The threeboard project

ABOUT

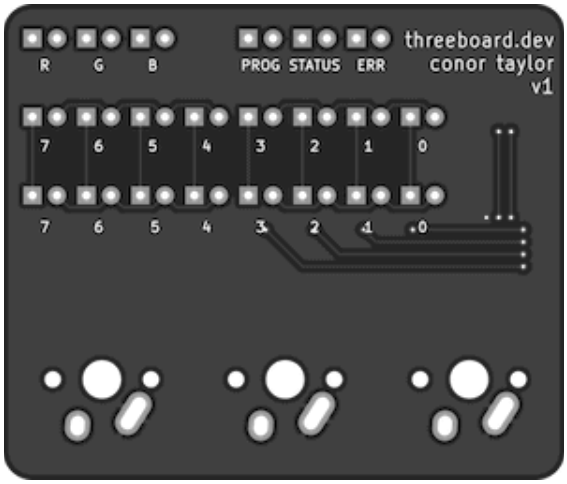
This document describes the design, functionality and architecture of the threeboard project: a fully-functional mechanical USB keyboard with only three keys.

A full copy of the firmware source code, hardware design definitions and collected documentation is available at github.com/taylorconor/threeboard.

Author Conor Taylor - taylorconor.com

Date December 2021

OVERVIEW



threeboard is a fully-functional mechanical USB keyboard with only three keys. It supports multiple programmable layers, and achieves the same functionality as a full-sized keyboard. Its firmware and hardware are built completely from scratch and are extensively documented.

Unlike a traditional keyboard, characters don't show up on the screen after each keypress, because there aren't enough keys. Instead, combinations of the three keys are used to specify key and modifier codes on two built-in 8-bit LED binary indicators. This can then be sent to the host computer as USB keycodes. The multiple layers allow users to program macros into the threeboard's storage for quick retrieval.

MOTIVATION

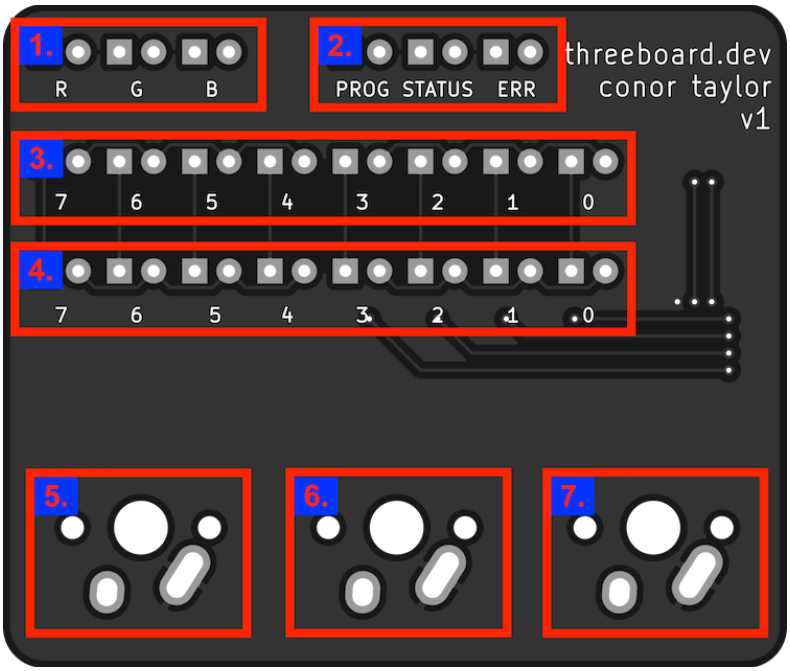
The main goal of the threeboard project is to build a relatively easy to understand, readable, self-contained and well-documented embedded software and hardware project. The threeboard is built from scratch, with no external dependencies. This means that all components, from the USB stack to the PCB hardware designs, are written from the ground up and are all contained in the threeboard git repository.

The threeboard project is extensively documented: within the firmware itself as comments, in a set of detailed markdown documents, and in this design document. The primary design goal of

the C++ firmware code is readability, to enable as many people to read and learn from it as possible, regardless of background. It's well tested, including end-to-end integration tests which execute tests against the firmware in a hardware simulator to emulate the physical hardware.

FUNCTIONALITY

The threeboard uses three keys to select keycodes to send over USB, and also provides three programmable layers to store and retrieve key combinations. The state of the keyboard is visualised using the 22 onboard LED lights, and can only be modified using the three keys. The image below highlights the different features of the device:



1. 3 LEDs indicating the currently selected keyboard layer (`R`, `G` and `B`).
2. 3 status LEDs for indicating the operating status of the threeboard (PROG, STATUS and ERR)
3. Bank0, a binary byte display of 8 LEDs. Used for displaying layer-specific information (e.g. displaying the USB keycode)
4. Bank1, another binary byte display with 8 LEDs. Used for displaying a different byte of layer-specific information (e.g. displaying the USB modifier code).
5. Mechanical key `X`.
6. Mechanical key `Y`.
7. Mechanical key `Z`.

A table mapping USB keycode and modcode bytes to key values are defined in section 10 of the HID usage tables document.

The user-facing functionality is split into layers. A layer changes the functionality of each key on the threeboard, similar to a shift or function key on a traditional keyboard. The threeboard has four different layers: a default `DFLT` layer, and three programmable layers:

- DFLT: The default layer of the threeboard. This is the layer that the threeboard starts in when it boots. No layer LEDs are lit in this layer. This is a single character input layer, where characters and modifiers in this layer are identified by their USB key codes.
- R: The character reprogramming layer of the threeboard. This layer allows users to reassign a keycode to a character. This allows faster input of common characters.
- G: The word shortcut layer. This layer allows users to program frequently used words into the threeboard. These words can then be accessed and sent over USB.
- B: The blob shortcut layer. This allows users to program arbitrary text blobs (including mod codes), and access them later.

The three programmable layers each have two modes: DFLT and PROG:

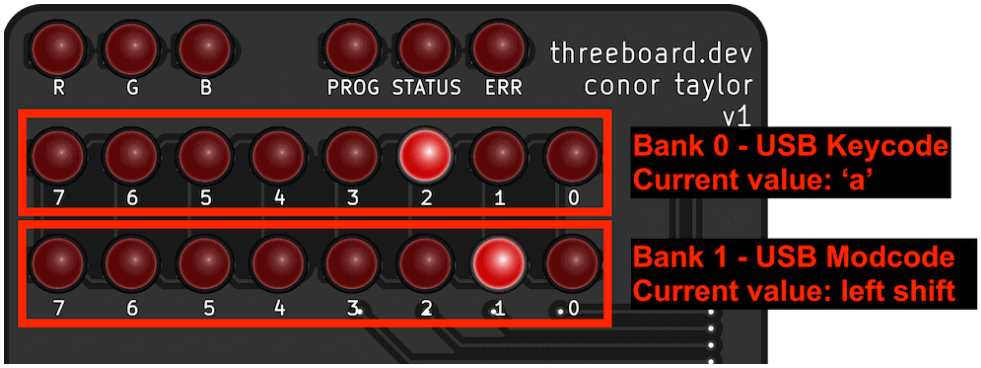
- DFLT: This is the default mode of the programmable layers. It is used to retrieve shortcuts that were programmed in the PROG mode, and send them over USB.
- PROG: This mode is used for programming the shortcuts used in the DFLT mode of the layer.

Each layer and mode defines its actions based on the input keypresses. Since there are many possible states, the full mapping of layers, modes, key combinations and actions is quite large, and is defined in the threeboard usage table in the appendix. However, most layers share some common actions for some keypresses:

- Key X: Increment bank 0.
- Key Y: Increment bank 1.
- Key Z: Send keypress(es) over USB (in DFLT mode), or store shortcut in PROG mode.
- Key combo X+Z: Clear bank 0.
- Key combo Y+Z: Clear bank 1.
- Key combo X+Y+Z: Go to the next layer (when in DFLT mode), or exit PROG mode otherwise.
- Key combo X+Y: Enter PROG mode (when in DFLT mode).

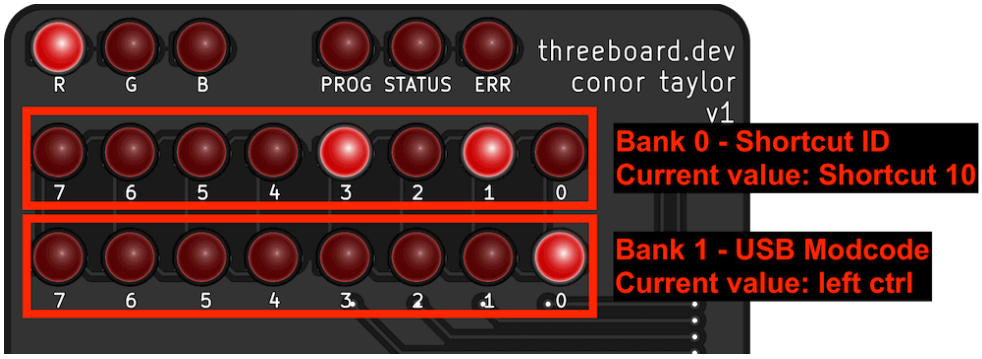
Brief examples of scenarios in each layer and mode configuration are visualised below:

DFLT Layer

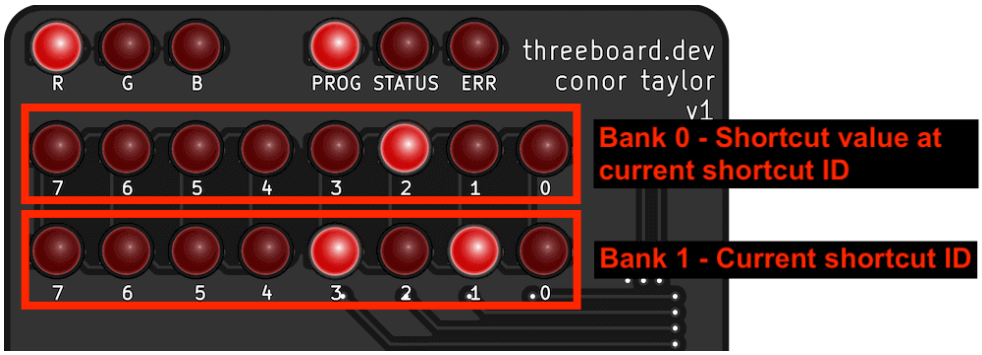


In the DFLT layer, the byte in bank 0 represents the raw USB keycode value to be sent to the host computer, and bank 1 represents the USB modifier code. This layer has no PROG mode.

