
 * ,-----------------------------------------------------------------------------------.
 * |   `  |   1  |   2  |   3  |   4  |   5  |   6  |   7  |   8  |   9  |   0  | Bksp |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * | Del  |  F1  |  F2  |  F3  |  F4  |  F5  |  F6  |   -  |   =  |   [  |   ]  |  \   |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * |      |  F7  |  F8  |  F9  |  F10 |  F11 |  F12 |ISO # |ISO / |Pg Up |Pg Dn |      |
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * |      |      |      |      |      |             |      | Next | Vol- | Vol+ | Play |
 * `-----------------------------------------------------------------------------------'
 */
[_RAISE] = KEYMAP(
    Key_Backtick,  Key_1,    Key_2,    Key_3,    Key_4,    Key_5,    Key_6,    Key_7,    ␣
↪Key_8,    Key_9,    Key_0,    Key_Backspace,
    Key_Delete,  Key_F1,   Key_F2,   Key_F3,   Key_F4,   Key_F5,   Key_F6,   Key_Minus,␣
↪Key_Equals,  Key_LeftBracket, Key_RightBracket, Key_Backslash,
    ___, Key_F7,   Key_F8,   Key_F9,   Key_F10,  Key_F11,  Key_F12,  Key_NonUsPound, Key_
↪NonUsBackslashAndPipe, Key_PageUp, Key_PageDown, ___,
    ___, ___, ___, ___, ___, ___, ___, ___, Consumer_ScanNextTrack, Consumer_
↪VolumeDecrement, Consumer_VolumeIncrement, Consumer_PlaySlashPause
),

/* Plover layer (http://opensteno.org)
 * ,-----------------------------------------------------------------------------------.
 * |   #  |   #  |   #  |   #  |   #  |   #  |   #  |   #  |   #  |   #  |   #  |   #  |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * |      |   S  |   T  |   P  |   H  |   *  |   *  |   F  |   P  |   L  |   T  |   D  |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * |      |   S  |   K  |   W  |   R  |   *  |   *  |   R  |   B  |   G  |   S  |   Z  |
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * | Exit |      |      |   A  |   O  |             |   E  |   U  |      |      |      |
 * `-----------------------------------------------------------------------------------'
 */

[_PLOVER] = KEYMAP(
    LSHIFT(Key_1),    LSHIFT(Key_1),    LSHIFT(Key_1),    LSHIFT(Key_1),    LSHIFT(Key_
↪1),    LSHIFT(Key_1),    LSHIFT(Key_1),    LSHIFT(Key_1),    LSHIFT(Key_1),   ␣
↪LSHIFT(Key_1),    LSHIFT(Key_1),    LSHIFT(Key_1)    ,
    XXX, Key_Q,    Key_W,    Key_E,    Key_R,    Key_T,    Key_Y,    Key_U,    Key_I,    ␣
↪Key_O,    Key_P,    Key_LeftBracket,
    XXX, Key_A,    Key_S,    Key_D,    Key_F,    Key_G,    Key_H,    Key_J,    Key_K,    ␣
↪Key_L,    Key_Semicolon, Key_Quote,
    LockLayer(_QWERTY), XXX, XXX, Key_C,    Key_V,    XXX, XXX, Key_N,    Key_M,    XXX,␣
↪XXX, XXX
),

/* Adjust (Lower + Raise)
 * ,-----------------------------------------------------------------------------------.
 * |      | Reset|      |      |      |      |      |      |      |      |      |  Del |
 * |------+------+------+------+------+-------------+------+------+------+------+------|
 * |      |      |      |Aud on|Audoff|AGnorm|AGswap|Qwerty|Colemk|Dvorak|Plover|      |
 * |------+------+------+------+------+------|------+------+------+------+------+------|
 * |      |Voice-|Voice+|Mus on|Musoff|MIDIon|MIDIof|      |      |      |      |      |
```

(continues on next page)

(continued from previous page)

```
 * |------+------+------+------+------+------+------+------+------+------+------+------|
 * |      |      |      |      |      |             |      |      |      |      |      |
 * `-----------------------------------------------------------------------------------'
 */
//[_ADJUST] = KEYMAP(
//    ___, RESET,   DEBUG,    RGB_TOG, RGB_MOD, RGB_HUI, RGB_HUD, RGB_SAI, RGB_SAD, RGB_
↪VAI, RGB_VAD, Key_Delete ,
//    ___, ___, MU_MOD,  AU_ON,   AU_OFF,  AG_NORM, AG_SWAP, LockLayer(_QWERTY), ␣
↪LockLayer(_COLEMAK), LockLayer(_DVORAK),  LockLayer(_PLOVER),  ___,
//    ___, MUV_DE,  MUV_IN,  MU_ON,   MU_OFF,  MI_ON,   MI_OFF,  TERM_ON, TERM_OFF, ___,␣
↪___, ___,
//    ___, ___, ___, ___, ___, ___, ___, ___, ___, ___, ___, ___
//)
);
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(Macros);

void setup() {
  Kaleidoscope.setup();
  Kaleidoscope.serialPort().begin(9600);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.9 Devices/SOFTHRUF/Splitography/Splitography.ino

```
,/* -*- mode: c++ -*-
 * Splitography-Sketch -- A complete, functional sketch for Splitography
 * Copyright (C) 2018-2022  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 * Modeled after the default QMK layout:
 *  https://github.com/sdothum/qmk_firmware/blob/
↪d865c82efa19beb7cb593e7d3affb2311017833e/keyboards/splitography/keymaps/default/keymap.
↪c
```

(continues on next page)

(continued from previous page)

```
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-Escape-OneShot.h"
#include "Kaleidoscope-DynamicMacros.h"
#include "Kaleidoscope-EEPROM-Settings.h"
#include "Kaleidoscope-EEPROM-Keymap.h"
#include "Kaleidoscope-FirmwareVersion.h"
#include "Kaleidoscope-FocusSerial.h"
#include "Kaleidoscope-MouseKeys.h"
#include "Kaleidoscope-OneShot.h"
#include "Kaleidoscope-Ranges.h"
#include "Kaleidoscope-Qukeys.h"
#include "Kaleidoscope-SpaceCadet.h"
#include "Kaleidoscope-Steno.h"

// Layers
enum {
  _QWERTY,
  _BLUE,
  _ORANGE,
  _GREEN,
  _STENO,
  _PLOVER
};

// Custom keys
enum {
  QWERTY_1 = kaleidoscope::ranges::SAFE_START,
  QWERTY_2
};

#define QWERTY1   Key(QWERTY_1)
#define QWERTY2   Key(QWERTY_2)

#define MO(layer) ShiftToLayer(layer)
#define TO(layer) LockLayer(layer)

#define K_STP     Consumer_Stop
#define K_PRV     Consumer_ScanPreviousTrack
#define K_NXT     Consumer_ScanNextTrack
#define K_PLY     Consumer_PlaySlashPause

// Key aliases
#define Key_PgUp   Key_PageUp
#define Key_PageDn Key_PageDown
#define Key_PgDn   Key_PageDown
#define Key_Del    Key_Delete
#define Key_Grave  Key_Backtick
#define K_APP      Key_PcApplication
#define K_SCRLK    Key_ScrollLock
#define K_CPSLK    Key_CapsLock
```

(continues on next page)

(continued from previous page)

```
#define K_PAUSE    Key_Pause
#define K_PRSCR    Key_PrintScreen
#define K_MUTE     Consumer_Mute
#define K_VUp      Consumer_VolumeIncrement
#define K_VDn      Consumer_VolumeDecrement
#define K_PST      LCTRL(Key_V)
#define K_CPY      LCTRL(Key_LeftControl)
#define K_CUT      LCTRL(Key_X)
#define K_UDO      LCTRL(Key_Z)

#define KP_0       Key_Keypad0
#define KP_1       Key_Keypad1
#define KP_2       Key_Keypad2
#define KP_3       Key_Keypad3
#define KP_4       Key_Keypad4
#define KP_5       Key_Keypad5
#define KP_6       Key_Keypad6
#define KP_7       Key_Keypad7
#define KP_8       Key_Keypad8
#define KP_9       Key_Keypad9

#define Key_Up     Key_UpArrow
#define Key_Dn     Key_DownArrow
#define Key_Left   Key_LeftArrow
#define Key_Rgt    Key_RightArrow
#define KP_SLS     Key_KeypadDivide
#define KP_STR     Key_KeypadMultiply
#define Key_Plus   Key_KeypadAdd

// clang-format off
KEYMAPS(
/* QWERTY
 * ,--------------------------.      ,--------------------------.
 * | Esc | Q | W | E | R | T |      | Y | U | I | O | P | Bspc |
 * |-----+---+---+---+---+---|      |---+---+---+---+---+------|
 * | Alt | A | S | D | F | G |      | H | J | K | L | ; | Entr |
 * |-----+---+---+---+---+---|      |---+---+---+---+---+------|
 * | Sft | Z | X | C | V | B |      | N | M | , | . | / | Gui  |
 * `-------------+---+---+---'     `-+---+---+-----------------'
 *               |ORG|BLU|           |SPC|CTR|
 *               `-------'          `-------'
 */
 [_QWERTY] = KEYMAP(
    Key_Esc       ,Key_Q ,Key_W ,Key_E ,Key_R ,Key_T    ,Key_Y ,Key_U ,Key_I      ,Key_O ␣
↪     ,Key_P          ,Key_Backspace
   ,Key_LeftAlt    ,Key_A ,Key_S ,Key_D ,Key_F ,Key_G    ,Key_H ,Key_J ,Key_K      ,Key_L ␣
↪     ,Key_Semicolon ,Key_Enter
   ,Key_LeftShift ,Key_Z ,Key_X ,Key_C ,Key_V ,Key_B    ,Key_N ,Key_M ,Key_Comma ,Key_
↪Period ,Key_Slash     ,Key_LeftGui
                             ,MO(_ORANGE) ,MO(_BLUE)    ,Key_Space ,Key_LeftControl
  ),
```

(continues on next page)

(continued from previous page)

```
/* Blue
 * ,-----------------------------------------.      ,----------------------------.
 * |  `  |  1   |  2   |  3   |   4   |  5   |      |  6   | 7 | 8 | 9 | 0 |     |
 * |-----+------+------+------+-------+------|      |------+---+---+---+---+-----|
 * | Alt | Stop | Prev | Play | Next  | Vol+ |      |      |   |   | [ | ] |  '  |
 * |-----+------+------+------+-------+------|      |------+---+---+---+---+-----|
 * | Sft | Undo | Cut  | Copy | Paste | Vol- |      | Mute |   |   | - | = | Gui |
 * `----------------------+------+------+----'     `-+---+---+------------------'
 *                        | GRN  |  []  |             |Del|CTR|
 *                        `-------------'            `-------'
 */
 [_BLUE] = KEYMAP(
    Key_Grave     ,Key_1 ,Key_2 ,Key_3 ,Key_4 ,Key_5    ,Key_6  ,Key_7 ,Key_8 ,Key_9     ␣
↪       ,Key_0              ,XXX
   ,Key_LeftAlt   ,K_STP ,K_PRV ,K_PLY ,K_NXT ,K_VUp    ,XXX    ,XXX   ,XXX   ,Key_
↪LeftBracket ,Key_RightBracket ,Key_Quote
   ,Key_LeftShift ,K_UDO ,K_CUT ,K_CPY ,K_PST ,K_VDn    ,K_MUTE ,XXX   ,XXX   ,Key_Minus␣
↪       ,Key_Equals       ,Key_LeftGui
                              ,MO(_GREEN) ,___           ,Key_Del ,Key_LeftControl
 ),

/* Orange
 * ,------------------------------------------.      ,-----------------------------------
↪------.
 * | Plvr |  F1  |  F2  |  F3  |  F4   |      |      |  App | PrScr | ScrLck | Pause |  ␣
↪|     |
 * |------+------+------+------+-------+------|      |------+-------+--------+-------+---
↪+-----|
 * | Alt  |  F5  |  F6  |  F7  |  F8   |      |      |      |  Ins  |  Home  | PgUp  |  ␣
↪|     |
 * |------+------+------+------+-------+------|      |------+-------+--------+-------+---
↪+-----|
 * | Sft  |  F9  |  F10 |  F11 |  F12  |      |      |      |  Del  |  End   | PgDn  | \␣
↪| Gui |
 * `----------------------+------+------+-----'     `-+---+---+--------------------------
↪-----'
 *                        |  []  | GRN  |              |Tab|CTR|
 *                        `-------------'             `-------'
 */
 [_ORANGE] = KEYMAP(
    TO(_PLOVER)   ,Key_F1 ,Key_F2  ,Key_F3  ,Key_F4  ,XXX    ,K_APP ,K_PRSCR    ,K_SCRLK␣
↪ ,K_PAUSE    ,XXX           ,XXX
   ,Key_LeftAlt   ,Key_F5 ,Key_F6  ,Key_F7  ,Key_F8  ,XXX    ,XXX    ,Key_Insert ,Key_
↪Home ,Key_PageUp ,XXX           ,XXX
   ,Key_LeftShift ,Key_F9 ,Key_F10 ,Key_F11 ,Key_F12 ,XXX    ,XXX    ,Key_Delete ,Key_End␣
↪ ,Key_PageDn ,Key_Backslash ,Key_LeftGui
                                     ,___   ,MO(_GREEN)          ,Key_Tab ,Key_
↪LeftControl
 ),

/* Green
 * ,----------------------------------------.      ,--------------------------.
```