Layer R

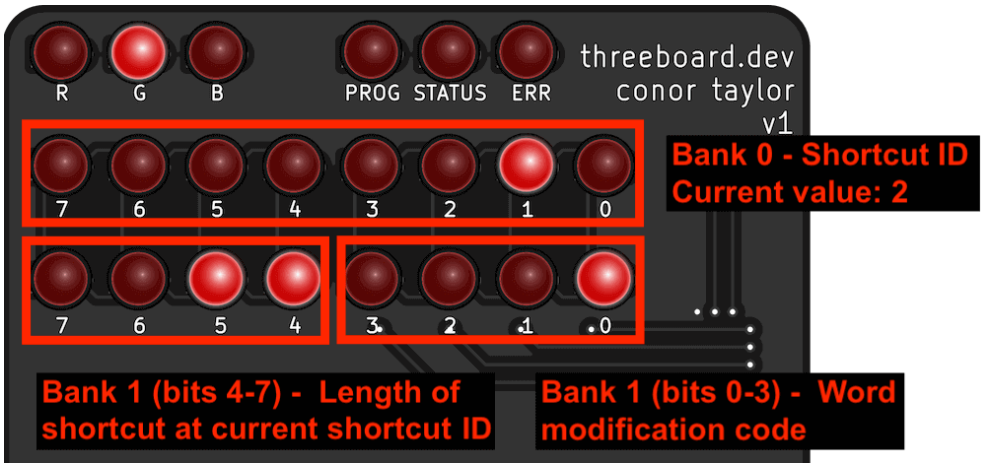


Layer R is used for character reprogramming, which allows re-assigning a value of bank 0 to a different USB keycode than it represents in the `DFLT` layer. When this layer is being used in `DFLT` mode, bank 0 is used to identify the shortcut ID, and bank 1 is used as it is in the `DFLT` layer, to specify the USB modifier code.



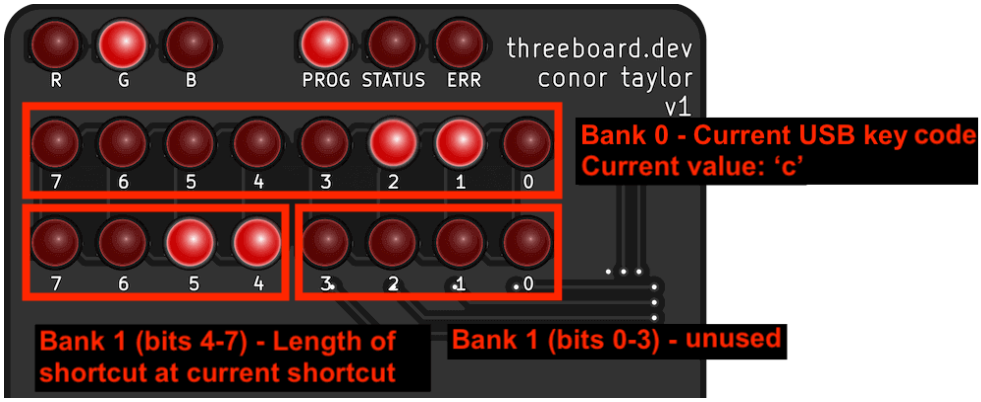
In `PROG` mode, the current shortcut value at the specified shortcut ID is displayed in bank 0. The shortcut ID is instead displayed in bank 0.

Layer G



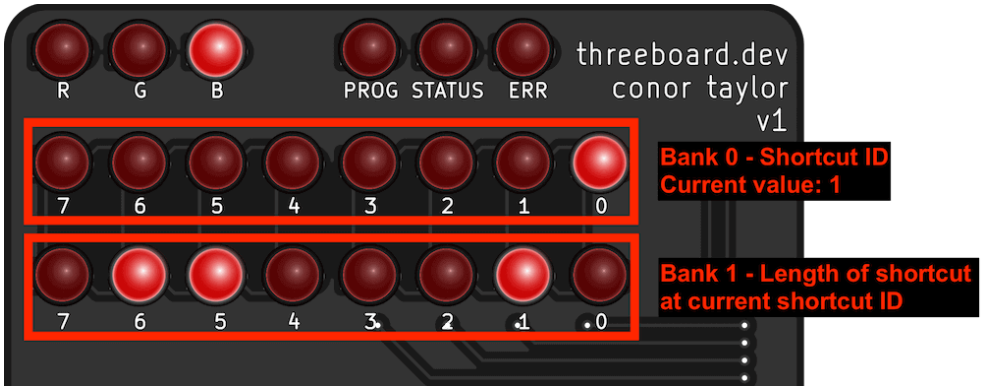
Layer G is used for word shortcut programming. Words up to 15 characters long can be stored per shortcut ID, and modifier codes are used to apply USB modifier codes to the word shortcut (such as capitalizing the first letter). In DFLT mode, bank 1 is split into two 4-bit indicators to display two different values; the 4 high bits are used to display the length of the current shortcut, and the 4 low bits display the word modification code. Word modification codes cause the following modifications:

- Code 0: Word is sent in lowercase.
- Code 1: Word is sent in uppercase.
- Code 2: The first letter of the word is capitalized.
- Code 3: A period (.) is appended to the end of the word.
- Code 4: A comma (,) is appended to the end of the word.
- Code 5: A hyphen (-) is appended to the end of the word.
- Codes 6-15: Reserved. The word is sent in lowercase.

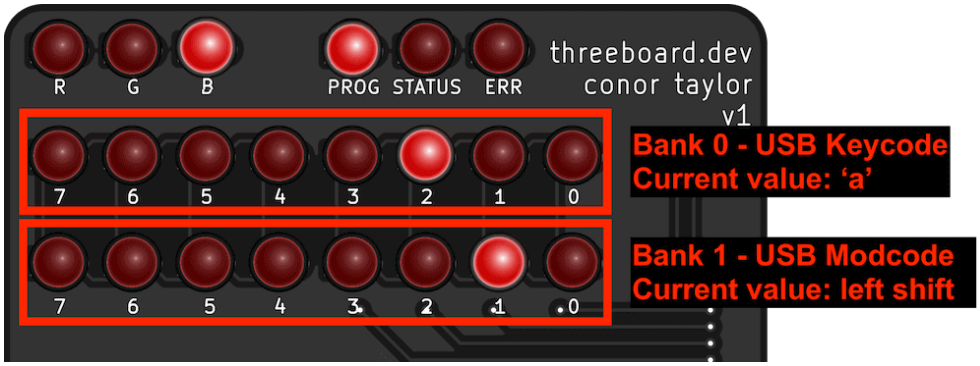


In PROG mode, the current USB keycode to be appended to the word shortcut is displayed in bank 0. The 4 high bits of bank 1 continues to display the length of the word shortcut, but the 4 low bits are unused and always set to 0.

Layer B



Layer B allows for freeform blob shortcut programming. Blobs of up to 255 keycode and modcode pairs can be stored. In DFLT mode, bank 0 displays the current shortcut ID, and bank 1 displays the length of the shortcut stored there.



In PROG mode, bank 0 displays the USB keycode being programmed, and bank 1 displays the modcode.

TECHNICAL INTRODUCTION

The remainder of this document is split into three large sections: firmware, simulator, and hardware. These sections provide an in-depth description of the background, design and architecture of both of these components, as well as detailed descriptions of some peripheral information, such as testing and simulating.

As already mentioned, the threeboard's firmware and hardware are both built from scratch. Because of this, some features of firmware are inspired or required by properties of the hardware, and some features of hardware are inspired or required by the possibilities afforded by the firmware. For this reason, the firmware and hardware sections of this document contain links to each other. It's possible to navigate between these sections to read relevant information about a feature of the threeboard from both a firmware and a hardware perspective, rather than reading the document from start to finish.

The appendix of this document contains the threeboard usage table, a definitive list of all of the possible threeboard states, inputs and outputs. Additionally, the appendix includes an explanation of how USB keyboards work in general. This might be good to read to understand the background of some of the concepts discussed in the document, but it's not threeboard-specific.

FIRMWARE

Introduction

The threeboard's firmware has two main functions. The first is to configure and interface with each hardware component on the threeboard (such as LEDs, EEPROMs and USB) according to their respective specifications. The second is to provide user-facing features and functionality to the threeboard as described in the functionality section.

Instructions on how to compile the threeboard firmware are available in the firmware build instructions section of the appendix.

Constraints

The threeboard uses an atmega32u4 microcontroller (MCU), a low-power 8-bit microcontroller based on the AVR enhanced RISC architecture. Firmware code compiled for this MCU uses the Atmel AVR instruction set.

The MCU itself has the following relevant features:

- 8-bit AVR architecture.
- 16MHz maximum frequency at 4.5V.
- 32KB flash-based program memory.
- 2.5KB SRAM.
- 1KB internal EEPROM.
- USB controller hardware support.
- Two 8-bit and two 16-bit timers.
- Byte oriented 2-wire serial interface.

Due to these MCU constraints, this is a bare-metal firmware project. This means that the firmware's instructions are executed directly on the MCU hardware, without an intervening operating system.

As a result of the lack of operating system, hardware MMU, and the severely restricted RAM size, the firmware does not perform any dynamic memory allocation (i.e. no heap allocation). All memory allocation is stack-based.