(continues on next page)

(continued from previous page)

```
 * | STENO |       |     |      |       | ScrLk |      | / | 7 | 8 | 9 | - |      |
 * |-------+-------+-----+------+-------+-------|      |---+---+---+---+---+------|
 * | Alt   | Home  | Up  | End  | PgUp  | CpsLk |      | * | 4 | 5 | 6 | + | Entr |
 * |-------+-------+-----+------+-------+-------|      |---+---+---+---+---+------|
 * | Sft   | Left  | Dn  | Rgt  | PgDn  |       |      | 0 | 1 | 2 | 3 |   | Gui  |
 * `-----------------------------+----+----+--'     `-+---+---+----------------'
 *                              | [] | [] |           |   |CTR|
 *                              `---------'           `-------'
 */

 [_GREEN] = KEYMAP(
    TO(_STENO)    ,XXX       ,XXX     ,XXX      ,XXX       ,K_SCRLK     ,KP_SLS ,KP_7   ,KP_8 ,
↪KP_9 ,Key_Minus     ,XXX
   ,Key_LeftAlt   ,Key_Home ,Key_Up ,Key_End ,Key_PgUp ,K_CPSLK     ,KP_STR ,KP_4   ,KP_5 ,
↪KP_6 ,Key_Plus      ,Key_Enter
   ,Key_LeftShift ,Key_Left ,Key_Dn ,Key_Rgt ,Key_PgDn ,XXX         ,KP_0   ,KP_1   ,KP_2 ,
↪KP_3 ,XXX           ,Key_LeftGui
                                            ,___          ,___              ,XXX   ,Key_
↪LeftControl
  ),

/* Steno (GeminiPR)
 * ,-----------------------.      ,-----------------------.
 * | # | # | # | # | # | # |      | # | # | # | # | # | # |
 * |---+---+---+---+---+---|      |---+---+---+---+---+---|
 * |QWR| S | T | P | H | * |      | * | F | P | L | T | D |
 * |---+---+---+---+---+---|      |---+---+---+---+---+---|
 * |QWR| S | K | W | R | * |      | * | R | B | G | S | Z |
 * `-------------+---+---+-'     `-+---+---+-------------'
 *              | A | O |         | E | U |
 *              `-------'         `-------'
 */
 [_STENO] = KEYMAP(
    S(N1)    ,S(N2) ,S(N3) ,S(N4) ,S(N5) ,S(N6)    ,S(N7)  ,S(N8) ,S(N9) ,S(NA) ,S(NB) ,
↪S(NC)
   ,QWERTY1  ,S(S1) ,S(TL) ,S(PL) ,S(HL) ,S(ST1)   ,S(ST3) ,S(FR) ,S(PR) ,S(LR) ,S(TR) ,
↪S(DR)
   ,QWERTY2  ,S(S2) ,S(KL) ,S(WL) ,S(RL) ,S(ST2)   ,S(ST4) ,S(RR) ,S(BR) ,S(GR) ,S(SR) ,
↪S(ZR)
                                   ,S(A)  ,S(O)     ,S(E)   ,S(U)
  ),

/* Steno (Keyboard, QWERTY)
 * ,-----------------------.      ,-----------------------.
 * | 1 | 1 | 1 | 1 | 1 | 1 |      | 1 | 1 | 1 | 1 | 1 | 1 |
 * |---+---+---+---+---+---|      |---+---+---+---+---+---|
 * |QWR| S | T | P | H | * |      | * | F | P | L | T | D |
 * |---+---+---+---+---+---|      |---+---+---+---+---+---|
 * |QWR| S | K | W | R | * |      | * | R | B | G | S | Z |
 * `-------------+---+---+-'     `-+---+---+-------------'
 *              | A | O |         | E | U |
 *              `-------'         `-------'
```

(continues on next page)

(continued from previous page)

```
 */
 [_PLOVER] = KEYMAP(
    Key_1     ,Key_1 ,Key_1 ,Key_1 ,Key_1 ,Key_1     ,Key_1 ,Key_1 ,Key_1 ,Key_1 ,Key_1   ␣
↪     ,Key_1
   ,QWERTY1   ,Key_Q ,Key_W ,Key_E ,Key_R ,Key_T     ,Key_Y ,Key_U ,Key_I ,Key_O ,Key_P   ␣
↪     ,Key_LeftBracket
   ,QWERTY2   ,Key_A ,Key_S ,Key_D ,Key_F ,Key_G     ,Key_H ,Key_J ,Key_K ,Key_L ,Key_
↪Semicolon ,Key_Quote
                                  ,Key_C ,Key_V     ,Key_N ,Key_M
  )
);
// clang-format on

namespace kaleidoscope {
namespace plugin {
class MultiSwitcher : public kaleidoscope::Plugin {
 public:
  MultiSwitcher() {}

  EventHandlerResult onKeyEvent(KeyEvent &event) {
    if (event.key < QWERTY_1 || event.key > QWERTY_2)
      return EventHandlerResult::OK;

    uint8_t bit = event.key.getRaw() - QWERTY_1;

    if (keyToggledOn(event.state)) {
      switch_state_ |= (1 << bit);

      if (switch_state_ == (1 << 0 | 1 << 1)) {
        Layer.move(_QWERTY);
      }
    } else {
      switch_state_ &= ~(1 << bit);
    }

    return EventHandlerResult::EVENT_CONSUMED;
  }

 private:
  uint8_t switch_state_ = 0;
};
}  // namespace plugin
}  // namespace kaleidoscope

kaleidoscope::plugin::MultiSwitcher MultiSwitcher;

KALEIDOSCOPE_INIT_PLUGINS(
  EscapeOneShot,
  GeminiPR,
  MultiSwitcher,
  Focus,
  EEPROMSettings,
```

(continues on next page)

(continued from previous page)

```
  EEPROMKeymap,
  FocusEEPROMCommand,
  FocusSettingsCommand,
  Qukeys,
  SpaceCadet,
  OneShot,
  MouseKeys,
  EscapeOneShotConfig,
  DynamicMacros,
  FirmwareVersion);

void setup() {
  Kaleidoscope.setup();
  EEPROMKeymap.setup(6);
  SpaceCadet.disable();
  DynamicMacros.reserve_storage(256);
  Layer.move(EEPROMSettings.default_layer());
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.10 Devices/Technomancy/Atreus/Atreus.ino

```
,/* -*- mode: c++ -*-
 * Atreus -- Chrysalis-enabled Sketch for Technomancy's Atreus
 * Copyright (C) 2018-2022  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-DynamicMacros.h"
#include "Kaleidoscope-EEPROM-Settings.h"
#include "Kaleidoscope-EEPROM-Keymap.h"
#include "Kaleidoscope-Escape-OneShot.h"
#include "Kaleidoscope-FirmwareVersion.h"
```

(continues on next page)

(continued from previous page)

```
#include "Kaleidoscope-FocusSerial.h"
#include "Kaleidoscope-Macros.h"
#include "Kaleidoscope-MouseKeys.h"
#include "Kaleidoscope-OneShot.h"
#include "Kaleidoscope-Qukeys.h"
#include "Kaleidoscope-SpaceCadet.h"

#define MO(n) ShiftToLayer(n)
#define TG(n) LockLayer(n)

enum {
  RESET,
  QW
};

#define Key_Exclamation LSHIFT(Key_1)
#define Key_At          LSHIFT(Key_2)
#define Key_Hash        LSHIFT(Key_3)
#define Key_Dollar      LSHIFT(Key_4)
#define Key_And         LSHIFT(Key_7)
#define Key_Star        LSHIFT(Key_8)
#define Key_Plus        LSHIFT(Key_Equals)

enum {
  _QW,
  _RS,
  _LW
};

// clang-format off
KEYMAPS(
  [_QW] = KEYMAP_STACKED
  (
       Key_Q   ,Key_W   ,Key_E       ,Key_R         ,Key_T
      ,Key_A   ,Key_S   ,Key_D       ,Key_F         ,Key_G
      ,Key_Z   ,Key_X   ,Key_C       ,Key_V         ,Key_B
      ,Key_Esc ,Key_Tab ,Key_LeftGui ,Key_LeftShift ,Key_Backspace ,Key_LeftControl

                     ,Key_Y     ,Key_U   ,Key_I     ,Key_O      ,Key_P
                     ,Key_H     ,Key_J   ,Key_K     ,Key_L      ,Key_Semicolon
                     ,Key_N     ,Key_M   ,Key_Comma ,Key_Period ,Key_Slash
       ,Key_LeftAlt ,Key_Space ,MO(_RS) ,Key_Minus ,Key_Quote  ,Key_Enter
  ),

  [_RS] = KEYMAP_STACKED
  (
       Key_Exclamation ,Key_At           ,Key_UpArrow   ,Key_LeftCurlyBracket ,Key_
↪RightCurlyBracket
      ,Key_Hash        ,Key_LeftArrow    ,Key_DownArrow ,Key_RightArrow       ,Key_Dollar
      ,Key_LeftBracket ,Key_RightBracket ,Key_LeftParen ,Key_RightParen       ,Key_And
      ,TG(_LW)         ,Key_Insert       ,Key_LeftGui   ,Key_LeftShift        ,Key_
↪Backspace         ,Key_LeftControl
```

(continues on next page)

(continued from previous page)

```
                   ,Key_PageUp   ,Key_7 ,Key_8      ,Key_9 ,Key_Star
                   ,Key_PageDown ,Key_4 ,Key_5      ,Key_6 ,Key_Plus
                   ,Key_Backtick ,Key_1 ,Key_2      ,Key_3 ,Key_Backslash
      ,Key_LeftAlt ,Key_Space    ,___   ,Key_Period ,Key_0 ,Key_Equals
   ),

  [_LW] = KEYMAP_STACKED
  (
       Key_Insert ,Key_Home                 ,Key_UpArrow   ,Key_End        ,Key_PageUp
      ,Key_Delete ,Key_LeftArrow            ,Key_DownArrow ,Key_RightArrow ,Key_PageDown
      ,XXX        ,Consumer_VolumeIncrement ,XXX           ,XXX            ,M(RESET)
      ,XXX        ,Consumer_VolumeDecrement ,___           ,___            ,___         ␣
↪,___

                ,Key_UpArrow   ,Key_F7 ,Key_F8          ,Key_F9         ,Key_F10
                ,Key_DownArrow ,Key_F4 ,Key_F5          ,Key_F6         ,Key_F11
                ,XXX           ,Key_F1 ,Key_F2          ,Key_F3         ,Key_F12
      ,___      ,___           ,M(QW)  ,Key_PrintScreen ,Key_ScrollLock ,Consumer_
↪PlaySlashPause
   )
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(
  EscapeOneShot,
  EEPROMSettings,
  EEPROMKeymap,
  Focus,
  FocusEEPROMCommand,
  FocusSettingsCommand,
  Macros,
  Qukeys,
  SpaceCadet,
  OneShot,
  MouseKeys,
  EscapeOneShotConfig,
  DynamicMacros,
  FirmwareVersion);

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {
  case RESET:
    Kaleidoscope.rebootBootloader();
    break;
  case QW:
    Layer.move(_QW);
    break;
  default:
    break;
  }
```

(continues on next page)

(continued from previous page)

```
  return MACRO_NONE;
}

void setup() {
  Kaleidoscope.setup();

  EEPROMKeymap.setup(5);
  SpaceCadet.disable();
  DynamicMacros.reserve_storage(256);
  Layer.move(EEPROMSettings.default_layer());
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.11 Devices/gHeavy/ButterStick/ButterStick.ino

```
,/* -*- mode: c++ -*-
 * kaleidoscope::device::gheavy::ButterStick -- gHeavy ButterStick hardware support for␣
↪Kaleidoscope
 * Copyright (C) 2020  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTabILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "Kaleidoscope.h"

enum {
  _DEFAULT
};


// clang-format off
KEYMAPS(
    [_DEFAULT] = KEYMAP(
      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Y, Key_U, Key_I, Key_O, Key_P,
      Key_A, Key_S, Key_D, Key_F, Key_G, Key_H, Key_J, Key_K, Key_L, Key_Semicolon
```

(continues on next page)

(continued from previous page)

```
    )
);
// clang-format on

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.12 Devices/gHeavy/FaunchPad/FaunchPad.ino

```
/* -*- mode: c++ -*-
 * kaleidoscope::device::gheavy::FaunchPad -- gHeavy FaunchPad hardware support for␣
↪Kaleidoscope
 * Copyright (C) 2020  Keyboard.io, Inc
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTabILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

#include "Kaleidoscope.h"

enum {
  _DEFAULT
};


// clang-format off
KEYMAPS(
    [_DEFAULT] = KEYMAP(
      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Y, Key_U, Key_I
    )
);
// clang-format on

void setup() {
```

(continues on next page)

(continued from previous page)

```
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.13 Features/AppSwitcher/AppSwitcher.cpp

```
,/* -*- mode: c++ -*-
 * AppSwitcher -- A Kaleidoscope Example
 * Copyright (C) 2021  Keyboardio, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#define KALEIDOSCOPE_HOSTOS_GUESSER 1

#include <Kaleidoscope-HostOS.h>

#include "AppSwitcher.h"

namespace kaleidoscope {
namespace plugin {

EventHandlerResult AppSwitcher::onKeyEvent(KeyEvent &event) {
  // Ignore all key releases
  if (keyToggledOff(event.state))
    return EventHandlerResult::OK;

  if (event.key == AppSwitcher_Next || event.key == AppSwitcher_Prev) {
    bool add_shift_flag = false;
    if (event.key == AppSwitcher_Prev) {
      add_shift_flag = true;
    }

    // For good measure:
    event.state |= INJECTED;

    // If AppSwitcher was not already active, hold its modifier first.
```