It's important to mention that there is no official C++ standard library support for AVR targets. Therefore targets compiled with the AVR C++ compiler, avr-gcc, will not link against libstdc++. All firmware code in the threeboard project cannot use any C++ standard library functions or headers.

Design principles

The threeboard firmware was designed with several core principles in mind. These are listed below, along with the reasoning for their inclusion:

1. Readability: This is the primary design principle of the firmware. The C++ source itself

should be clear, consistent, and well documented. It should be possible for anyone with passable C/C++ knowledge and no embedded programming knowledge to read it and learn from it.
    - Reasoning: Source code with consistent style reduces unnecessary mental load when reading the code, as the code style is predictable. Additionally, the motivation section above states that as many people as possible should be able to read and learn from this project, regardless of engineering background. Reducing readability friction helps achieve this goal.
2. Modularity: There should be a clear separation of concerns between different modules of the code, and no interdependence. Modules that need to affect change on other modules should do so via delegation.
    - Reasoning: It's hard to determine if a class fulfils its responsibilities if it has multiple distinct responsibilities, and it's hard to guarantee separation of concerns when this mix of responsibilities inevitably leads to reduced encapsulation. Reduced encapsulation causes dependency hell, which makes code hard to maintain and greatly reduces readability by making the control flow difficult to follow.
3. Testability: Each module must be testable in isolation. All external module dependencies should be abstract to enable mocking and deterministic testing.
    - Reasoning: Untested code should be assumed to have bugs, so full test coverage is essential. Making classes easier to test in isolation makes it easier to write concise test cases.

Toolchain

The atmega32u4 chip manufacturer (Atmel / Microchip) provides a toolchain for use with AVR development, in particular avr-gcc, a GCC compiler for AVR. This toolchain works out of the box for all stages of firmware development.

The threeboard firmware is built with Bazel. Bazel was chosen over more traditional build systems like CMake because it has the following useful properties:

1. Explicit dependencies: All dependencies on a translation unit must be explicitly defined in the Bazel BUILD file, and each translation unit is compiled in a sandbox environment that only has access to the dependencies that have been explicitly declared. This makes it impossible to introduce implicit project dependencies. Additionally, Bazel fails the build if it detects a dependency cycle.
2. Customisable target declarations: Targets can be used to produce multiple object files, compiled with different compiler settings and even with different compilers. This allows compatible modules to be compiled for both x86 and AVR, depending on the context in which it's being compiled.

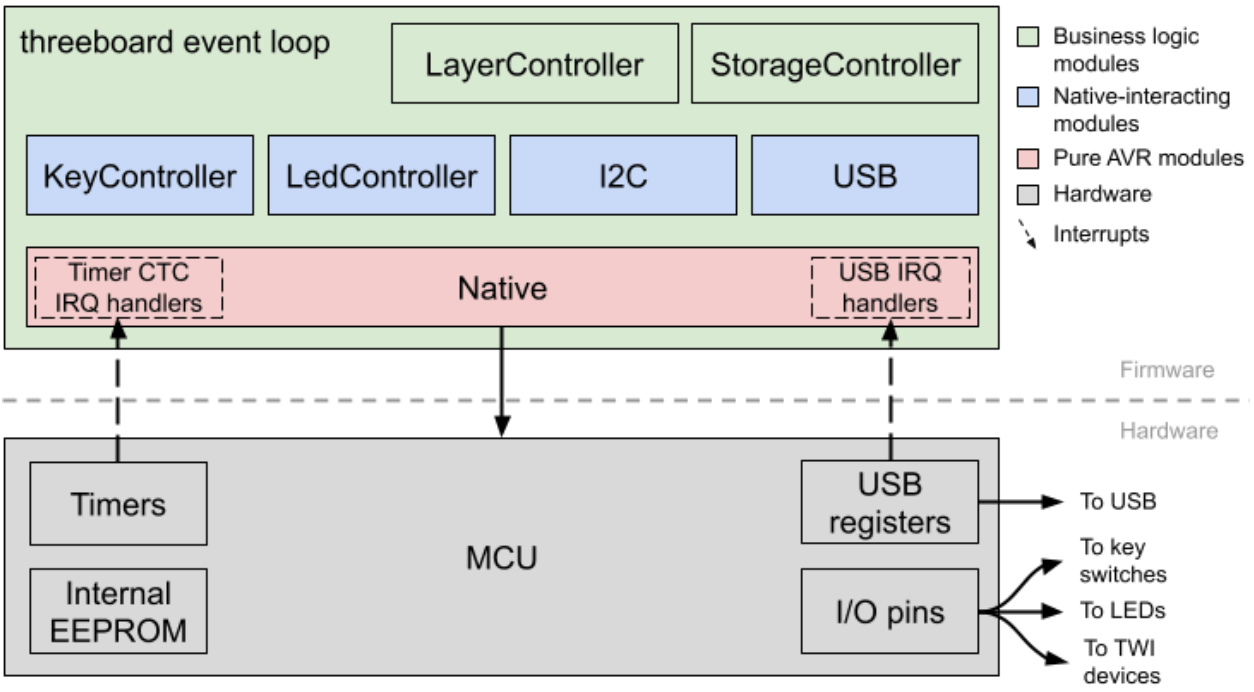
Bazel has no pre-existing support for the AVR toolchain, so support had to be added for this project. The avr-bazel repository was built especially for the threeboard project. It contains some special Bazel build rules to produce joint x86 and AVR build targets. This allows defining a single target definition per BUILD file to produce libraries for both targets, to avoid having to double-declare them.

Architecture

Overview

The architecture of the threeboard firmware can roughly be grouped into three categories of modules/classes:

1. Business logic modules: These codify the functionality of the firmware, and coordinate other modules to execute this functionality.
2. Native-interacting modules: Generally serve the purpose of configuring and controlling a single hardware element, such as LEDs or USB.
3. Pure AVR modules: Modules that can only be compiled for the AVR architecture, i.e. those that interact directly with hardware.

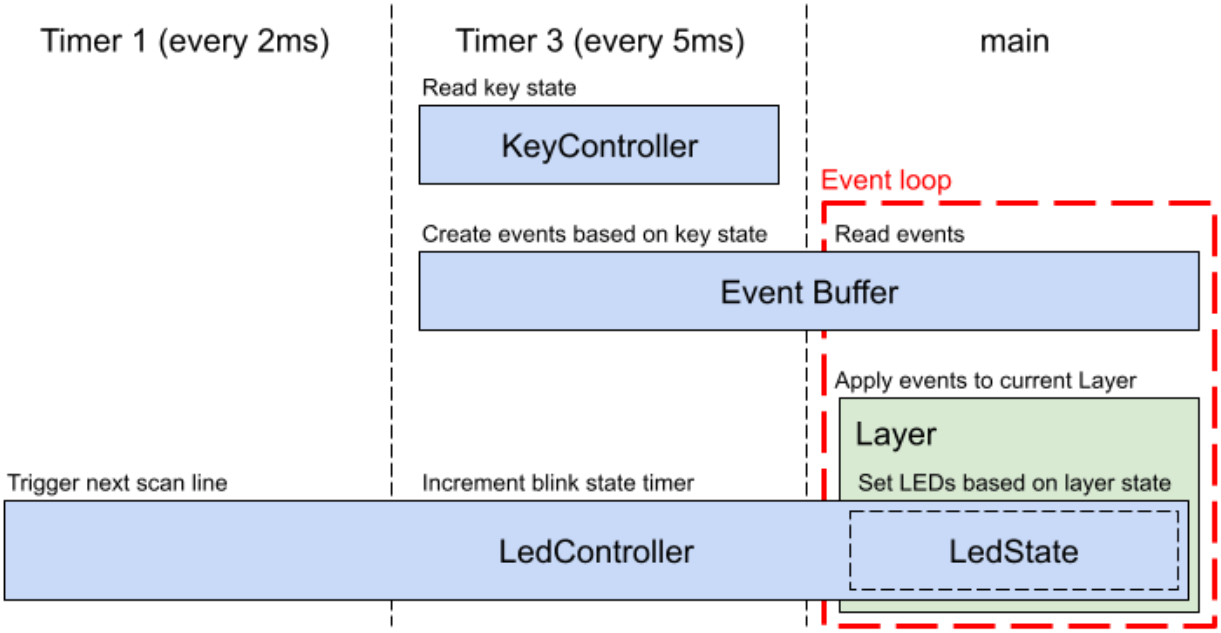


Everything except pure AVR modules target both AVR and x86 targets, which enables tests to run on x86 development hosts. These modules don't depend on any avr-libc headers, so all of their interactions with hardware are proxied through the firmware's `Native` interface, which exposes an API to interact with hardware. The interface itself does not depend on avr-libc, so non-pure modules can depend on it. When the firmware is compiled to be run on an AVR host or simulator, an avr-libc dependent `Native` implementation (`NativeImpl`) is used.

Event loop

The core of the threeboard firmware is a simple event loop; a design pattern that waits for and dispatches events to relevant modules when they arrive. The event loop is a never-ending loop that runs as long as the firmware is running (i.e. as long as the device has power).

The responsibilities of the different timers used, and the encapsulation afforded by the event loop is summarised in the diagram below:



Each event loop iteration is triggered by an interrupt. Two timer-based interrupts are used in the threeboard firmware: Timer 1 in CTC mode is configured to produce a software interrupt every 2ms, and is used to refresh the next LED scan line; Timer 3, also in CTC mode, is configured to produce a software interrupt every 5ms, used mainly to poll the key switches and produce events on the event buffer if necessary. Running timer 3 slower and separately from timer 1 allows us to avoid debouncing the Cherry MX key switches in software. Debouncing is discussed in more detail in the mechanical key switch section in hardware.

The purpose of the event loop is to receive and process all keypress events according to the actions defined in the current `Layer` of the threeboard. Each `Layer` instance encapsulates all business logic relating to inputs and actions for a given layer, so this doesn't need to happen in a long list of if/else statements within the main program loop.

```
void Threeboard::RunEventLoop() {
  // USB setup and configuration.
  ...

  // Main event loop.
  while (true) {
    // Atomically check for new keyboard events, and either handle them or
    // sleep the CPU until the next interrupt.
    native_->DisableInterrupts();
    if (!event_buffer_->HasEvent()) {
      // Sleep the CPU until another interrupt fires.
      native_->EnableCpuSleep();
      native_->EnableInterrupts();
      native_->SleepCpu();
      native_->DisableCpuSleep();
    } else {
      if (event_buffer_->HasKeypressEvent()) {
        layer_controller_.HandleEvent(event_buffer_->GetKeypressEvent());
```

```
      }
      // Re-enable interrupts after handling the event.
      native_->EnableInterrupts();
    }
  }
}
```

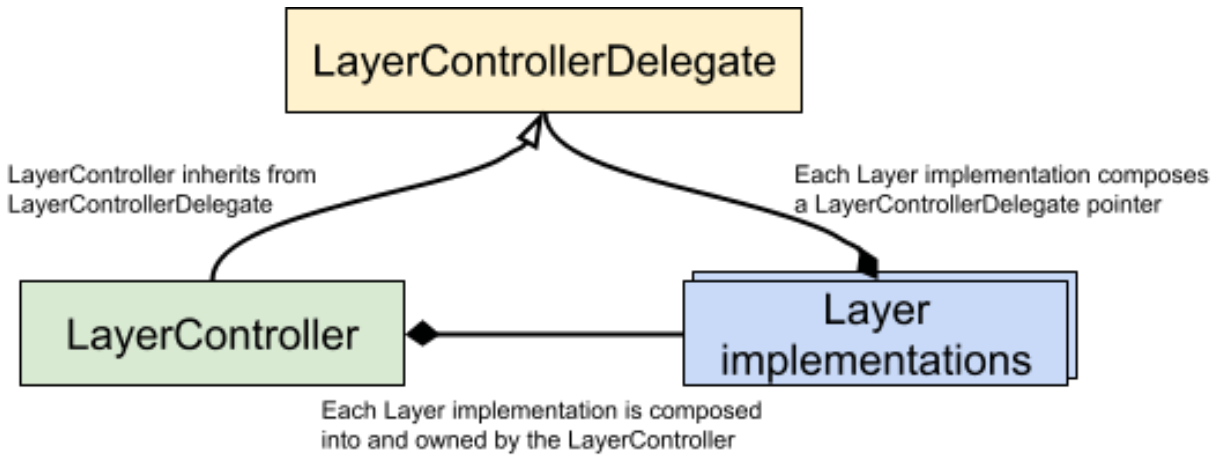
The sleep enabled bit is set before each sleep and cleared after each sleep, as recommended in the atmega32u4 datasheet (section 7.9.1): *it is recommended to write the Sleep Enable (SE) bit to one just before the execution of the SLEEP instruction and to clear it immediately after waking up.*

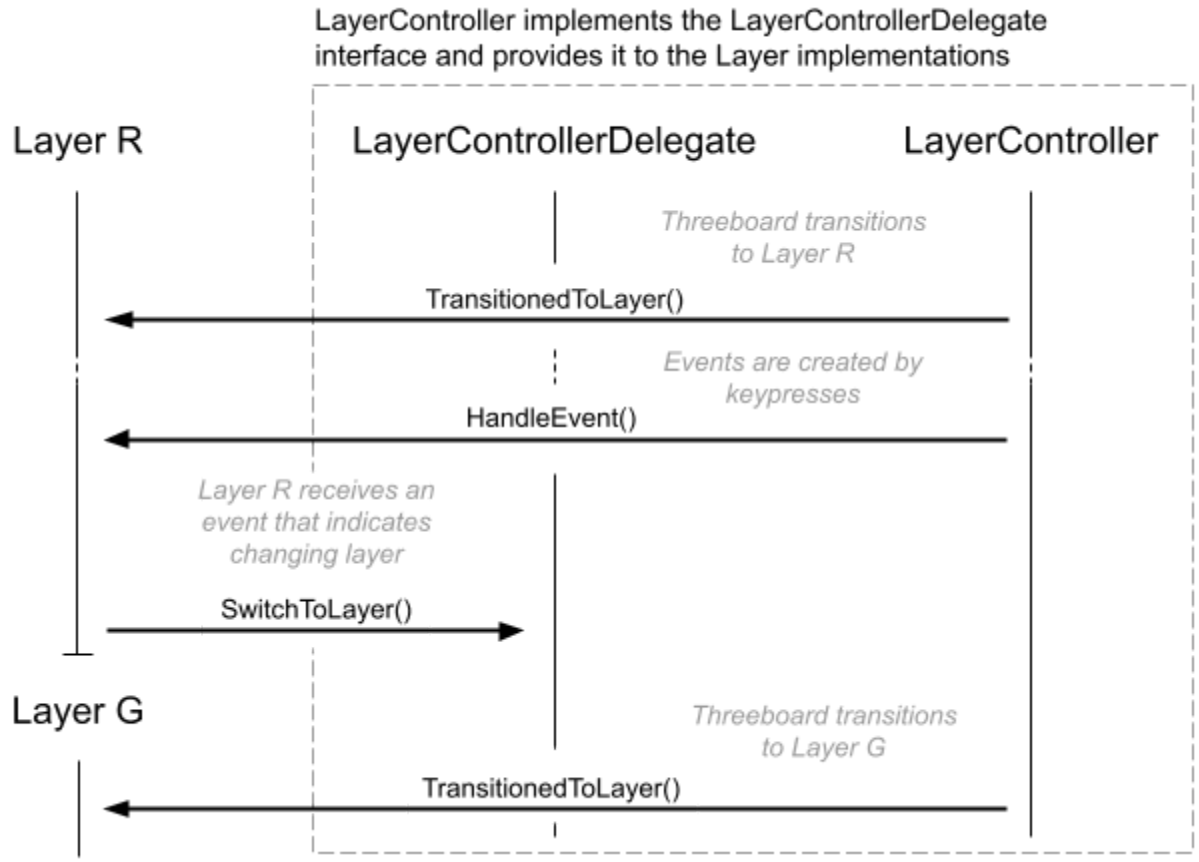
When finished handling events in the event buffer, the event loop puts the MCU into idle mode. In this mode, the CPU clock is stopped, but the clock powering the MCU's timers continues. The MCU is brought out of idle mode when the timer produces a software interrupt. The purpose of the event loop can therefore be thought of as to efficiently facilitate delegation between each module in the threeboard, without requiring busy-waiting.

Delegation

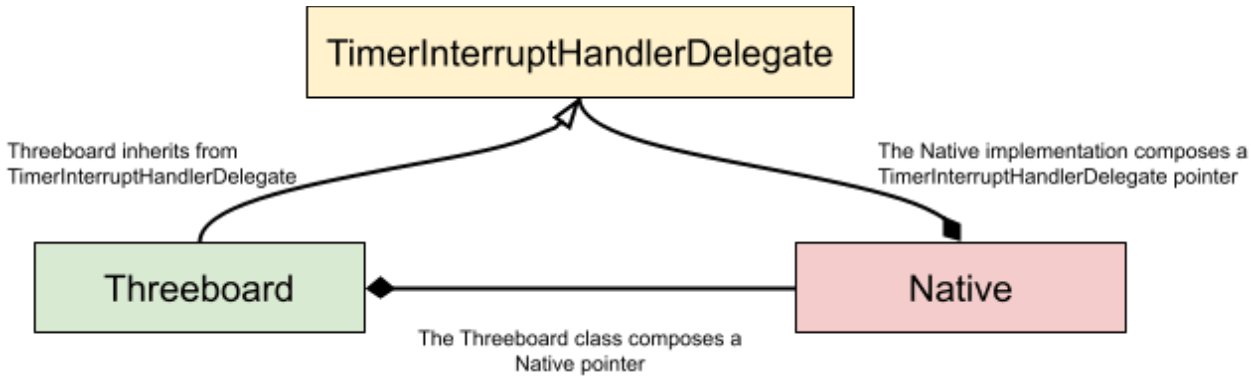
The threeboard makes extensive use of delegation to allow different modules in the firmware to communicate with each other without introducing circular dependencies.

In certain situations, two-way function calling between modules is necessary. The first of such situations this document will explain is the `LayerController`. It needs to be able to provide events to individual `Layers` to be handled. But some of these events may trigger an action that affects the `LayerController` itself, such as a keypress event that triggers the threeboard to switch layers. In this case, the current `Layer` uses the `LayerControllerDelegate` to call `SwitchToLayer()` on the `LayerController`, without having to explicitly depend on the `LayerController`:



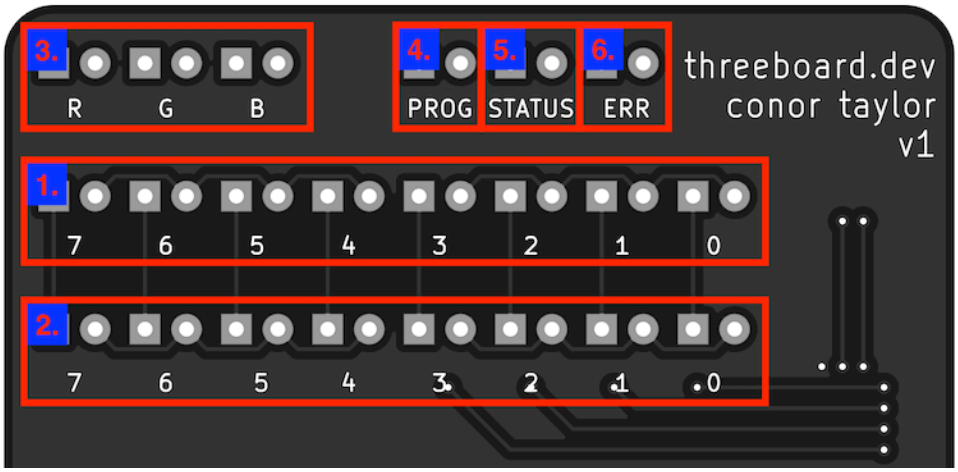


The `Threeboard`'s event loop receives timer interrupts in a similar fashion to the `LayerController`. In this case, the `Native` implementation is not owned by the `Threeboard` class, instead it's passed in as part of the bootstrapping process. This allows `Native`-specific timers to call functions in the `Threeboard` class (via the `TimerInterruptHandlerDelegate`) without introducing a circular dependency:



LED indicators

The LED indicators on the threeboard are vital to understanding the current state of the threeboard. The hardware is equipped with 22 single-colour LEDs. These are arranged as follows:



1. 8 LEDs to display the binary value of `bank_0` (1 byte).
2. 8 LEDs to display the binary value of `bank_1` (1 byte).
3. 3 LEDs to display the current layer, which is one of `R`, `G` or `B` (or `DFLT` if no LEDs are lit).
4. 1 `PROG` LED to identify when the device is in `PROG` mode.
5. 1 `ERR` LED to indicate an error.
6. 1 `STATUS` LED to provide context-specific status information.