(continues on next page)

(continued from previous page)

```
    if (!active_addr_.isValid()) {
      if (::HostOS.os() == hostos::MACOS) {
        event.key = Key_LeftGui;
      } else {
        event.key = Key_LeftAlt;
      }
      Runtime.handleKeyEvent(event);
    }

    // Clear the event's key address so we don't clobber the modifier.
    event.addr.clear();
    event.key = Key_Tab;
    if (add_shift_flag)
      event.key.setFlags(SHIFT_HELD);
    // Press tab
    Runtime.handleKeyEvent(event);
    // Change state to release; this will get processed when we return OK below.
    event.state = WAS_PRESSED | INJECTED;
  } else if (active_addr_.isValid()) {
    // If any non-AppSwitcher key is pressed while AppSwitcher is active, that
    // will close AppSwitcher instead of processing that keypress. We mask the
    // address of the key that closed AppSwitcher so that its release doesn't
    // have any effect. Then we turn the event for that key's press into an
    // event for the release of the AppSwitcher's modifier key.
    live_keys.mask(event.addr);
    event.addr  = active_addr_;
    event.state = WAS_PRESSED | INJECTED;
    event.key   = live_keys[event.addr];
    // Turn off AppSwitcher:
    active_addr_.clear();
  }
  return EventHandlerResult::OK;
}

}  // namespace plugin
}  // namespace kaleidoscope

kaleidoscope::plugin::AppSwitcher AppSwitcher;
```

11.22.14 Features/AppSwitcher/AppSwitcher.h

```
,/* -*- mode: c++ -*-
 * AppSwitcher -- A Kaleidoscope Example
 * Copyright (C) 2021  Keyboardio, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
```

(continues on next page)

(continued from previous page)

```
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#pragma once

#include <Kaleidoscope.h>
#include "Kaleidoscope-Ranges.h"

constexpr Key AppSwitcher_Next{kaleidoscope::ranges::SAFE_START};
constexpr uint16_t _prev_val = AppSwitcher_Next.getRaw() + 1;
constexpr Key AppSwitcher_Prev{_prev_val};

namespace kaleidoscope {
namespace plugin {

class AppSwitcher : public kaleidoscope::Plugin {

 public:
  EventHandlerResult onKeyEvent(KeyEvent &event);

 private:
  KeyAddr active_addr_ = KeyAddr::none();
};

}  // namespace plugin
}  // namespace kaleidoscope

extern kaleidoscope::plugin::AppSwitcher AppSwitcher;
```

11.22.15 Features/AppSwitcher/AppSwitcher.ino

```
,/* -*- mode: c++ -*-
 * AppSwitcher -- A Kaleidoscope Example
 * Copyright (C) 2016-2018  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
```

(continues on next page)

(continued from previous page)

```
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-EEPROM-Settings.h"
#include "Kaleidoscope-HostOS.h"
#include "Kaleidoscope-Ranges.h"

#include "AppSwitcher.h"

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
   Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,   Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   AppSwitcher_Next,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   AppSwitcher_Prev
   ),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(HostOS,
                          AppSwitcher);

void setup() {
  Kaleidoscope.setup();
  // Uncomment to manually set the OS, as Kaleidoscope will not autodetect it.
  // (Possible values are in HostOS.h.)
  // HostOS.os(kaleidoscope::hostos::LINUX);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.16 Features/CycleTimeReport/CycleTimeReport.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-CycleTimeReport -- Scan cycle time reporting
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-CycleTimeReport.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,   Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_skip,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_skip),
)
// clang-format on

// Override CycleTimeReport's reporting function:
void kaleidoscope::plugin::CycleTimeReport::report(uint16_t mean_cycle_time) {
  Serial.print(F("average loop time = "));
  Serial.println(mean_cycle_time, DEC);
}

KALEIDOSCOPE_INIT_PLUGINS(CycleTimeReport);

void setup() {
```

(continues on next page)

(continued from previous page)

```
  Kaleidoscope.serialPort().begin(9600);
  Kaleidoscope.setup();

  // Change the report interval to 2 seconds:
  CycleTimeReport.setReportInterval(2000);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.17 Features/EEPROM/DynamicMacros/DynamicMacros.ino

```
,/* -*- mode: c++ -*-
* DynamicMacros - Dynamic macro support for Kaleidoscope.
* Copyright (C) 2019  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-EEPROM-Keymap.h>
#include <Kaleidoscope-DynamicMacros.h>
#include <Kaleidoscope-FocusSerial.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,         Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_skip,

   DM(0),     Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
```

(continues on next page)

(continued from previous page)

```
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(
  EEPROMSettings,
  EEPROMKeymap,
  DynamicMacros,
  Focus);

void setup() {
  Kaleidoscope.setup();

  EEPROMKeymap.setup(1);
  DynamicMacros.reserve_storage(128);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.18 Features/EEPROM/EEPROM-Keymap-Programmer/EEPROM-Keymap-Programmer.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-EEPROM-Keymap-Programmer -- On-the-fly reprogrammable keymap.
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-EEPROM-Keymap.h>
#include <Kaleidoscope-EEPROM-Keymap-Programmer.h>
#include <Kaleidoscope-Macros.h>
```

(continues on next page)

(continued from previous page)

```
// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (M(0),              Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_NoKey,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_NoKey),
)
// clang-format on

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  if (macro_id == 0 && keyToggledOff(event.state)) {
    EEPROMKeymapProgrammer.activate();
  }

  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          EEPROMKeymapProgrammer,
                          EEPROMKeymap,
                          Macros);

void setup() {
  Kaleidoscope.serialPort().begin(9600);

  Kaleidoscope.setup();

  Layer.getKey = EEPROMKeymap.getKey;

  EEPROMKeymap.max_layers(1);
  EEPROMSettings.seal();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.19 Features/EEPROM/EEPROM-Keymap/EEPROM-Keymap.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-EEPROM-Keymap -- EEPROM-based keymap support.
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Keymap.h>
#include <Kaleidoscope-FocusSerial.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,         Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_skip,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(EEPROMKeymap, Focus);

void setup() {
  Kaleidoscope.setup();

  EEPROMKeymap.setup(1);
}

void loop() {
```

(continues on next page)

(continued from previous page)

```
  Kaleidoscope.loop();
}
```

11.22.20 Features/EEPROM/EEPROM-Settings/EEPROM-Settings.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-EEPROM-Settings -- Basic EEPROM settings plugin for Kaleidoscope.
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,         Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_skip,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings);

void setup() {
  auto &serial_port = Kaleidoscope.serialPort();
```

(continues on next page)

(continued from previous page)

```
  serial_port.begin(9600);

  Kaleidoscope.setup();

  while (!serial_port) {
  }

  serial_port.println(EEPROMSettings.isValid() ? F("valid EEPROM settings") : F("invalid␣
↪EEPROM settings"));
  serial_port.println(EEPROMSettings.crc(), HEX);
  serial_port.println(EEPROMSettings.version());
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.21 Features/FocusSerial/FocusSerial.ino

```
/* -*- mode: c++ -*-
 * Kaleidoscope-FocusSerial -- Bidirectional communication plugin
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-FocusSerial.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,         Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_skip,
```

(continues on next page)

(continued from previous page)

```
   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_skip),
)
// clang-format on

namespace kaleidoscope {
class FocusTestCommand : public Plugin {
 public:
  FocusTestCommand() {}

  EventHandlerResult onFocusEvent(const char *input) {
    const char *cmd = PSTR("test");

    if (::Focus.inputMatchesHelp(input))
      return ::Focus.printHelp(cmd);

    if (::Focus.inputMatchesCommand(input, cmd)) {
      ::Focus.send(F("ok!"));
      return EventHandlerResult::EVENT_CONSUMED;
    }

    return EventHandlerResult::OK;
  }
};

class FocusHelpCommand : public Plugin {
 public:
  FocusHelpCommand() {}

  EventHandlerResult onFocusEvent(const char *input) {
    if (::Focus.inputMatchesHelp(input))
      return ::Focus.printHelp(PSTR("help"));

    return EventHandlerResult::OK;
  }
};

}  // namespace kaleidoscope

kaleidoscope::FocusTestCommand FocusTestCommand;
kaleidoscope::FocusHelpCommand FocusHelpCommand;

KALEIDOSCOPE_INIT_PLUGINS(Focus, FocusTestCommand, FocusHelpCommand);

void setup() {
  Kaleidoscope.setup();
```

(continues on next page)

(continued from previous page)

```
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.22 Features/GhostInTheFirmware/GhostInTheFirmware.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-GhostInTheFirmware -- Let the keyboard write for you!
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-GhostInTheFirmware.h>
#include <Kaleidoscope-LED-Stalker.h>
#include <Kaleidoscope-Macros.h>

// This sketch is set up to demonstrate the GhostInTheFirmware plugin. The left
// palm key will activate the plugin, virtually pressing each key on the bottom
// row in sequence, and lighting up the keys using the Stalker LED effect. It
// will type out the letters from A to N, but the right palm key can be used to
// toggle the custom EventDropper plugin to suppress USB output.

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___,
   Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,

   ___, ___, ___, ___,
   M(0),

   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
```

(continues on next page)

(continued from previous page)

```
     ___, ___, ___, ___, ___, ___,
   Key_H, Key_I, Key_J, Key_K, Key_L, Key_M, Key_N,

   ___, ___, ___, ___,
   M(1)),
        )
// clang-format on

namespace kaleidoscope {
namespace plugin {

class EventDropper : public Plugin {
 public:
  EventHandlerResult onKeyEvent(KeyEvent &event) {
    if (active_)
      return EventHandlerResult::EVENT_CONSUMED;
    return EventHandlerResult::OK;
  }
  void toggle() {
    active_ = !active_;
  }

 private:
  bool active_ = false;
};

}  // namespace plugin
}  // namespace kaleidoscope

kaleidoscope::plugin::EventDropper EventDropper;

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  if (macro_id == 0 && keyToggledOn(event.state))
    GhostInTheFirmware.activate();
  if (macro_id == 1 && keyToggledOn(event.state))
    EventDropper.toggle();

  return MACRO_NONE;
}

static const kaleidoscope::plugin::GhostInTheFirmware::GhostKey ghost_keys[] PROGMEM = {
  {KeyAddr(3, 0), 200, 50},
  {KeyAddr(3, 1), 200, 50},
  {KeyAddr(3, 2), 200, 50},
  {KeyAddr(3, 3), 200, 50},
  {KeyAddr(3, 4), 200, 50},
  {KeyAddr(3, 5), 200, 50},
  {KeyAddr(2, 6), 200, 50},
  {KeyAddr(2, 9), 200, 50},
  {KeyAddr(3, 10), 200, 50},
  {KeyAddr(3, 11), 200, 50},
  {KeyAddr(3, 12), 200, 50},
```

(continues on next page)

(continued from previous page)

```
  {KeyAddr(3, 13), 200, 50},
  {KeyAddr(3, 14), 200, 50},
  {KeyAddr(3, 15), 200, 50},

  {KeyAddr::none(), 0, 0}};

KALEIDOSCOPE_INIT_PLUGINS(GhostInTheFirmware,
                          LEDControl,
                          StalkerEffect,
                          Macros,
                          EventDropper);

void setup() {
  Kaleidoscope.setup();

  StalkerEffect.variant          = STALKER(BlazingTrail);
  GhostInTheFirmware.ghost_keys = ghost_keys;
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.23 Features/HostOS/HostOS.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-HostOS -- Host OS detection and tracking for Kaleidoscope
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-HostOS.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
```

(continues on next page)

(continued from previous page)

```
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_skip,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings, HostOS);

void setup() {
  auto &serial_port = Kaleidoscope.serialPort();

  serial_port.begin(9600);

  Kaleidoscope.setup();

  serial_port.print("Host OS id is: ");
  serial_port.println(HostOS.os(), DEC);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.24 Features/HostPowerManagement/HostPowerManagement.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-HostPowerManagement -- Host power management support plugin.
 * Copyright (C) 2017, 2018, 2020  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
```

(continues on next page)

(continued from previous page)

```
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-HostPowerManagement.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey
  ),
)
// clang-format on

void hostPowerManagementEventHandler(kaleidoscope::plugin::HostPowerManagement::Event␣
↪event) {
  switch (event) {
  case kaleidoscope::plugin::HostPowerManagement::Suspend:
    LEDControl.disable();
    break;
  case kaleidoscope::plugin::HostPowerManagement::Resume:
    LEDControl.enable();
    break;
  case kaleidoscope::plugin::HostPowerManagement::Sleep:
    break;
  }
}

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          HostPowerManagement);

void setup() {
  Kaleidoscope.setup();
}
```

(continues on next page)

(continued from previous page)