The hardware-specific details of these LEDs, such as their wiring and tolerance, is described in the hardware section.

The LED indicator lights are all controlled by the `LedController`. This class is responsible for maintaining the state of the raster scan and outputting the `LedState` (a simple class for storing the state of each LED) to hardware. Its interface is very simple:

```
class LedController {
 public:
  explicit LedController(native::Native *native);

  // Handles rendering of the next scan row. Called by the timer interrupt
  // handler every 2ms.
  void ScanNextLine();

  // Handles timing of LED blinking. Called by the timer interrupt handler every
  // 5ms.
  void UpdateBlinkStatus();

  // state_ is guaranteed to live for the entire lifetime of the firmware.
  LedState *GetLedState() { return &state_; }
};
```

As discussed in the hardware section, the LEDs on the threeboard are arranged in a 5 row, 4 column matrix. When the `ScanNextLine()` function is called, the active LEDs in the next row of the matrix are lit, and all other LEDs are turned off. `ScanNextLine()` is called every 2ms by timer 1, which provides a 100Hz refresh rate on the threeboard's LEDs.

The `LedController` is also responsible for maintaining the timing of blinking LEDs. `UpdateBlinkStatus()` increments an 8-bit blink timer. Two blink states are supported by the threeboard: `LedState::BLINK` flashes the LED when bit 7 of the timer toggles (every 640ms), and `LedState::BLINK_FAST` flashes when bit 6 toggles (every 320ms).

USB stack

The threeboard contains a full USB device implementation. The threeboard presents itself to hosts as a USB version 2.0 keyboard HID device, complies with HID version 1.11, and passes USB CV test specification 0.72.

The USB protocol is extremely complex and so is not explained in full in this document. To learn how USB works, I recommend reading how do USB keyboards work? In the appendix of this document, the USB in a NutShell series of articles, and to use the USB 2.0 specification as a reference. This section will summarise the design and architecture of the threeboard's USB stack without going into any detail about the USB standard.

In the threeboard firmware, the USB stack is abstracted behind a minimal interface:

```
class Usb {
 public:
  // Setup the USB hardware to begin connection to the host. This method blocks
  // until the device hardware is correctly set up. If this method returns false
  // it means setup was halted by an error issued via the ErrorHandlerDelegate
  // and is in an undefined state.
  virtual bool Setup() = 0;

  // Returns true if the HID device configuration has completed successfully.
  virtual bool HasConfigured() = 0;

  // Send the provided key and modifier code to the host device. Returns false
  // if an error occurred during sending.
  virtual bool SendKeypress(uint8_t key, uint8_t mod) = 0;
};
```

The `UsbImpl` class is the concrete implementation of the `Usb` interface and interacts with the MCU's USB registers and functionality using the `Native` interface. Failure to set up correctly via `Usb::Setup()`, and subsequent failure to configure within the first 50 USB frames (50ms) using `Usb::HasConfigured()` will stall the threeboard's firmware startup logic, and enter an error state that continuously attempts to establish a connection to the host over USB.

USB setup begins with the firmware configuring the MCU to enable an electrical USB connection with the host device, as defined in the atmega32u4 datasheet, section 21.12. For example, the MCU contains phase-locked loop (PLL) hardware. When enabled, this can convert the external 16MHz clock signal to a 12MHz clock needed for the USB controller hardware in the MCU when configured in full-speed mode (atmega32u4 datasheet, section 21.2):

```
bool UsbImpl::Setup() {
```

```
  // Enable the USB pad regulator (which uses the external 1uF UCap).
  native_->SetUHWCON(1 << native::UVREGE);
  // Enable USB and freeze the clock.
  native_->SetUSBCON((1 << native::USBE) | (1 << native::FRZCLK));
  // PLL Control and Status Register. Configure the prescaler for the 16MHz
  // clock, and enable the PLL.
  native_->SetPLLCSR((1 << native::PINDIV) | (1 << native::PLLE));
  // Busy loop to wait for the PLL to lock to the 16MHz reference clock.
  WAIT_OR_RETURN(!(native_->GetPLLCSR() & (1 << native::PLOCK)), UINT16_MAX,
                 "Failed to lock PLL during USB setup");
  // Enable USB and the VBUS pad.
  native_->SetUSBCON((1 << native::USBE) | (1 << native::OTGPADE));
  // Configure USB general interrupts (handled by the USB_GEN_vect routine). We
  // want to interrupt on start of frame (SOFE), and also on end of reset
  // (EORSTE).
  native_->SetUDIEN((1 << native::EORSTE) | (1 << native::SOFE));
  // Connect internal pull-up attach resistor. This must be the final step in
  // the setup process because it indicates that the device is now ready.
  native_->SetUDCON(native_->GetUDCON() & ~(1 << native::DETACH));
  return true;
}
```

Once the device hardware is correctly set up, the threeboard informs the host of its capabilities using a series of descriptors. The device descriptor specifies fundamental information about the device (including which USB standard it supports and its unique ID). Other important descriptors follow, which inform the host of the class of the device; in this case a HID-compatible keyboard device.

All USB communication is host-centric. This means that no data can be sent from the device to the host unless the host requests it. For this reason, the threeboard's USB stack is interrupt-driven. The atmega32u4 provides hardware support to issue software interrupts identifying the beginning of a USB message frame, which is the USB host potentially requesting information from the device, such as keypress data or specific descriptor information, for example, to determine what strings to display as the name of the USB device on the host computer. The threeboard's USB interrupts are facilitated by the `UsbInterruptHandlerDelegate`, which allows the `Native` layer to pass messages to the USB stack without dependency issues, as discussed in the delegation section.

Once the USB stack has correctly configured, the threeboard's firmware enters the event loop. Individual layers can decide when to flush their state to the USB host (using `Layer::FlushToHost()`) based on their received keypress events. This triggers one or more `UsbImpl::SendKeypress()` calls, which attempts to send an individual key press to the host over USB:

```
bool UsbImpl::SendKeypress() {
  uint8_t intr_state = native_->GetSREG();
  native_->DisableInterrupts();
  uint8_t initial_frame_num = native_->GetUDFNUML();
```

```
  while (true) {
    native_->SetUENUM(kKeyboardEndpoint);
    native_->EnableInterrupts();
    // Check if we're allowed to push data into the FIFO. If we are, we can
    // immediately break and begin transmitting.
    if (native_->GetUEINTX() & (1 << native::RWAL)) {
      break;
    }
    native_->SetSREG(intr_state);
    // Ensure the device is still configured.
    RETURN_IF_ERROR(hid_state_.configuration);
    // Only continue polling RWAL for 50 frames (50ms on our full-speed bus).
    if ((native_->GetUDFNUML() - initial_frame_num) >= kFrameTimeout) {
      return false;
    }
    intr_state = native_->GetSREG();
    native_->DisableInterrupts();
  }

  SendHidState();
  native_->SetSREG(intr_state);
  return true;
}
```

Storage

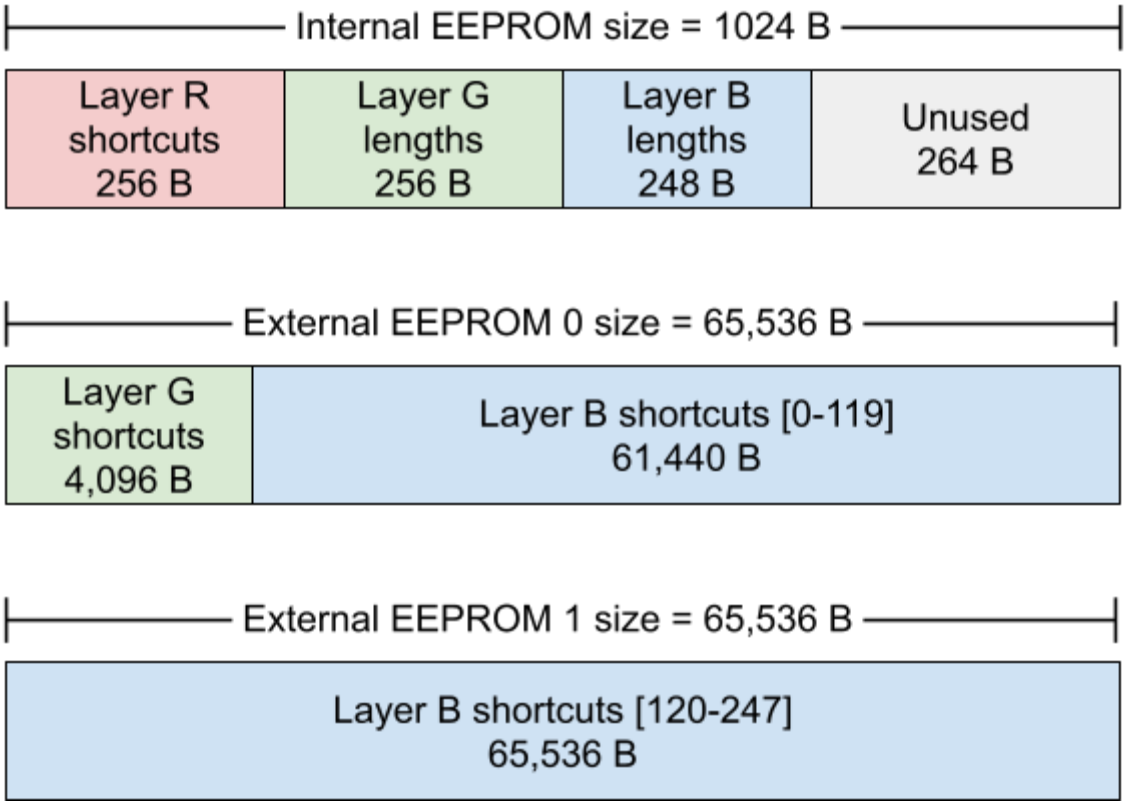
The threeboard is equipped with three EEPROM storage devices: One 1 KB EEPROM built into the atmega32u4 MCU, and two 512 kbit external EEPROMs connected to the MCU via the MCU's TWI (two-wire interface) bus. These communicate with the MCU using the I2C protocol.