```
void loop() {
  Kaleidoscope.loop();
}
```

11.22.25 Features/Layers/Layers.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope - Firmware for computer input devices
 * Copyright (C) 2020  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-MouseKeys.h>

enum {
  PRIMARY,
  NUMPAD,
  FUNCTION,
};  // layers

// clang-format off
KEYMAPS(
  [PRIMARY] = KEYMAP_STACKED
  (___,          Key_1, Key_2, Key_3, Key_4, Key_5, XXX,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(FUNCTION),

   XXX,  Key_6, Key_7, Key_8,     Key_9,         Key_0,         LockLayer(NUMPAD),
   Key_Enter,     Key_Y, Key_U, Key_I,     Key_O,         Key_P,         Key_Equals,
   Key_H, Key_J, Key_K,     Key_L,         Key_Semicolon, Key_Quote,
   Key_RightAlt,  Key_N, Key_M, Key_Comma, Key_Period,    Key_Slash,     Key_Minus,
   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(FUNCTION)),
```

(continues on next page)

(continued from previous page)

```
  [NUMPAD] =  KEYMAP_STACKED
  (___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___,
   ___,

   XXX,  ___, Key_7, Key_8,      Key_9,              Key_KeypadSubtract, ___,
   ___,                    ___, Key_4, Key_5,      Key_6,              Key_KeypadAdd,   ␣
↪  ___,
                           ___, Key_1, Key_2,      Key_3,              Key_Equals,      ␣
↪  ___,
   ___,                    ___, Key_0, Key_Period, Key_KeypadMultiply, Key_KeypadDivide,␣
↪  Key_Enter,
   ___, ___, ___, ___,
   ___),

  [FUNCTION] =  KEYMAP_STACKED
  (ShiftToLayer(NUMPAD),      Key_F1,           Key_F2,      Key_F3,     Key_F4,       ␣
↪Key_F5,           Key_CapsLock,
   Key_Tab,  ___,              Key_mouseUp, ___,        Key_mouseBtnR, Key_mouseWarpEnd,␣
↪Key_mouseWarpNE,
   Key_Home, Key_mouseL,       Key_mouseDn, Key_mouseR, Key_mouseBtnL, Key_mouseWarpNW,
   Key_End,  Key_PrintScreen,  Key_Insert,  ___,        Key_mouseBtnM, Key_mouseWarpSW, ␣
↪Key_mouseWarpSE,
   ___, Key_Delete, ___, ___,
   ___,

   Consumer_ScanPreviousTrack, Key_F6,                 Key_F7,                   Key_F8,␣
↪                  Key_F9,          Key_F10,           Key_F11,
   Consumer_PlaySlashPause,    Consumer_ScanNextTrack, Key_LeftCurlyBracket,     Key_
↪RightCurlyBracket,    Key_LeftBracket, Key_RightBracket, Key_F12,
                               Key_LeftArrow,          Key_DownArrow,            Key_
↪UpArrow,                 Key_RightArrow,  ___,              ___,
   Key_PcApplication,          Consumer_Mute,          Consumer_VolumeDecrement,␣
↪Consumer_VolumeIncrement, ___,             Key_Backslash,    Key_Pipe,
   ___, ___, Key_Enter, ___,
   ___)
)
// clang-format on

namespace kaleidoscope {
class LayerDumper : public Plugin {
 public:
  LayerDumper() {}

  static void dumpLayerState(uint8_t index, uint8_t layer) {
    Serial.print(index);
    Serial.print(" -> ");
    Serial.println(layer);
  }
```

(continues on next page)

(continued from previous page)

```
  EventHandlerResult onLayerChange() {
    Serial.println("Active Layers:");
    Layer.forEachActiveLayer(&dumpLayerState);
    Serial.println();
    return EventHandlerResult::OK;
  }
};

}  // namespace kaleidoscope

kaleidoscope::LayerDumper LayerDumper;

KALEIDOSCOPE_INIT_PLUGINS(Focus, LayerDumper, MouseKeys);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.26 Features/ModLayer/ModLayer.ino

```
,// -*- mode: c++ -*-

#include <Kaleidoscope.h>
#include <Kaleidoscope-Qukeys.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
      Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
      Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
      Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
      Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

      Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
      ML(LeftShift, 1),

      XXX,       Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
      Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
                 Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
      Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

      Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
      ShiftToLayer(1)
   ),
```

(continues on next page)

(continued from previous page)

```
  [1] = KEYMAP_STACKED
  (
      ___,    Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,

      Key_1, Key_2, Key_3, Key_4,
      ___,


      ___,    Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,

      Key_1, Key_2, Key_3, Key_4,
      ___
   ),
)
// clang-format on

// Use Qukeys
KALEIDOSCOPE_INIT_PLUGINS(Qukeys);

void setup() {
  QUKEYS(
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 1), Key_LeftGui),      // A/cmd
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 2), Key_LeftAlt),      // S/alt
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 3), Key_LeftControl),  // D/ctrl
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 4), Key_LeftShift),    // F/shift

    kaleidoscope::plugin::Qukey(0, KeyAddr(1, 1), ML(LeftGui, 1)),      // Q/cmd+1
    kaleidoscope::plugin::Qukey(0, KeyAddr(1, 2), ML(LeftAlt, 1)),      // W/alt+1
    kaleidoscope::plugin::Qukey(0, KeyAddr(1, 3), ML(LeftControl, 1)),  // E/ctrl+1
    kaleidoscope::plugin::Qukey(0, KeyAddr(1, 4), ML(LeftShift, 1))     // R/shift+1
  )

  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.27 Features/MouseKeys/MouseKeys.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope - A Kaleidoscope example
 * Copyright (C) 2016-2022  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#define DEBUG_SERIAL false

#include <Kaleidoscope.h>
#include <Kaleidoscope-MouseKeys.h>

enum {
  PRIMARY,
  MOUSEKEYS,
};

// clang-format off
KEYMAPS(
  [PRIMARY] = KEYMAP_STACKED
  (Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(MOUSEKEYS),

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   LockLayer(MOUSEKEYS)
   ),

  [MOUSEKEYS] =  KEYMAP_STACKED
  (___, ___, ___,           ___,           ___,             ___,             ___,
   ___, ___, ___,           ___,           Key_mouseWarpNW, Key_mouseWarpNE, ___,
   ___, ___, ___,           ___,           Key_mouseWarpSW, Key_mouseWarpSE,
```

(continues on next page)

(continued from previous page)

```
   ___, ___, Key_mouseBtnL, Key_mouseBtnM, Key_mouseBtnR,   ___,            ___,
   ___, ___, ___, ___,
   ___,

   ___, ___, ___,         ___,          ___,         ___, ___,
   ___, ___, ___,         Key_mouseUp, ___,         ___, ___,
        ___, Key_mouseUp, Key_mouseDn, Key_mouseR, ___, ___,
   ___, ___, ___,         ___,          ___,         ___, ___,
   ___, ___, ___, ___,
   ___)
)
// clang-format on


KALEIDOSCOPE_INIT_PLUGINS(MouseKeys);

void setup() {
  Kaleidoscope.setup();
  MouseKeys.setSpeedLimit(100);
  MouseKeys.setWarpGridSize(MOUSE_WARP_GRID_2X2);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.28 Features/ShiftBlocker/ShiftBlocker.ino

```
,/* -*- mode: c++ -*-
 * ShiftBlocker -- A Kaleidoscope Example
 * Copyright (C) 2016-2022  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include "Kaleidoscope.h"
#include "Kaleidoscope-Macros.h"

// clang-format off
KEYMAPS(
```

(continues on next page)

(continued from previous page)

```
  [0] = KEYMAP_STACKED
  (
   Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,   Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   M(0),

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   M(0)
   ),
)
// clang-format on

namespace kaleidoscope {
namespace plugin {

// When activated, this plugin will suppress any `Shift` key (including modifier
// combos with `Shift`) before it's added to the HID report.
class ShiftBlocker : public Plugin {

 public:
  EventHandlerResult onAddToReport(Key key) {
    if (active_ && key.isKeyboardShift())
      return EventHandlerResult::ABORT;
    return EventHandlerResult::OK;
  }

  void enable() {
    active_ = true;
  }
  void disable() {
    active_ = false;
  }

 private:
  bool active_{false};
};

}  // namespace plugin
}  // namespace kaleidoscope

kaleidoscope::plugin::ShiftBlocker ShiftBlocker;

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
```

(continues on next page)

(continued from previous page)

```
  if (keyToggledOn(event.state)) {
    switch (macro_id) {
    case 0:
      // First, enable ShiftBlocker to suppress any held `Shift` key(s).
      ShiftBlocker.enable();
      // Tap `AltGr` + `7` to activate the grave accent dead key.
      Macros.tap(RALT(Key_7));
      // Disable ShiftBlocker so it won't affect the `E` event.
      ShiftBlocker.disable();
      // Change the Macros key into a plain `E` key before its press event is
      // processed.
      event.key = Key_E;
      break;
    }
  }
  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(Macros,
                          ShiftBlocker);

void setup() {
  Kaleidoscope.setup();
  // Uncomment to manually set the OS, as Kaleidoscope will not autodetect it.
  // (Possible values are in HostOS.h.)
  // HostOS.os(kaleidoscope::hostos::LINUX);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.29 Features/Steno/Steno.ino

```
/* -*- mode: c++ -*-
 * Kaleidoscope-Steno -- Steno protocols for Kaleidoscope
 * Copyright (C) 2017  Joseph Wasson
 * Copyright (C) 2017, 2018  Gergely Nagy
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
```

(continues on next page)

(continued from previous page)

```
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Steno.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,         Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_Keymap1,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_Keymap1),

  [1] = KEYMAP_STACKED
  (XXX,    XXX,   XXX,   XXX,   XXX,   XXX,   S(N6),
   XXX,    S(N1), S(N2), S(N3), S(N4), S(N5), S(ST1),
   S(FN),  S(S1), S(TL), S(PL), S(HL), S(ST1),
   S(PWR), S(S2), S(KL), S(WL), S(RL), S(ST2), S(ST2),

   S(RE1), XXX, S(A), S(O),
   ___,

   S(N7),  XXX,    XXX,   XXX,   XXX,   XXX,   XXX,
   S(ST3), S(N8),  S(N9), S(NA), S(NB), S(NC), XXX,
           S(ST3), S(FR), S(PR), S(LR), S(TR), S(DR),
   S(ST4), S(ST4), S(RR), S(BR), S(GR), S(SR), S(ZR),

   S(E), S(U), XXX, S(RE2),
   ___),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(GeminiPR);

void setup() {
  Kaleidoscope.serialPort().begin(9600);
  Kaleidoscope.setup();
}

void loop() {
```

(continues on next page)

(continued from previous page)

```
  Kaleidoscope.loop();
}
```

11.22.30 Features/TypingBreaks/TypingBreaks.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-TypingBreaks -- Enforced typing breaks
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-TypingBreaks.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,   Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_skip,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings, TypingBreaks);
```

(continues on next page)

(continued from previous page)

```
void setup() {
  Kaleidoscope.setup();

  TypingBreaks.settings.idle_time_limit = 60;
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.31 Internal/Sketch_Exploration/Sketch_Exploration.ino

```
,/* -*- mode: c++ -*-
 * Basic -- A very basic Kaleidoscope example
 * Copyright (C) 2018  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include "Kaleidoscope.h"


// This example demonstrates how a plugin can gather information about
// the keymap at compile time, e.g. to adapt its behavior, safe resources, ...

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
   Key_NoKey,    Key_1, Key_1, Key_1, Key_4, Key_5, Key_NoKey,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_NoKey,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
```

(continues on next page)

(continued from previous page)

```
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_NoKey
  ),
)
// clang-format on

using namespace kaleidoscope::sketch_exploration;  // NOLINT(build/namespaces)

class BPlugin : public kaleidoscope::Plugin {};
class CPlugin : public kaleidoscope::Plugin {};

// A simple plugin that defines just one hook.
//
class APlugin : public kaleidoscope::Plugin {

 public:
  APlugin()
    : has_key_1_{false} {}

  template<typename _Sketch>
  kaleidoscope::EventHandlerResult exploreSketch() {

    // Static keymap exploration

    typedef typename _Sketch::StaticKeymap K;

    // Important: Always make sure to call _Sketch::StaticKeymap's methods
    //            in a constexpr context. This is done by
    //            passing their value to a constexpr temporary variable.

    constexpr uint8_t n_key_1 = K::collect(NumKeysEqual{Key_1});
    static_assert(n_key_1 == 3, "Error determining key count");

    constexpr bool has_key_1 = K::collect(HasKey{Key_1});
    static_assert(has_key_1, "Error querying key existence");
    has_key_1_ = has_key_1;  // Assign the temporary that was computed
    // at compile time.

    constexpr Key max_key = K::collect(MaxKeyRaw{});
    static_assert(max_key.getRaw() > 0, "");

    static_assert(K::getKey(0 /*layer*/, KeyAddr{2, 3}) == Key_D,
                  "Key lookup failed");

    constexpr auto n_layers   = K::nLayers();
    constexpr auto layer_size = K::layerSize();

    // Plugin exploration
    //
    // Use macros ENTRY_TYPE, ENRTY_IS_LAST, PLUGIN_POSITION,
```

(continues on next page)

(continued from previous page)

```
    // PLUGIN_IS_REGISTERED and NUM_OCCURRENCES to retreive information
    // about the plugins that are registered in the sketch.