In firmware, the data in these storage devices are all controlled on a high level by the `StorageController`, which is used by each of the programmable `Layers` for their respective storage needs. The `StorageController` uses the `I2C` module to abstract away the interfacing logic between the MCU and relevant EEPROMs.

The internal 1 KB EEPROM is used to store all of the character shortcuts for Layer R, in addition to the lengths of each of the shortcuts stored in layers G and B. The first external EEPROM (referred to as EEPROM 0) stores each of the word shortcuts for Layer G, along with the first 120 blob shortcuts for Layer B. The second external EEPROM (EEPROM 1) stores the remaining 128 shortcuts Layer B shortcuts.

Because Layer B (the blob shortcut layer) allows storage of per-character USB modifier codes, these must be stored in EEPROM along with each keycode. This means that each 256-character blob shortcut requires 512 bytes to store.

The storage layout is visualised below:



Style guide

A number of high-level style decisions have been made to make the code as uniform and as readable as possible:

1. All source files must be formatted with clang-format using the included `.clang-format` config file.
2. Class and member function names use `UpperCamelCase` style. Member variables use `trailing_snake_case_` style, and local variables use `regular_snake_case`.
3. Raw pointers should be used to inject dependencies or for output parameters. They must be non-owning and non-null. Raw pointers are used over references purely because they exhibit pointer semantics, making interactions with them more explicit. It follows that non-const references are not allowed.
4. Single-argument constructors must be marked `explicit`, or commented to indicate why implicit construction is favourable.
5. Relative include paths are not allowed.
6. Every class definition should have a comment explaining the function of the class. Each method and member declaration should also have comments unless trivial.

Testing strategy

The threeboard firmware is tested using three different testing levels:

1. Unit tests: Simple test cases for individual classes. These tests generally mock all of the dependencies of the class under test and set expectations on how the mocks will be used by the class. The threeboard firmware is written with this testability in mind, and most classes take pointers to abstract classes as their dependencies in their

constructors. These abstract classes generally have two implementations: a ‘real’ implementation and a mock. Mock instances will be passed to the class during testing.

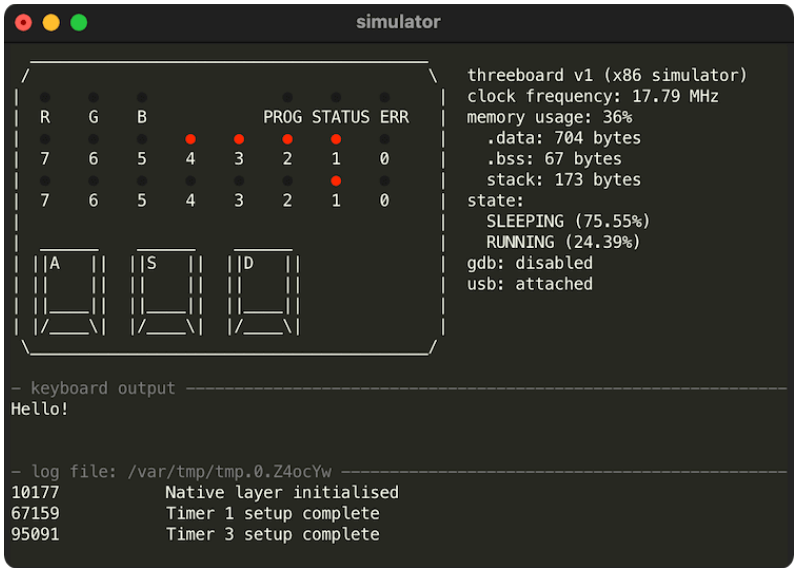
2. Simulated integration tests: This test suite uses the full simulated firmware (using the threeboard simulator) to run tests against the entire threeboard firmware end-to-end. These tests apply a sequence of keypresses to the simulated firmware and make expectations on USB interactions, LED states and character output.
3. Simulated property tests: The property test suite uses a similar setup to the integration tests in that tests are run end-to-end on the simulated threeboard firmware. This test executes keypresses on the simulator, and also on a threeboard model, used to represent the desired behaviour of the system under test. The test generates a random sequence of 25,000 input keypresses, executes them one at a time, and asserts the property that the simulator and the threeboard model must share the same state before and after the keypress is processed.

All tests run within the Google Test framework for testing and mocking.

SIMULATOR

Introduction

Debugging embedded firmware is generally difficult and slow, as target hardware limitations usually prevent meaningful logging or the ability to connect a debugger and use real-time software breakpoints. Developing embedded firmware without access to a simulator means all firmware needs to be flashed and run on the target hardware, which becomes cumbersome.



For these reasons, the threeboard project includes a fully functional terminal-based threeboard simulator for Linux and macOS, built on top of the simavr project. The simulator enables running and debugging of the threeboard AVR firmware on the same x86 platform used for development, which greatly improves development speed and makes debugging far easier and more accessible. Most importantly, the firmware being simulated is the same firmware binary that runs on physical hardware: no simulator-specific code paths exist in the firmware, although UART-based logging is disabled when building for physical hardware.

Instructions on how to compile and run the simulator are available in the simulator build instructions section of the appendix.

Features

The simulator runs a terminal-based graphical emulator of the threeboard hardware, designed to look as similar as possible to the actual threeboard hardware, with the LEDs and key switches laid out almost identically.

Other simulator features include:

- GDB debugging support: The simulator exposes a gdbserver to enable debugging of the firmware being simulated. Disabled by default, when enabled (by using the shortcut key g) the server runs locally on port 1234.
- Mock USB host: To simulate communication between the threeboard and a USB host (a computer), the simulator includes a mocked USB host which appears to the simulated firmware as a real USB host computer. The firmware will send keypresses over USB to the mocked host just as it would to a real host, and the simulator outputs these keypresses to the simulator UI in the “keyboard output” section of the UI.
- Logging: The threeboard simulator includes a mock data receiver capable of receiving

logging information from the simulated threeboard firmware via the simulated hardware's UART pins. `LOG()` and `LOG_ONCE()` macros in the firmware produce logs that are transmitted to this mock data receiver in the simulator. These macros are only enabled when building the firmware for the simulator, because the threeboard firmware doesn't correctly initialise the UART pins or clock, it takes advantage of simavr's UART implementation to transmit data instantaneously from the firmware to the simulator.

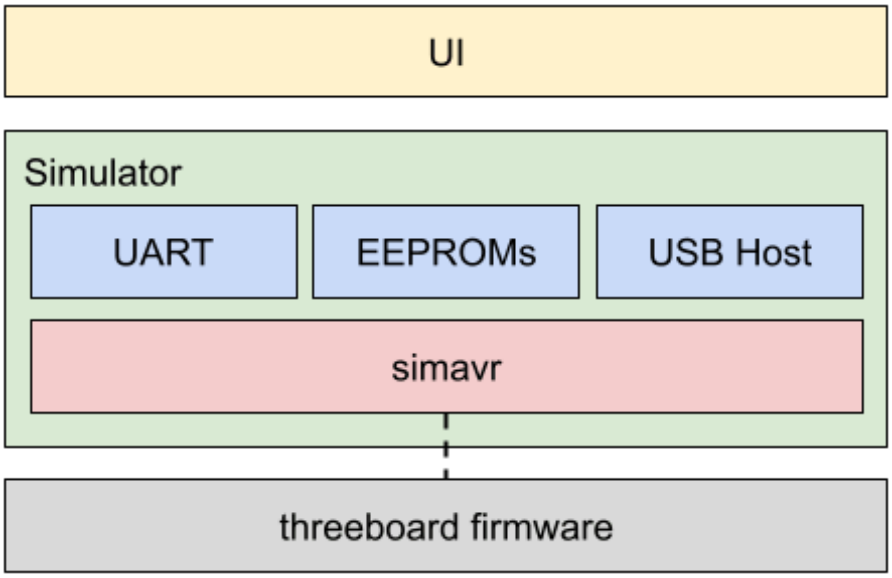
- MCU state visualisation: Several high-level states are used by simavr to categorize the MCU's current operating mode. Examples include `RUNNING`, `SLEEPING` and `CRASHED`. The threeboard simulator includes a visualisation of the amount of time spent in each state while running the simulator, which is useful when trying to optimise power usage or detect infinite or expensive loops in firmware.
- Memory usage indicator: The simulator includes a simple memory usage indicator, which displays both static memory usage (from `.data` and `.bss` segments) and dynamic memory usage. Since the threeboard hardware constraints forbid dynamic memory allocation, dynamic memory usage is an indication of stack use.
- Mock LED permanence: The simulator's terminal-based UI displays the threeboard's LEDs in red (on) or grey (off). The simulator attempts to emulate the LED matrix permanence property discussed in the LED hardware section, by setting a minimum time an LED may be lit for. This makes the simulated LEDs in the terminal behave as similarly as possible to the real LEDs on the threeboard, and makes it possible to spot timing bugs or other issues with the LEDs raster scanning logic.

Design

The simulator is effectively a wrapper around simavr, an exceptionally good AVR simulator that supports lots of features of many AVR microcontrollers, and is easy to configure with new MCUs and features. Simavr does most of the difficult work of simulating the firmware: given the threeboard firmware binary file, simavr parses it, determines the properties of the MCU being simulated based on headers in the binary, and can execute the instructions one cycle at a time. It supports interrupts and timing, and has full support for the atmega32u4's USB controller.

Events are issued and handled in simavr using IOCTL calls. These allow a way of triggering arbitrary functionality and message passing within simavr, which enables its configurability.

The simulator is comprised of several interoperating modules which emulate the properties of different hardware components on the threeboard. Modules are needed for external components (EEPROMs and UART) and for emulating a USB host machine to send keypress information.

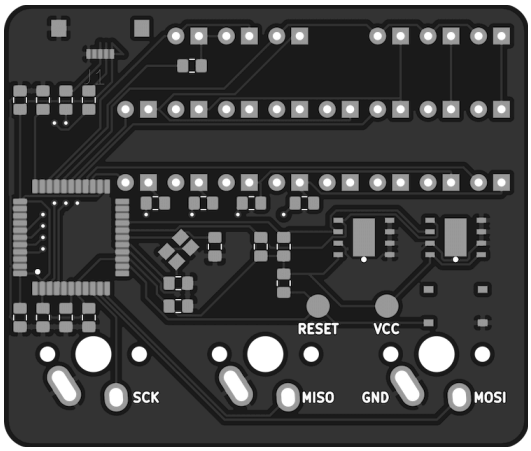


The UI module is responsible for translating the state of the simulator into a visualisation of the result of that state on the threeboard's hardware, in particular its LEDs. It's a purely text-based user interface implemented using ncurses. The static outlines and frames of the UI are drawn once, and the dynamic components are redrawn at a rate of 200Hz.

The USB host component is far from a standards-compliant host implementation. It contains the least amount of logic required to make the threeboard firmware believe it is communicating with a real USB host. It simply polls the USB controller in firmware by triggering USB general interrupts which cause the firmware to periodically send its HID state over USB.

HARDWARE

Introduction



The threeboard's hardware is a custom-designed PCB. It includes all electrical components needed to operate the threeboard, as well as test points that can be used to flash new firmware to the MCU. The majority of the components used are surface mounted, with through-holes used for some of the larger components.

This section of the document describes the design goals of the threeboard hardware, the choice of components, and electrical design decisions.

Hardware design goals

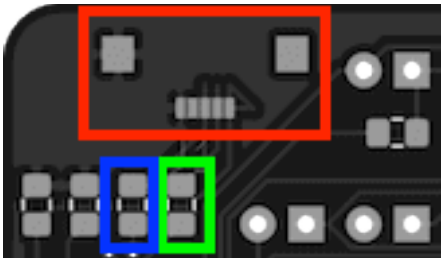
Similar to the design principles of the firmware, the threeboard's hardware was designed with several goals in mind:

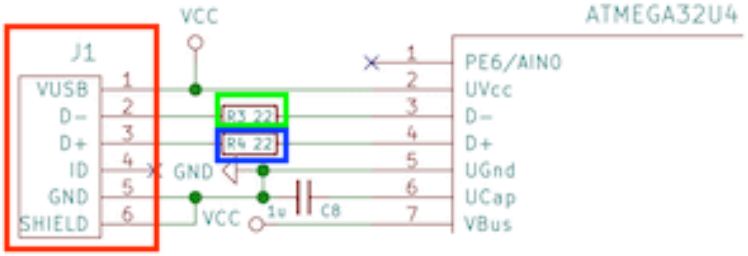
1. Simplicity: This implies that nothing should exist on the PCB that isn't completely necessary for the operation of the threeboard, and any hardware that can be offloaded to the MCU's hardware (such as making use of the internal pull-up resistors on certain pins) should be. More importantly, simplicity also applies to the hardware design patterns used; it should be possible to learn from the hardware design of the threeboard with little or no electrical engineering experience.
2. Hand-solderable: It should be possible to solder all components onto the threeboard by hand, with only basic soldering experience. This improves the accessibility of the hardware so it can be easily assembled. For example, all resistor and capacitor pads use an SMD 0805 package size. Also, the atmega32u4 MCU used by the threeboard is the 10x10mm TQFP package, despite the MCU being offered in a smaller 7x7mm QFN package, as it would be more difficult to solder.
3. Inexpensive to manufacture: No unnecessarily expensive components should be used, and no PCB features should be included that are difficult (and expensive) to manufacture unless absolutely necessary. For this reason, the threeboard PCB only uses two layers, minimal through-hole components, and as few via holes as possible.

Design

This section is split into subsections that explain the design of each major hardware component. In many cases, images of the threeboard's PCB and electrical schematic are used to aid the explanation. In these cases, the image of the relevant piece of the PCB will always be on the left, and the relevant part of the schematic on the right.

USB

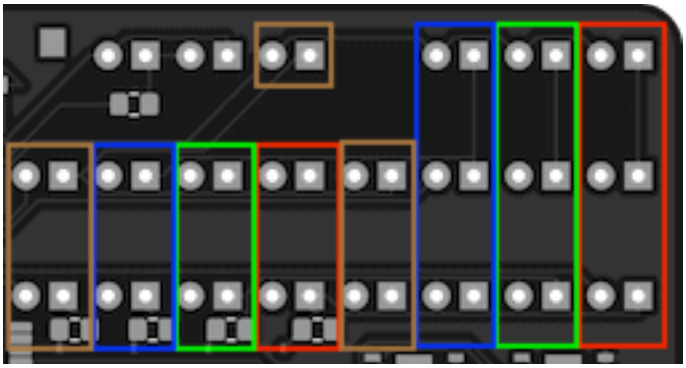


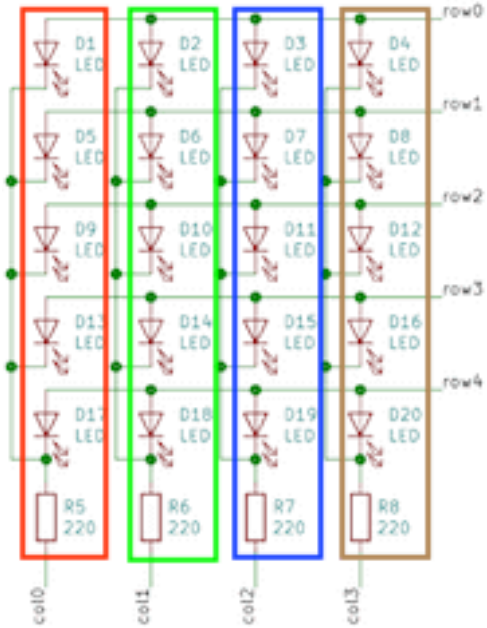


The threeboard's USB hardware is very simple because the atmega32u4 has dedicated USB pins that map directly to the USB connector (pictured in red). The only additional hardware required is specified by the atmega32u4 datasheet, sections 2.2.8 and 2.2.9, which states that the D+ and D- USB data upstream ports should be connected to the USB data connector pins via serial 22Ω resistors (pictured in blue and green).

The complexity of the USB integration lies in the firmware. It's responsible for keeping track of bus timing, issuing interrupts, and parsing and sending messages. This is discussed in detail in the firmware section's USB stack subsection.

LEDs



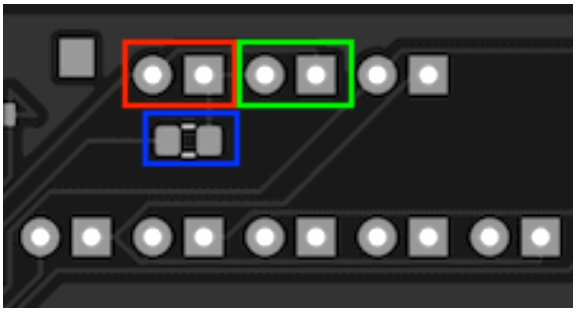


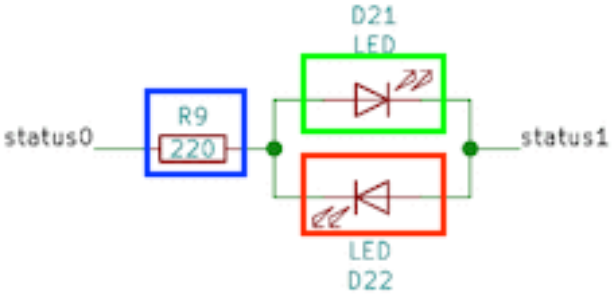
In order to be able to drive 22 LEDs using only 11 MCU pins, the majority of the threeboard's LEDs are wired in a multiplexed matrix, where each individual LED in the matrix is addressable by a row and column value. In this configuration, all of the LEDs can't be lit at the same time. Instead, each row of LEDs is scanned over in a raster scanning fashion, and individual LEDs that should be lit are turned on until the next raster scan turns them off.

This row scanning happens within the firmware and is triggered every 2ms by timer interrupt 1. Given that there are 5 rows, a full refresh takes 10ms, which gives the threeboard's LED indicators a refresh rate of 100Hz, which is well above the threshold for perceivable flicker by the human eye.

Only one LED from a column is lit at any one time, since only one LED from each column exists in a given row. Each column of LEDs is protected by one 220Ω current-limiting resistor in series

to restrict the current to each LED to 22mA (calculated using Ohm's law given the 5V input voltage from USB).