    typedef typename _Sketch::Plugins P;

    static_assert(std::is_same<ENTRY_TYPE(P, 0), APlugin>::value, "");
    static_assert(std::is_same<ENTRY_TYPE(P, 1), BPlugin>::value, "");

    static_assert(P::size == 3, "");

    static_assert(!ENRTY_IS_LAST(P, 0), "");
    static_assert(!ENRTY_IS_LAST(P, 1), "");
    static_assert(ENRTY_IS_LAST(P, 2), "");

    static_assert(PLUGIN_POSITION(P, APlugin) == 0, "");
    static_assert(PLUGIN_POSITION(P, BPlugin) == 1, "");
    static_assert(PLUGIN_POSITION(P, CPlugin) == -1, "");

    static_assert(PLUGIN_IS_REGISTERED(P, APlugin) == true, "");
    static_assert(PLUGIN_IS_REGISTERED(P, BPlugin) == true, "");
    static_assert(PLUGIN_IS_REGISTERED(P, CPlugin) == false, "");

    static_assert(NUM_OCCURRENCES(P, APlugin) == 2, "");
    static_assert(NUM_OCCURRENCES(P, BPlugin) == 1, "");
    static_assert(NUM_OCCURRENCES(P, CPlugin) == 0, "");

    return kaleidoscope::EventHandlerResult::OK;
  }

 private:
  bool has_key_1_;
};

APlugin a_plugin1, a_plugin2;
BPlugin b_plugin;

KALEIDOSCOPE_INIT_PLUGINS(
  a_plugin1,
  b_plugin,
  a_plugin2)

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.32 Keystrokes/AutoShift/AutoShift.ino

```
,// -*- mode: c++ -*-

#include <Kaleidoscope.h>

#include <Kaleidoscope-AutoShift.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-EEPROM-Keymap.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-Macros.h>

enum {
  TOGGLE_AUTOSHIFT,
};

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
      Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
      Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
      Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
      Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

      Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
      XXX,

      M(TOGGLE_AUTOSHIFT), Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
      Key_Enter,           Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_
↪Equals,
                           Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
      Key_skip,            Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

      Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
      XXX
   ),
)
// clang-format on

// Defining a macro (on the "any" key: see above) to turn AutoShift on and off
const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {
  case TOGGLE_AUTOSHIFT:
    if (keyToggledOn(event.state))
      AutoShift.toggle();
    break;
  }
  return MACRO_NONE;
}

// This sketch uses the AutoShiftConfig plugin, which enables run-time
// configuration of AutoShift configuration settings.  All of the plugins marked
```

(continues on next page)

(continued from previous page)

```
// "for AutoShiftConfig" are optional; AutoShift itself will work without them.
KALEIDOSCOPE_INIT_PLUGINS(
  EEPROMSettings,        // for AutoShiftConfig
  EEPROMKeymap,          // for AutoShiftConfig
  Focus,                 // for AutoShiftConfig
  FocusEEPROMCommand,    // for AutoShiftConfig
  FocusSettingsCommand,  // for AutoShiftConfig
  AutoShift,
  AutoShiftConfig,  // for AutoShiftConfig
  Macros            // for toggle AutoShift Macro
);

void setup() {
  // Enable AutoShift for letter keys and number keys only:
  AutoShift.setEnabled(AutoShift.letterKeys() | AutoShift.numberKeys());
  // Add symbol keys to the enabled categories:
  AutoShift.enable(AutoShift.symbolKeys());
  // Set the AutoShift long-press time to 150ms:
  AutoShift.setTimeout(150);
  // Start with AutoShift turned off:
  AutoShift.disable();

  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.33 Keystrokes/CharShift/CharShift.ino

```
,// -*- mode: c++ -*-

#include <Kaleidoscope.h>

#include <Kaleidoscope-CharShift.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
      XXX,          Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
      Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
      Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
      Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

      Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
      XXX,

      XXX,       Key_6, Key_7, Key_8, Key_9, Key_0,     Key_skip,
```

(continues on next page)

(continued from previous page)

```
      Key_Enter, Key_Y, Key_U, Key_I, Key_O, Key_P,     Key_Equals,
                 Key_H, Key_J, Key_K, Key_L, CS(2),     Key_Quote,
      Key_skip,  Key_N, Key_M, CS(0), CS(1), Key_Slash, Key_Minus,

      Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
      XXX
   ),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(CharShift);

void setup() {
  CS_KEYS(
    kaleidoscope::plugin::CharShift::KeyPair(Key_Comma, Key_Semicolon),               //␣
↪CS(0)
    kaleidoscope::plugin::CharShift::KeyPair(Key_Period, LSHIFT(Key_Semicolon)),      //␣
↪CS(1)
    kaleidoscope::plugin::CharShift::KeyPair(LSHIFT(Key_Comma), LSHIFT(Key_Period)),  //␣
↪CS(2)
  );
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.34 Keystrokes/Chord/Chord.ino

```
,// -*- mode: c++ -*-

#include <Kaleidoscope.h>
#include "Kaleidoscope-TopsyTurvy.h"
#include <Kaleidoscope-Chord.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
      Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
      Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
      Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
      Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

      Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
      Key_NoKey,

      Key_NoKey, Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
      Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
```

(continues on next page)

(continued from previous page)

```
                 Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
      Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

      Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
      Key_NoKey
   ),
)

KALEIDOSCOPE_INIT_PLUGINS(TopsyTurvy, Chord);

void setup() {
  CHORDS(
    CHORD(Key_J, Key_K), Key_Escape,
    CHORD(Key_D, Key_F), Key_LeftShift,
    CHORD(Key_S, Key_D), TOPSY(Semicolon),
    CHORD(Key_S, Key_D, Key_F), Key_Spacebar,
  )
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.35 Keystrokes/Cycle/Cycle.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Cycle -- Key sequence cycling dead key for Kaleidoscope.
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Cycle.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
```

(continues on next page)

(continued from previous page)

```
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_Cycle,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_Cycle),
)
// clang-format on

void cycleAction(Key previous_key, uint8_t cycle_count) {
  if (previous_key == Key_E) {
    if (cycle_count == 1) {
      Cycle.replace(Key_F);
    } else if (cycle_count == 2) {
      Cycle.replace(Key_G);
    }
  }

  bool is_shifted = previous_key.getFlags() & SHIFT_HELD;
  if (previous_key.getKeyCode() == Key_A.getKeyCode() && is_shifted)
    cycleThrough(LSHIFT(Key_A), LSHIFT(Key_B), LSHIFT(Key_C), LSHIFT(Key_D));
  if (previous_key.getKeyCode() == Key_A.getKeyCode() && !is_shifted)
    cycleThrough(Key_A, Key_B, Key_C, Key_D);
}

KALEIDOSCOPE_INIT_PLUGINS(Cycle);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.36 Keystrokes/DynamicTapDance/DynamicTapDance.ino

```
,/* -*- mode: c++ -*-
 * DynamicTapDance -- Dynamic TapDance support for Kaleidoscope
 * Copyright (C) 2019  Keyboard.io, Inc
 * Copyright (C) 2019  Dygma Lab S.L.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-TapDance.h>
#include <Kaleidoscope-DynamicTapDance.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    TD(0),

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, TD(2),
    TD(1)),
)
// clang-format on

enum {
  TD_TAB_ESC,
  TD_LAST
};
```

(continues on next page)

(continued from previous page)

```
void tapDanceAction(uint8_t tap_dance_index, KeyAddr key_addr, uint8_t tap_count,␣
↪kaleidoscope::plugin::TapDance::ActionType tap_dance_action) {
  switch (tap_dance_index) {
  case TD_TAB_ESC:
    return tapDanceActionKeys(tap_count, tap_dance_action, Key_A, Key_B);
  default:
    DynamicTapDance.dance(tap_dance_index, key_addr, tap_count, tap_dance_action);
  }
}

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          Focus,
                          TapDance,
                          DynamicTapDance);

void setup() {
  Kaleidoscope.setup();
  DynamicTapDance.setup(TD_LAST, 32);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.37 Keystrokes/Escape-OneShot/Escape-OneShot.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Escape-OneShot -- Turn ESC into a key that cancels OneShots, if active.
 * Copyright (C) 2016-2021  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-OneShot.h>
#include <Kaleidoscope-Escape-OneShot.h>
#include <Kaleidoscope-FocusSerial.h>

// clang-format off
```

(continues on next page)

(continued from previous page)

```
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    OSM(LeftControl), Key_Backspace, OSM(LeftGui), Key_LeftShift,
    Key_Keymap1_Momentary,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, OSM(RightAlt), Key_Spacebar, OSM(RightControl),
    OSL(1)
  ),

  [1] = KEYMAP_STACKED
  (
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,

    ___, ___, ___, ___,
    ___,

    ___, ___,         ___,                ___,                ___,                ___, _
↪__,
    ___, ___,         ___,                ___,                ___,                ___, _
↪__,
         Key_UpArrow, Key_DownArrow,         Key_LeftArrow,         Key_RightArrow, ___,␣
↪___,
    ___, ___,         ___,                ___,                ___,                ___, _
↪__,

    ___, ___, ___, ___,
    ___
  ),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          Focus,
                          OneShot,
                          EscapeOneShot,
                          EscapeOneShotConfig);

void setup() {
```

(continues on next page)

(continued from previous page)

```
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.38 Keystrokes/Leader/Leader.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Leader -- VIM-style leader keys
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Leader.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey, Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   LEAD(0),

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   LEAD(0)),
)
// clang-format on
```

(continues on next page)

(continued from previous page)

```
auto &serial_port = Kaleidoscope.serialPort();

static void leaderTestA(uint8_t seq_index) {
  serial_port.println(F("leaderTestA"));
}

static void leaderTestAA(uint8_t seq_index) {
  serial_port.println(F("leaderTestAA"));
}

static const kaleidoscope::plugin::Leader::dictionary_t leader_dictionary[] PROGMEM =
  LEADER_DICT({LEADER_SEQ(LEAD(0), Key_A), leaderTestA},
              {LEADER_SEQ(LEAD(0), Key_A, Key_A), leaderTestAA});

KALEIDOSCOPE_INIT_PLUGINS(Leader);

void setup() {
  Kaleidoscope.setup();

  Leader.dictionary = leader_dictionary;
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.39 Keystrokes/LeaderPrefix/LeaderPrefix.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LeaderPrefix -- Prefix arg for Leader plugin
 * Copyright (C) 2021  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Leader.h>
#include <Kaleidoscope-MacroSupport.h>

#include <Kaleidoscope-Ranges.h>
```

(continues on next page)

(continued from previous page)

```
#include "kaleidoscope/KeyEventTracker.h"
#include "kaleidoscope/LiveKeys.h"
#include "kaleidoscope/plugin.h"

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   LEAD(0),

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   LEAD(0)),
)
// clang-format on

namespace kaleidoscope {
namespace plugin {

// =============================================================================
/// Plugin to supply a numeric prefix argument to Leader key functions
///
/// This plugin lets the user type a numeric prefix after a Leader key is
/// pressed, but before the rest of the Leader sequence is begun, storing the
/// "prefix argument" and making it available to functions called from the
/// leader dictionary.  LeaderPrefix allows us to define keys other than the
/// ones on the number row to be interpreted as the "digit" keys, because
/// whatever we use will need to be accessed without a layer change.
class LeaderPrefix : public Plugin {
 public:
  // We need to define `onKeyswitchEvent()` instead of `onKeyEvent()` because we
  // need to intercept events before Leader sees them, and the Leader plugin
  // uses the former.
  EventHandlerResult onKeyswitchEvent(KeyEvent &event) {
    // Every `onKeyswitchEvent()` function should begin with this to prevent
    // re-processing events that it has already seen.
    if (event_tracker_.shouldIgnore(event))
      return EventHandlerResult::OK;

    // `Active` means that we're actively building the prefix argument.  If the
    // plugin is not active, we're looking for a Leader key toggling on.
    if (!active_) {
      if (keyToggledOn(event.state) && isLeaderKey(event.key)) {
```

(continues on next page)

(continued from previous page)

```
        // A Leader key toggled on, so we set our state to "active", and set the
        // arg value to zero.
        active_     = true;
        leader_arg_ = 0;
      }
      // Whether or not the plugin just became active, there's nothing more to
      // do for this event.
      return EventHandlerResult::OK;
    }

    // The plugin is "active", so we're looking for a "digit" key that just
    // toggled on.
    if (keyToggledOn(event.state)) {
      // We search our array of digit keys to find one that matches the event.
      // These "digit keys" are defined by their `KeyAddr` because they're
      // probably independent of keymap and layer, and because a `KeyAddr` only
      // takes one byte, whereas a `Key` takes two.
      for (uint8_t i{0}; i < 10; ++i) {
        if (digit_addrs_[i] == event.addr) {
          // We found a match, which means that one of our "digit keys" toggled
          // on.  If this happens more than once, the user is typing a number
          // with multiple digits, so we multiply the current value by ten
          // before adding the new digit to the total.
          leader_arg_ *= 10;
          leader_arg_ += i;
          // Next, we mask the key that was just pressed, so that nothing will
          // happen when it is released.
          live_keys.mask(event.addr);
          // We return `ABORT` so that no other plugins (i.e. Leader) will see
          // this keypress event.
          return EventHandlerResult::ABORT;
        }
      }
    }
    // No match was found, so the key that toggled on was not one of our "digit
    // keys".  Presumably, this is the first key in the Leader sequence that is
    // being typed.  We leave the prefix argument at its current value so that
    // it will still be set when the sequence is finished, and allow the event
    // to pass through to the next plugin (i.e. Leader).
    active_ = false;
    return EventHandlerResult::OK;
  }

  uint16_t arg() const {
    return leader_arg_;
  }

 private:
  // The "digit keys": these are the keys on the number row of the Model01.
  KeyAddr digit_addrs_[10] = {
    KeyAddr(0, 14),
    KeyAddr(0, 1),
```

(continues on next page)

(continued from previous page)

```
    KeyAddr(0, 2),
    KeyAddr(0, 3),
    KeyAddr(0, 4),
    KeyAddr(0, 5),
    KeyAddr(0, 10),
    KeyAddr(0, 11),
    KeyAddr(0, 12),
    KeyAddr(0, 13),
  };

  // This event tracker is necessary to prevent re-processing events.  Any
  // plugin that defines `onKeyswitchEvent()` should use one.
  KeyEventTracker event_tracker_;

  // The current state of the plugin.  It determines whether we're looking for a
  // Leader keypress or building a prefix argument.
  bool active_{false};

  // The prefix argument itself.
  uint16_t leader_arg_{0};

  // Leader should probably provide this test, but since it doesn't, we add it
  // here to determine if a key is a Leader key.
  bool isLeaderKey(Key key) {
    return (key >= ranges::LEAD_FIRST && key <= ranges::LEAD_LAST);
  }
};

}  // namespace plugin
}  // namespace kaleidoscope

// This creates our plugin object.
kaleidoscope::plugin::LeaderPrefix LeaderPrefix;

auto &serial_port = Kaleidoscope.serialPort();

static void leaderTestX(uint8_t seq_index) {
  serial_port.println(F("leaderTestX"));
}

static void leaderTestXX(uint8_t seq_index) {
  serial_port.println(F("leaderTestXX"));
}

// This demonstrates how to use the prefix argument in a Leader function.  In
// this case, our function just types as many `x` characters as specified by the
// prefix arg.
void leaderTestPrefix(uint8_t seq_index) {
  // Read the prefix argument into a temporary variable:
  uint8_t prefix_arg = LeaderPrefix.arg();
  // Use a Macros helper function to tap the `X` key repeatedly.
  while (prefix_arg-- > 0)
```

(continues on next page)

(continued from previous page)

```
    MacroSupport.tap(Key_X);
}

static const kaleidoscope::plugin::Leader::dictionary_t leader_dictionary[] PROGMEM =
  LEADER_DICT({LEADER_SEQ(LEAD(0), Key_X), leaderTestX},
              {LEADER_SEQ(LEAD(0), Key_X, Key_X), leaderTestXX},
              {LEADER_SEQ(LEAD(0), Key_Z), leaderTestPrefix});

// The order matters here; LeaderPrefix won't work unless it precedes Leader in
// this list.  If there are other plugins in the list, these two should ideally
// be next to each other, but that's not necessary.
KALEIDOSCOPE_INIT_PLUGINS(LeaderPrefix, Leader);

void setup() {
  Kaleidoscope.setup();

  Leader.dictionary = leader_dictionary;
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.40 Keystrokes/Macros/Macros.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Macros Examples
 * Copyright (C) 2021  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Macros.h>
#include <Kaleidoscope-OneShot.h>

// Macros
enum {
  TOGGLE_ONESHOT,
};
```

(continues on next page)

(continued from previous page)

```
// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (___, M(1), M(2), M(3), M(4), M(5), ___,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   ShiftToLayer(1),

   ___, M(6), M(7), M(8), M(9), M(0), ___,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   ___,       Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_LeftAlt, Key_Spacebar, Key_RightControl,
   ShiftToLayer(1)),

  [1] = KEYMAP_STACKED
  (
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,

    ___, ___, ___, ___,
    ___,

    ___, ___,         ___,          ___,         ___,                ___, ___,
    ___, ___,         ___,          ___,         ___,                ___, ___,
         Key_UpArrow, Key_DownArrow, Key_LeftArrow, Key_RightArrow,___, ___,
    ___, ___,         ___,          ___,         ___,                ___, ___,

    ___, ___, ___, ___,
    ___),
)
// clang-format on

// Example macro for typing a string of characters.
void macroTypeString(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    Macros.type(PSTR("Hello, world!"));
  }
}

// Example macro for macro step sequence.
const macro_t *macroSteps(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    // Note that the following sequence leaves two keys down (`Key_RightAlt` and
    // `Key_C`). These virtual keys will remain in effect until the Macros key
```

(continues on next page)

(continued from previous page)

```
    // is released.
    return MACRO(I(200), D(LeftShift), T(A), D(RightAlt), T(B), U(LeftShift), D(C));
  }
  return MACRO_NONE;
}

// Example macro that sets `event.key`.
const macro_t *macroNewSentence1(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    event.key = OSM(LeftShift);
    return MACRO(Tc(Period), Tc(Spacebar), Tc(Spacebar));
  }
  return MACRO_NONE;
}

// Alternate example for above.
void macroNewSentence2(KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    Macros.type(PSTR(".  "));
    event.key = OSM(LeftShift);
  }
}

// Macro that calls `handleKeyEvent()`. This version works even if the OneShot
// plugin is registered before Macros in `KALEIDOSCOPE_INIT_PLUGINS()`.
void macroNewSentence3(KeyEvent &event) {
  Macros.tap(Key_Period);
  Macros.tap(Key_Spacebar);
  Macros.tap(Key_Spacebar);
  // Change the event into a OneShot key event.
  event.key = OSM(LeftShift);
  kaleidoscope::Runtime.handleKeyEvent(event);
  // We can effectively erase the Macros key event, effectively aborting it.
  event.key = Key_NoKey;
  event.addr.clear();
}

// Macro that auto-repeats?

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {

  case 0:
    macroTypeString(event);
    break;

  case 1:
    return macroNewSentence1(event);

  case 2:
    macroNewSentence2(event);
    break;
```

(continues on next page)

(continued from previous page)

```
  case 3:
    macroNewSentence3(event);
    break;

  case 4:
    return macroSteps(event);

  default:
    break;
  }
  return MACRO_NONE;
}

// For some of the above examples, it's important that Macros is registered
// before OneShot here.
KALEIDOSCOPE_INIT_PLUGINS(Macros, OneShot);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.41 Keystrokes/MagicCombo/MagicCombo.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-MagicCombo -- Magic combo framework
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Macros.h>
#include <Kaleidoscope-MagicCombo.h>

enum {
```

(continues on next page)

(continued from previous page)

```
  KIND_OF_MAGIC
};

void kindOfMagic(uint8_t combo_index) {
  Macros.type(PSTR("It's a kind of magic!"));
}

USE_MAGIC_COMBOS([KIND_OF_MAGIC] = {.action = kindOfMagic, .keys = {R3C6, R3C9}});

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey, Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(MagicCombo, Macros);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.42 Keystrokes/OneShot/OneShot.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-OneShot -- One-shot modifiers and layers
 * Copyright (C) 2016-2018  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
```

(continues on next page)

(continued from previous page)

```
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Macros.h>
#include <Kaleidoscope-OneShot.h>

// Macros
enum {
  TOGGLE_ONESHOT,
};

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    M(TOGGLE_ONESHOT), Key_1, Key_2, Key_3, Key_4, Key_5, ___,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    OSM(LeftControl), Key_Backspace, OSM(LeftGui), OSM(LeftShift),
    Key_Meh,

    ___, Key_6, Key_7, Key_8, Key_9, Key_0,    ___,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    ___,       Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    OSL(1)),

  [1] = KEYMAP_STACKED
  (
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,

    ___, ___, ___, ___,
    ___,

    ___, ___,          ___,          ___,         ___,                ___, ___,
    ___, ___,          ___,          ___,         ___,                ___, ___,
         Key_UpArrow, Key_DownArrow, Key_LeftArrow, Key_RightArrow,___, ___,
```

(continues on next page)

(continued from previous page)

```
    ___, ___,           ___,           ___,          ___,                  ___, ___,

    ___, ___, ___, ___,
    ___),
)
// clang-format on

void macroToggleOneShot() {
  OneShot.toggleAutoOneShot();
}

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  if (macro_id == TOGGLE_ONESHOT) {
    macroToggleOneShot();
  }

  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(OneShot, Macros);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.43 Keystrokes/OneShotMetaKeys/OneShotMetaKeys.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-OneShotMetaKeys -- Special OneShot keys
 * Copyright (C) 2021  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Macros.h>
```

(continues on next page)

(continued from previous page)

```
#include <Kaleidoscope-OneShot.h>
#include <Kaleidoscope-OneShotMetaKeys.h>

// Macros
enum {
  TOGGLE_ONESHOT,
};

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    M(TOGGLE_ONESHOT), Key_1, Key_2, Key_3, Key_4, Key_5, OneShot_MetaStickyKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    OSM(LeftControl), Key_Backspace, OSM(LeftGui), OSM(LeftShift),
    Key_Meh,

    OneShot_ActiveStickyKey, Key_6, Key_7, Key_8, Key_9, Key_0,    ___,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    ___,       Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    OSL(1)),

  [1] = KEYMAP_STACKED
  (
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___,
    ___, ___, ___, ___, ___, ___, ___,

    ___, ___, ___, ___,
    ___,

    ___, ___,          ___,          ___,         ___,               ___, ___,
    ___, ___,          ___,          ___,         ___,               ___, ___,
         Key_UpArrow, Key_DownArrow, Key_LeftArrow, Key_RightArrow,___, ___,
    ___, ___,          ___,          ___,         ___,               ___, ___,

    ___, ___, ___, ___,
    ___),
)
// clang-format on

void macroToggleOneShot() {
  OneShot.toggleAutoOneShot();
}
```

(continues on next page)

(continued from previous page)

```
const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  if (macro_id == TOGGLE_ONESHOT) {
    macroToggleOneShot();
  }

  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(OneShot, OneShotMetaKeys, Macros);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.44 Keystrokes/PrefixLayer/PrefixLayer.ino

```
,// -*- mode: c++ -*-

/* This example demonstrates the Model 01 / Model 100 butterfly logo key as a
 * tmux prefix key. When the key is held, Ctrl-B is pressed prior to the key
 * you pressed.
 *
 * This example also demonstrates the purpose of using an entire layer for this
 * plugin: the h/j/k/l keys in the TMUX layer are swapped for arrow keys to
 * make switching between panes easier.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-PrefixLayer.h>

enum {
  PRIMARY,
  TMUX,
};  // layers

/* Used in setup() below. */
static const kaleidoscope::plugin::PrefixLayer::Entry prefix_layers[] PROGMEM = {
  kaleidoscope::plugin::PrefixLayer::Entry(TMUX, LCTRL(Key_B)),
};

// clang-format off
KEYMAPS(
  [PRIMARY] = KEYMAP_STACKED
  (XXX,          Key_1, Key_2, Key_3, Key_4, Key_5, XXX,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
```

(continues on next page)

(continued from previous page)

```
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,
   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   XXX,

   XXX,                Key_6, Key_7, Key_8,     Key_9,      Key_0,         XXX,
   Key_Enter,          Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
                       Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   ShiftToLayer(TMUX), Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,
   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   XXX),

  [TMUX] = KEYMAP_STACKED
  (___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___, ___, ___, ___,
   ___, ___, ___, ___,
   ___,

   ___, ___,           ___,           ___,         ___,            ___, ___,
   ___, ___,           ___,           ___,         ___,            ___, ___,
        Key_LeftArrow, Key_DownArrow, Key_UpArrow, Key_RightArrow, ___, ___,
   ___, ___,           ___,           ___,         ___,            ___, ___,
   ___, ___, ___, ___,
   ___),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(PrefixLayer);

void setup() {
  Kaleidoscope.setup();
  /* Configure the previously-defined prefix layers. */
  PrefixLayer.setPrefixLayers(prefix_layers);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.45 Keystrokes/Qukeys/Qukeys.ino

```
,// -*- mode: c++ -*-

#include <Kaleidoscope.h>
#include <Kaleidoscope-Qukeys.h>
#include <Kaleidoscope-Macros.h>

enum { MACRO_TOGGLE_QUKEYS };
```

(continues on next page)

(continued from previous page)

```
// To define DualUse Qukeys in the keymap itself, we can use `SFT_T(key)`,
// `CTL_T(key)`, `ALT_T(key)`, `GUI_T(key)`, and `LT(layer, key)`, as defined
// for the home row on the right side of the keymap (and one of the palm keys)
// below.

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
      Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
      Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
      Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
      Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

      Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
      Key_Q,

      M(MACRO_TOGGLE_QUKEYS), Key_6, Key_7,    Key_8,     Key_9,      Key_0,         ␣
↪Key_skip,
      Key_Enter,              Key_Y, Key_U,    Key_I,     Key_O,      Key_P,         ␣
↪Key_Equals,
                              Key_H, SFT_T(J), CTL_T(K),  ALT_T(L),   GUI_T(Semicolon),␣
↪Key_Quote,
      Key_skip,               Key_N, Key_M,    Key_Comma, Key_Period, Key_Slash,     ␣
↪Key_Minus,

      Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
      LT(1,E)
   ),
  [1] = KEYMAP_STACKED
  (
      ___,   Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,

      Key_1, Key_2, Key_3, Key_4,
      ___,


      ___,   Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F,
      Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G,

      Key_1, Key_2, Key_3, Key_4,
      ___
   ),
)
// clang-format on

// Defining a macro (on the "any" key: see above) to toggle Qukeys on and off
```

(continues on next page)

(continued from previous page)

```
const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {
  case MACRO_TOGGLE_QUKEYS:
    if (keyToggledOn(event.state))
      Qukeys.toggle();
    break;
  }
  return MACRO_NONE;
}

// Use Qukeys
KALEIDOSCOPE_INIT_PLUGINS(Qukeys, Macros);

void setup() {
  // The following Qukey definitions are for the left side of the home row (and
  // the left palm key) of the Keyboardio Model01 keyboard.  For other
  // keyboards, the `KeyAddr(row, col)` coordinates will need adjustment.
  QUKEYS(
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 1), Key_LeftGui),      // A/cmd
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 2), Key_LeftAlt),      // S/alt
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 3), Key_LeftControl),  // D/ctrl
    kaleidoscope::plugin::Qukey(0, KeyAddr(2, 4), Key_LeftShift),    // F/shift
    kaleidoscope::plugin::Qukey(0, KeyAddr(3, 6), ShiftToLayer(1))   // Q/layer-shift␣
↪(on `fn`)
  )
  Qukeys.setHoldTimeout(1000);
  Qukeys.setOverlapThreshold(50);
  Qukeys.setMinimumHoldTime(100);
  Qukeys.setMinimumPriorInterval(80);
  Qukeys.setMaxIntervalForTapRepeat(150);

  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.46 Keystrokes/Redial/Redial.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Redial -- Redial support for Kaleidoscope
 * Copyright (C) 2018, 2019  Keyboard.io, Inc.
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
```

(continues on next page)

(continued from previous page)

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-Redial.h>

bool kaleidoscope::plugin::Redial::shouldRemember(Key mapped_key) {
  if (mapped_key >= Key_A && mapped_key <= Key_Z)
    return true;
  return false;
}

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_Redial,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_Redial),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(Redial);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.47 Keystrokes/ShapeShifter/ShapeShifter.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-ShapeShifter -- Change the shifted symbols on any key of your choice
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-ShapeShifter.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_skip,          Key_1, Key_2, Key_3, Key_4, Key_5, Key_skip,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

static const kaleidoscope::plugin::ShapeShifter::dictionary_t shape_shift_dictionary[]␣
↪PROGMEM = {
  {Key_1, Key_2},
  {Key_2, Key_1},
  {Key_NoKey, Key_NoKey},
};

KALEIDOSCOPE_INIT_PLUGINS(ShapeShifter);
```

(continues on next page)

(continued from previous page)

```
void setup() {
  Kaleidoscope.setup();

  ShapeShifter.dictionary = shape_shift_dictionary;
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.48 Keystrokes/SpaceCadet/SpaceCadet.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-SpaceCadet -- Space Cadet Shift
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-SpaceCadet.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_SpaceCadetEnable,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_skip,

    Key_SpaceCadetDisable, Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter,             Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_
↪Equals,
                           Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,              Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,
```

(continues on next page)

(continued from previous page)

```
    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(SpaceCadet);

void setup() {
  Kaleidoscope.setup();

  //Set the SpaceCadet map
  //Setting is {KeyThatWasPressed, AlternativeKeyToSend, TimeoutInMS}
  //Note: must end with the SPACECADET_MAP_END delimiter
  static kaleidoscope::plugin::SpaceCadet::KeyBinding spacecadetmap[] = {
    {Key_LeftShift, Key_LeftParen, 250},
    {Key_RightShift, Key_RightParen, 250},
    {Key_LeftGui, Key_LeftCurlyBracket, 250},
    {Key_RightAlt, Key_RightCurlyBracket, 250},
    {Key_LeftAlt, Key_RightCurlyBracket, 250},
    {Key_LeftControl, Key_LeftBracket, 250},
    {Key_RightControl, Key_RightBracket, 250},
    SPACECADET_MAP_END,
  };
  //Set the map.
  SpaceCadet.setMap(spacecadetmap);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.49 Keystrokes/Syster/Syster.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Syster -- Symbolic input system
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */
```

(continues on next page)

(continued from previous page)

```
#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-HostOS.h>
#include <Kaleidoscope-Syster.h>
#include <Kaleidoscope-Unicode.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    SYSTER,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    SYSTER),
)
// clang-format on

void systerAction(kaleidoscope::plugin::Syster::action_t action, const char *symbol) {
  switch (action) {
  case kaleidoscope::plugin::Syster::StartAction:
    Unicode.type(0x2328);
    break;
  case kaleidoscope::plugin::Syster::EndAction:
    kaleidoscope::eraseChars(1);
    break;
  case kaleidoscope::plugin::Syster::SymbolAction:
    Kaleidoscope.serialPort().print("systerAction: symbol=");
    Kaleidoscope.serialPort().println(symbol);
    if (strcmp(symbol, "coffee") == 0) {
      Unicode.type(0x2615);
    }
    break;
  }
}

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          HostOS,
                          Unicode,
                          Syster);

void setup() {
```

(continues on next page)

(continued from previous page)

```
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.50 Keystrokes/TapDance/TapDance.ino

```
/* -*- mode: c++ -*-
 * Kaleidoscope-TapDance -- Tap-dance keys
 * Copyright (C) 2016, 2017, 2018, 2019  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-TapDance.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    TD(0),

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    TD(1)),
)
// clang-format on
```

(continues on next page)

(continued from previous page)

```
static void tapDanceEsc(uint8_t tap_dance_index, uint8_t tap_count,␣
↪kaleidoscope::plugin::TapDance::ActionType tap_dance_action) {
  tapDanceActionKeys(tap_count, tap_dance_action, Key_Escape, Key_Tab);
}

void tapDanceAction(uint8_t tap_dance_index, KeyAddr key_addr, uint8_t tap_count,␣
↪kaleidoscope::plugin::TapDance::ActionType tap_dance_action) {
  switch (tap_dance_index) {
  case 0:
    return tapDanceActionKeys(tap_count, tap_dance_action, Key_Tab, Key_Escape);
  case 1:
    return tapDanceEsc(tap_dance_index, tap_count, tap_dance_action);
  }
}

KALEIDOSCOPE_INIT_PLUGINS(TapDance);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.51 Keystrokes/TopsyTurvy/TopsyTurvy.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-TopsyTurvy -- Turn the effect of Shift upside down for certain keys
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-TopsyTurvy.h>

// clang-format off
KEYMAPS(
```

(continues on next page)

(continued from previous page)

```
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    TOPSY(1), TOPSY(2), TOPSY(3), TOPSY(4), TOPSY(5), Key_NoKey,
    Key_Backtick, Key_Q,    Key_W,    Key_E,    Key_R,    Key_T, Key_Tab,
    Key_PageUp,   Key_A,    Key_S,    Key_D,    Key_F,    Key_G,
    Key_PageDown, Key_Z,    Key_X,    Key_C,    Key_V,    Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_skip,

    Key_skip,  TOPSY(6), TOPSY(7), TOPSY(8),     TOPSY(9),      TOPSY(0),  Key_skip,
    Key_Enter, Key_Y,    Key_U,    Key_I,        Key_O,         Key_P,     Key_Equals,
    Key_H,     Key_J,    Key_K,    Key_L,        Key_Semicolon, Key_Quote,
    Key_skip,  Key_N,    Key_M,    Key_Comma,    Key_Period,    Key_Slash, Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(TopsyTurvy);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.52 Keystrokes/Turbo/Turbo.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Turbo
 * Copyright (C) 2018
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
```

(continues on next page)

(continued from previous page)

```
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-Turbo.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey, Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_skip,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_Turbo, Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl, Turbo);

void setup() {
  Kaleidoscope.setup();

  Turbo.interval(30);
  Turbo.sticky(true);
  Turbo.flash(true);
  Turbo.flashInterval(80);
  Turbo.activeColor(CRGB(0x64, 0x96, 0xed));
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.53 Keystrokes/Unicode/Unicode.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Unicode -- Unicode input helpers
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
```

(continues on next page)

(continued from previous page)

```
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-HostOS.h>
#include "Kaleidoscope-Macros.h"
#include <Kaleidoscope-Unicode.h>

enum { MACRO_KEYBOARD_EMOJI };

// clang-format off

KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_skip,

    M(MACRO_KEYBOARD_EMOJI), Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_
↪skip,
    Key_Enter,               Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_
↪Equals,
                             Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_
↪Quote,
    Key_skip,                Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_
↪Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_skip),
)
// clang-format on

static void unicode(uint32_t character, uint8_t keyState) {
  if (keyToggledOn(keyState)) {
    Unicode.type(character);
  }
}

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  switch (macro_id) {
```

(continues on next page)

(continued from previous page)

```
  case MACRO_KEYBOARD_EMOJI:
    unicode(0x2328, event.state);
    break;
  }
  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          HostOS,
                          Macros,
                          Unicode);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.54 Keystrokes/WinKeyToggle/WinKeyToggle.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-WinKeyToggle -- Toggle the Windows (GUI) key on/off
 * Copyright (C) 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-MagicCombo.h>
#include <Kaleidoscope-WinKeyToggle.h>

enum {
  WINKEYTOGGLE
};

void toggleWinKey(uint8_t index) {
  WinKeyToggle.toggle();
}
```

(continues on next page)

(continued from previous page)

```
USE_MAGIC_COMBOS([WINKEYTOGGLE] = {
                   .action = toggleWinKey,
                   .keys   = {R3C6, R3C9}});

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_NoKey, Key_1, Key_2, Key_3, Key_4, Key_5, Key_NoKey,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(MagicCombo, WinKeyToggle);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.55 LEDs/Colormap/Colormap.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-EEPROM-Colormap -- Per-layer colormap effect
 * Copyright (C) 2017-2022  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT but␣
↪WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
```

(continues on next page)

(continued from previous page)

```
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with along␣
↪with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-EEPROM-Keymap.h>
#include <Kaleidoscope-Colormap.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-LED-Palette-Theme.h>
#include <Kaleidoscope-LEDControl.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,    Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick, Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,   Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown, Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_NoKey,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_NoKey),
)

// Colors names of the EGA palette, for convenient use in colormaps. Should
// match the palette definition below. Optional, one can just use the indexes
// directly, too.
enum {
  BLACK,
  BLUE,
  GREEN,
  CYAN,
  RED,
  MAGENTA,
  BROWN,
  LIGHT_GRAY,
  DARK_GRAY,
  BRIGHT_BLUE,
  BRIGHT_GREEN,
  BRIGHT_CYAN,
```

(continues on next page)

(continued from previous page)

```
  BRIGHT_RED,
  BRIGHT_MAGENTA,
  YELLOW,
  WHITE
};

// Define an EGA palette. Conveniently, that's exactly 16 colors, just like the
// limit of LEDPaletteTheme.
PALETTE(
    CRGB(0x00, 0x00, 0x00),  // [0x0] black
    CRGB(0x00, 0x00, 0xaa),  // [0x1] blue
    CRGB(0x00, 0xaa, 0x00),  // [0x2] green
    CRGB(0x00, 0xaa, 0xaa),  // [0x3] cyan
    CRGB(0xaa, 0x00, 0x00),  // [0x4] red
    CRGB(0xaa, 0x00, 0xaa),  // [0x5] magenta
    CRGB(0xaa, 0x55, 0x00),  // [0x6] brown
    CRGB(0xaa, 0xaa, 0xaa),  // [0x7] light gray
    CRGB(0x55, 0x55, 0x55),  // [0x8] dark gray
    CRGB(0x55, 0x55, 0xff),  // [0x9] bright blue
    CRGB(0x55, 0xff, 0x55),  // [0xa] bright green
    CRGB(0x55, 0xff, 0xff),  // [0xb] bright cyan
    CRGB(0xff, 0x55, 0x55),  // [0xc] bright red
    CRGB(0xff, 0x55, 0xff),  // [0xd] bright magenta
    CRGB(0xff, 0xff, 0x55),  // [0xe] yellow
    CRGB(0xff, 0xff, 0xff)   // [0xf] white
)

COLORMAPS(
    [0] = COLORMAP_STACKED
    (BLACK,   GREEN, GREEN, GREEN, GREEN, GREEN, BLUE,
     MAGENTA, CYAN,  CYAN,  CYAN,  CYAN,  CYAN,  RED,
     BROWN,   CYAN,  CYAN,  CYAN,  CYAN,  CYAN,
     BROWN,   CYAN,  CYAN,  CYAN,  CYAN,  CYAN,  RED,

     LIGHT_GRAY, RED, LIGHT_GRAY, LIGHT_GRAY,
     BLACK,

     BLACK,      BRIGHT_GREEN, BRIGHT_GREEN, BRIGHT_GREEN, BRIGHT_GREEN, BRIGHT_GREEN,␣
↪BLACK,
     BRIGHT_RED, BRIGHT_CYAN,  BRIGHT_CYAN,  BRIGHT_CYAN,  BRIGHT_CYAN,  BRIGHT_CYAN, ␣
↪YELLOW,
                 BRIGHT_CYAN,  BRIGHT_CYAN,  BRIGHT_CYAN,  BRIGHT_CYAN,  BRIGHT_RED,  ␣
↪BRIGHT_RED,
     BLACK,      BRIGHT_CYAN,  BRIGHT_CYAN,  BRIGHT_RED,   BRIGHT_RED,   BRIGHT_RED,  ␣
↪BRIGHT_RED,

     DARK_GRAY, BRIGHT_RED, DARK_GRAY, DARK_GRAY,
     BLACK)
)

// clang-format on
```

(continues on next page)

(continued from previous page)

```
KALEIDOSCOPE_INIT_PLUGINS(EEPROMSettings,
                          EEPROMKeymap,
                          LEDControl,
                          LEDPaletteTheme,
                          LEDOff,
                          ColormapEffect,
                          DefaultColormap,
                          Focus,
                          FocusEEPROMCommand,
                          FocusSettingsCommand);

void setup() {
  Kaleidoscope.setup();

  EEPROMKeymap.setup(1);

  ColormapEffect.max_layers(1);
  ColormapEffect.activate();

  DefaultColormap.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.56 LEDs/FingerPainter/FingerPainter.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-FingerPainter -- On-the-fly keyboard painting.
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Palette-Theme.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-FingerPainter.h>
```

(continues on next page)

(continued from previous page)

```
#include <Kaleidoscope-FocusSerial.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_NoKey,         Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_skip,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_skip),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDOff,
                          EEPROMSettings,
                          LEDPaletteTheme,
                          FingerPainter,
                          Focus);

void setup() {
  Kaleidoscope.setup();

  EEPROMSettings.seal();

  FingerPainter.activate();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.57 LEDs/Heatmap/Heatmap.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Heatmap -- Heatmap LED effect for Kaleidoscope.
 * Copyright (C) 2016, 2017, 2018  Gergely Nagy
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-Heatmap.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
   Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
   Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
   Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

   Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
   Key_NoKey,

   Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
   Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
              Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
   Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

   Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
   Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          HeatmapEffect);

void setup() {
  Kaleidoscope.setup();

  HeatmapEffect.activate();
}
```

(continues on next page)

(continued from previous page)

```
void loop() {
  Kaleidoscope.loop();
}
```

11.22.58 LEDs/IdleLEDs/IdleLEDs.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-Idle-LEDs -- Turn off the LEDs when the keyboard's idle
 * Copyright (C) 2018, 2019  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-FocusSerial.h>
#include <Kaleidoscope-IdleLEDs.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
```

(continues on next page)

(continued from previous page)

```
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          EEPROMSettings,
                          Focus,
                          PersistentIdleLEDs,
                          LEDRainbowWaveEffect,
                          LEDOff);

void setup() {
  Kaleidoscope.serialPort().begin(9600);

  Kaleidoscope.setup();

  PersistentIdleLEDs.setIdleTimeoutSeconds(300);  // 5 minutes

  LEDRainbowWaveEffect.activate();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.59 LEDs/LED-ActiveLayerColor/LED-ActiveLayerColor.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LED-ActiveLayerColor -- Light up the LEDs based on the active layers
 * Copyright (C) 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-ActiveLayerColor.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
```

(continues on next page)

(continued from previous page)

```
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    ShiftToLayer(1),

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    ShiftToLayer(1)
  ),
  [1] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    ShiftToLayer(0),

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    ShiftToLayer(0)
  )
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDActiveLayerColorEffect);

void setup() {
  static const cRGB layerColormap[] PROGMEM = {
    CRGB(128, 0, 0),
    CRGB(0, 128, 0)};

  Kaleidoscope.setup();
  LEDActiveLayerColorEffect.setColormap(layerColormap);
}

void loop() {
  Kaleidoscope.loop();
```

(continues on next page)

(continued from previous page)

```
}
```

11.22.60 LEDs/LED-ActiveModColor/LED-ActiveModColor.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LED-ActiveModColor -- Light up the LEDs under the active modifiers
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-ActiveModColor.h>
#include <Kaleidoscope-OneShot.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_skip,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    OSM(RightShift), OSM(RightAlt), Key_Spacebar, OSM(RightControl),
    Key_skip),
)
// clang-format on

// OneShot is included to illustrate the different colors highlighting sticky
// and one-shot keys.  LEDOff is included because we need an LED mode active to
```

(continues on next page)

(continued from previous page)

```
// allow highlighted keys to return to "normal" when released (or timed out).
KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDOff,
                          ActiveModColorEffect,
                          OneShot);

void setup() {
  Kaleidoscope.setup();

  ActiveModColorEffect.setHighlightColor(CRGB(0x00, 0xff, 0xff));

  // Uncomment the following to enable OneShot on normal modifier keys:
  // OneShot.enableAutoOneShot();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.61 LEDs/LED-AlphaSquare/LED-AlphaSquare.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LED-AlphaSquare -- 4x4 pixel LED alphabet
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-AlphaSquare.h>
#include <Kaleidoscope-Macros.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, M(0),
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
```

(continues on next page)

(continued from previous page)

```
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_skip,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_skip),
)
// clang-format on

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  if (!keyToggledOn(event.state))
    return MACRO_NONE;

  if (macro_id == 0) {
    for (uint8_t i = Key_A.getKeyCode(); i <= Key_0.getKeyCode(); i++) {
      LEDControl.set_all_leds_to(0, 0, 0);
      LEDControl.syncLeds();
      delay(100);

      uint8_t col = 2;
      if (i % 2)
        col = 10;

      for (uint8_t step = 0; step <= 0xf0; step += 8) {
        AlphaSquare.color = {step, step, step};
        AlphaSquare.display({i, 0}, col);
        delay(10);
      }
      for (uint8_t step = 0xff; step >= 8; step -= 8) {
        AlphaSquare.color = {step, step, step};
        AlphaSquare.display({i, 0}, col);
        delay(10);
      }
      delay(100);
    }

    LEDControl.set_all_leds_to(0, 0, 0);
    LEDControl.syncLeds();
    delay(100);

    for (uint8_t step = 0; step <= 0xf0; step += 8) {
      AlphaSquare.color = {step, step, step};
      AlphaSquare.display(kaleidoscope::plugin::alpha_square::symbols::Lambda, 2);
      AlphaSquare.display(kaleidoscope::plugin::alpha_square::symbols::Lambda, 10);
      delay(10);
    }
```

(continues on next page)

(continued from previous page)

```
    for (uint8_t step = 0xff; step >= 8; step -= 8) {
      AlphaSquare.color = {step, step, step};
      AlphaSquare.display(kaleidoscope::plugin::alpha_square::symbols::Lambda, 2);
      AlphaSquare.display(kaleidoscope::plugin::alpha_square::symbols::Lambda, 10);
      delay(10);
    }
    delay(100);
  }

  LEDControl.set_all_leds_to(0, 0, 0);

  return MACRO_NONE;
}

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          AlphaSquare,
                          AlphaSquareEffect,
                          Macros);

void setup() {
  Kaleidoscope.setup();

  AlphaSquare.color = {0xcb, 0xc0, 0xff};
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.62 LEDs/LED-Brightness/LED-Brightness.ino

```
,/* -*- mode: c++ -*-
 * LED-Brightness.ino -- Example to show LED brightness control
 * Copyright (C) 2020  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
```

(continues on next page)

(continued from previous page)

```
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>
#include <Kaleidoscope-Macros.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    M(0),

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    M(1)),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          Macros,
                          LEDRainbowWaveEffect);

const macro_t *macroAction(uint8_t macro_id, KeyEvent &event) {
  if (keyToggledOn(event.state)) {
    uint8_t brightness = LEDControl.getBrightness();

    if (macro_id == 0) {
      if (brightness > 10)
        brightness -= 10;
      else
        brightness = 0;
    } else if (macro_id == 1) {
      if (brightness < 245)
        brightness += 10;
      else
        brightness = 255;
    }

    LEDControl.setBrightness(brightness);
  }

  return MACRO_NONE;
}
```

(continues on next page)

(continued from previous page)

```
void setup() {
  Kaleidoscope.setup();
  LEDRainbowWaveEffect.brightness(255);
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.63 LEDs/LED-Palette-Theme/LED-Palette-Theme.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LED-Palette-Theme -- Palette-based LED theme foundation
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Palette-Theme.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-FocusSerial.h>

namespace example {

class TestLEDMode : public kaleidoscope::plugin::LEDMode {
 public:
  TestLEDMode() {}

  kaleidoscope::EventHandlerResult onFocusEvent(const char *input);

 protected:
  void setup() final;
  void update(void) final;

 private:
  static uint16_t map_base_;
};
```

(continues on next page)

(continued from previous page)

```
uint16_t TestLEDMode::map_base_;

void TestLEDMode::setup() {
  map_base_ = LEDPaletteTheme.reserveThemes(1);
}

void TestLEDMode::update(void) {
  LEDPaletteTheme.updateHandler(map_base_, 0);
}

kaleidoscope::EventHandlerResult
TestLEDMode::onFocusEvent(const char *input) {
  return LEDPaletteTheme.themeFocusEvent(input, PSTR("testLedMode.map"), map_base_, 1);
}

}  // namespace example

example::TestLEDMode TestLEDMode;

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(Focus, LEDPaletteTheme, TestLEDMode, EEPROMSettings);

void setup() {
  Kaleidoscope.setup();

  TestLEDMode.activate();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.64 LEDs/LED-Stalker/LED-Stalker.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LED-Stalker -- Stalk keys pressed by lighting up and fading back the LED␣
↪under them
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Stalker.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDOff,
                          StalkerEffect);

void setup() {
  Kaleidoscope.setup();
```

(continues on next page)

(continued from previous page)

```
  StalkerEffect.variant = STALKER(BlazingTrail);
  StalkerEffect.activate();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.65 LEDs/LED-Wavepool/LED-Wavepool.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LED-Wavepool
 * Copyright (C) 2017 Selene Scriven
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LED-Wavepool.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
        (
          Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
          Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
          Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
          Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

          Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
          Key_NoKey,

          Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
          Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
          Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
          Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

          Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
```

(continues on next page)

(continued from previous page)

```
          Key_NoKey
        )
) // KEYMAPS(

KALEIDOSCOPE_INIT_PLUGINS(
  LEDControl,
  LEDOff,
  WavepoolEffect
);
// clang-format on

void setup() {
  Kaleidoscope.setup();

  WavepoolEffect.idle_timeout = 5000;  // 5 seconds
  WavepoolEffect.ripple_hue   = WavepoolEffect.rainbow_hue;
  WavepoolEffect.activate();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.66 LEDs/LEDEffect-BootGreeting/LEDEffect-BootGreeting.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LEDEffect-BootGreeting -- Small greeting at boot time
 * Copyright (C) 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-BootGreeting.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
```

(continues on next page)

(continued from previous page)

```
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          BootGreetingEffect,
                          LEDOff);

void setup() {
  Kaleidoscope.setup();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.67 LEDs/LEDEffects/LEDEffects.ino

```
,/* -*- mode: c++ -*-
 * Kaleidoscope-LEDEffects -- An assorted collection of LED effects for Kaleidoscope
 * Copyright (C) 2016, 2017, 2018  Keyboard.io, Inc
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
```

(continues on next page)

(continued from previous page)

```
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffects.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          LEDOff,
                          MiamiEffect,
                          JukeboxEffect,
                          JukeboxAlternateEffect);

void setup() {
  Kaleidoscope.setup();

  MiamiEffect.activate();
}

void loop() {
  Kaleidoscope.loop();
}
```

11.22.68 LEDs/PersistentLEDMode/PersistentLEDMode.ino

```
,/* -*- mode: c++ -*-
 * kaleidoscope::plugin::PersistentLEDMode -- Persist the current LED mode to Storage
 * Copyright (C) 2019  Keyboard.io, Inc.
 * Copyright (C) 2019  Dygma, Inc.
 *
 * This program is free software: you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free Software
```

(continues on next page)

(continued from previous page)

```
 * Foundation, version 3.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program. If not, see <http://www.gnu.org/licenses/>.
 */

#include <Kaleidoscope.h>
#include <Kaleidoscope-EEPROM-Settings.h>
#include <Kaleidoscope-LEDControl.h>
#include <Kaleidoscope-LEDEffect-Rainbow.h>
#include <Kaleidoscope-LEDEffect-Breathe.h>
#include <Kaleidoscope-LEDEffect-Chase.h>
#include <Kaleidoscope-PersistentLEDMode.h>

// clang-format off
KEYMAPS(
  [0] = KEYMAP_STACKED
  (
    Key_LEDEffectNext, Key_1, Key_2, Key_3, Key_4, Key_5, Key_LEDEffectNext,
    Key_Backtick,      Key_Q, Key_W, Key_E, Key_R, Key_T, Key_Tab,
    Key_PageUp,        Key_A, Key_S, Key_D, Key_F, Key_G,
    Key_PageDown,      Key_Z, Key_X, Key_C, Key_V, Key_B, Key_Escape,

    Key_LeftControl, Key_Backspace, Key_LeftGui, Key_LeftShift,
    Key_NoKey,

    Key_skip,  Key_6, Key_7, Key_8,     Key_9,      Key_0,         Key_skip,
    Key_Enter, Key_Y, Key_U, Key_I,     Key_O,      Key_P,         Key_Equals,
               Key_H, Key_J, Key_K,     Key_L,      Key_Semicolon, Key_Quote,
    Key_skip,  Key_N, Key_M, Key_Comma, Key_Period, Key_Slash,     Key_Minus,

    Key_RightShift, Key_RightAlt, Key_Spacebar, Key_RightControl,
    Key_NoKey),
)
// clang-format on

KALEIDOSCOPE_INIT_PLUGINS(LEDControl,
                          EEPROMSettings,
                          PersistentLEDMode,
                          LEDRainbowWaveEffect,
                          LEDRainbowEffect,
                          LEDChaseEffect,
                          LEDBreatheEffect,
                          LEDOff);

void setup() {
  Kaleidoscope.setup();
```

(continues on next page)

(continued from previous page)

```
}

void loop() {
  Kaleidoscope.loop();
}
```

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER THIRTEEN

LINKS

Source code on GitHub