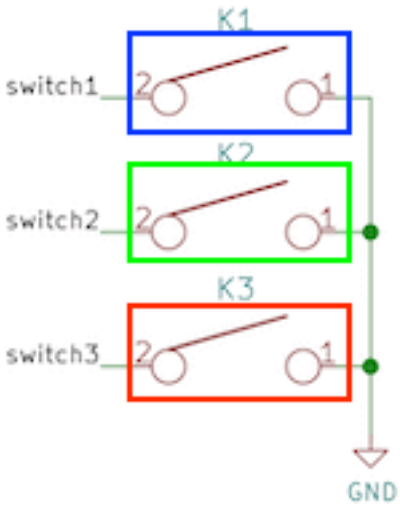




There are two LEDs on the threeboard that are not included in the multiplexed LED matrix. These are the STATUS (shown in green) and ERR (shown in red) LEDs. They're configured in parallel with their polarities inverted so that only one can be lit at any given time, and so that only one resistor and two MCU pins are required to drive them. These LEDs are configured in this way for two reasons:

1. 22 LEDs are hard to arrange in a row-column matrix since the number 22 has very few factors. The only configurations would be a 1x22 matrix (which isn't a matrix) or a 2x11 matrix (which doesn't reduce the use of LEDs or pins very much). So it's more economical to put 20 LEDs in a 5x4 matrix and the remaining two in a standalone pair.
2. It's beneficial when reporting errors or debugging issues with the firmware on the physical board (rather than the simulator) to have some LEDs which can be decoupled from the raster scanning loop. This means that if there are firmware issues with the loop logic, or issues with the MCU's timer configuration, the STATUS or ERR LEDs can be lit independently of the matrix to help with debugging or surfacing errors.

Mechanical key switches



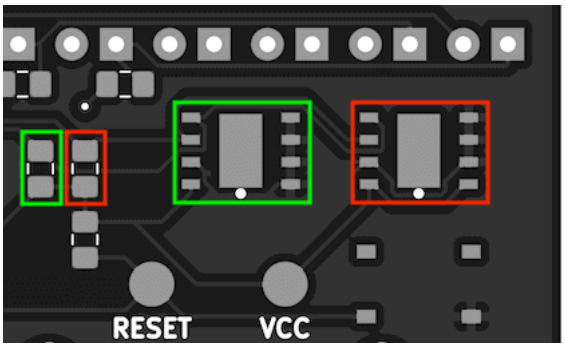
The threeboard uses Cherry MX SPST-NO (single pole, single throw, normally open) mechanical switches for its physical key switches. The bounce time of these switches differs depending on the type of switch and the date of manufacture, but it's safe to assume that it will be below 5ms. Most modern Cherry MX key switches have a 1ms bounce time.

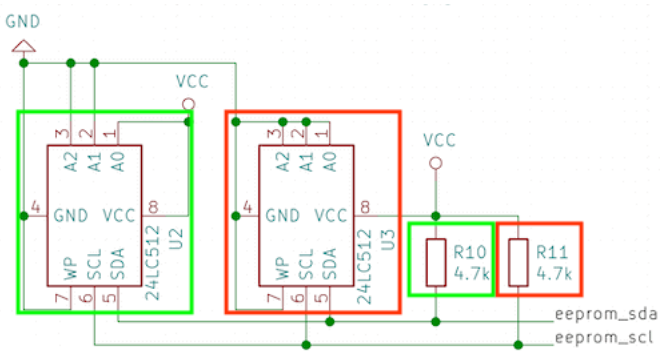
These switches are wired directly to ground, which means they are configured to be active low;

when pressed, they connect their MCU pin to ground. To avoid the situation where an unpressed switch leaves its corresponding MCU pin floating, each switch defaults to high using a pull-up resistor. The atmega32u4 includes internal pull-up resistors on a number of its pin ports, so all three switches are wired to pins on PORTB which includes internal pull-up resistors as specified by the atmega32u4 datasheet, section 2.2.3, with typical resistances of 20-50kΩ.

The MCU pins used for the key switches were chosen carefully to correspond to the pins used for `SCK` (shown in red), `MISO` (shown in green), and `MOSI` (shown in blue) of the atmega32u4's SPI interface. These pins are necessary when flashing new firmware onto the MCU using SPI, and so wiring them to the legs of the key switches removes the need for adding dedicated testing pads to connect these pins to an SPI programmer during flashing.

External EEPROMs

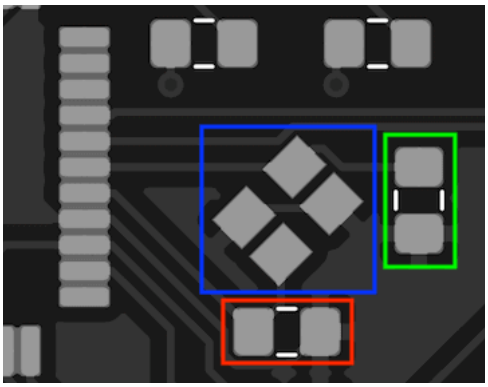


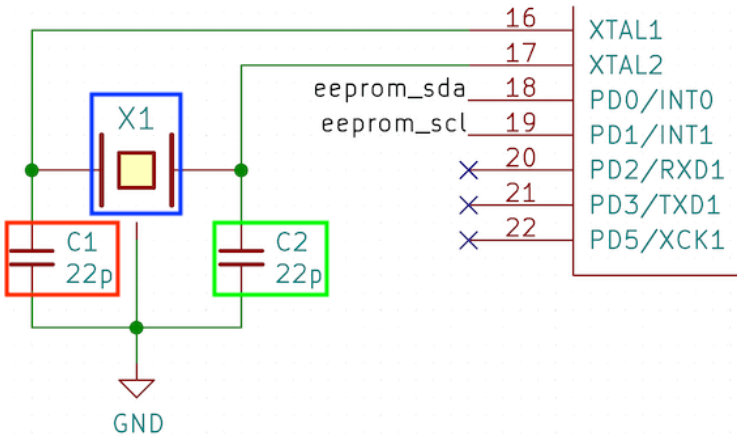


The threeboard contains two external 512 kbit EEPROM devices. These devices communicate with the MCU using its 2-wire serial interface (the I2C protocol). The atmega32u4 MCU has built-in hardware support for this interface, with a dedicated data pin (SDA) and clock pin (SCL), which collectively form the TWI (two-wire interface) bus. This interface is described in detail in the atmega32u4 datasheet, section 20.

The EEPROM devices can both be connected in parallel to these pins, as the I2C protocol uses 7-bit addresses, allowing for up to 128 devices on the same bus. Two 4.7kΩ pull-up resistors are used to pull each bus line high when not driven low by the open-drain interface, as mentioned in the atmega32u4 datasheet, section 20.2: *The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines*.

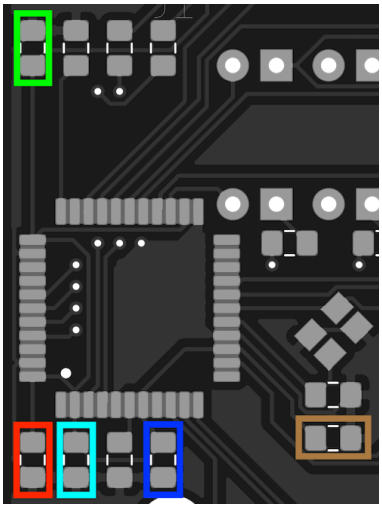
External quartz crystal clock

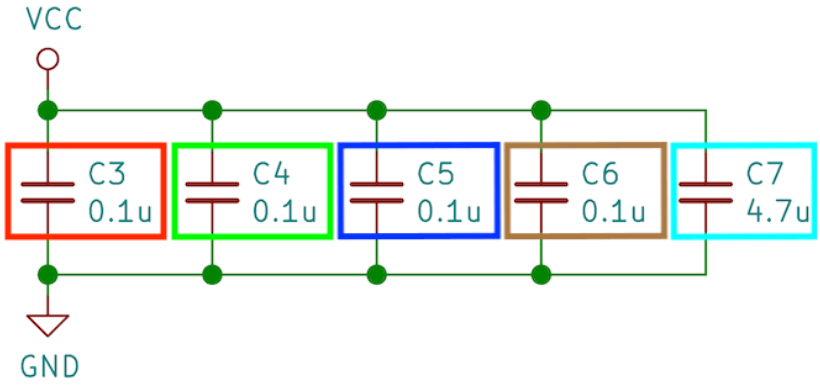




The threeboard includes a 16MHz quartz crystal oscillator (shown above in blue) as a clock source to the MCU. The MCU contains dedicated hardware pins for use with the external clock (XTAL1 and XTAL2), and the atmega32u4 datasheet, section 6.3, recommends the crystal to be wired as shown in the diagram above, with external capacitors within a 12pF - 22pF range (shown in red and green).

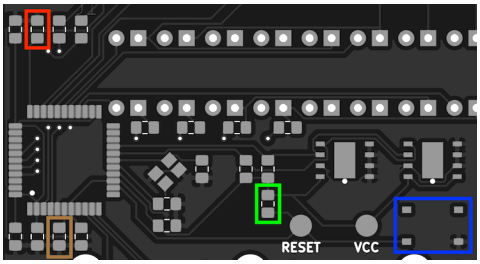
Decoupling capacitors

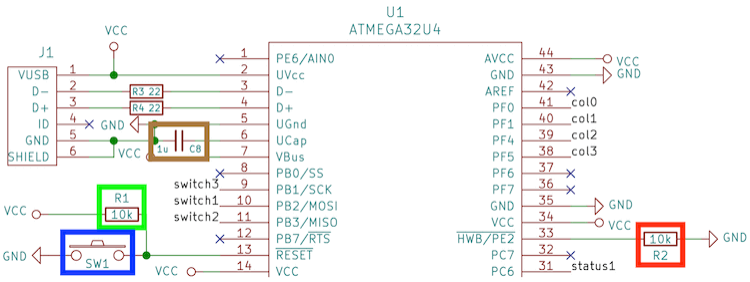




Decoupling capacitors are positioned close to each VCC-connected pin on the MCU, to reduce the effect of voltage spikes and drops from the USB power supply. They are also necessary to facilitate instantaneous current increases that may be required by the MCU, as various actions performed by the MCU may have different current requirements.

MCU

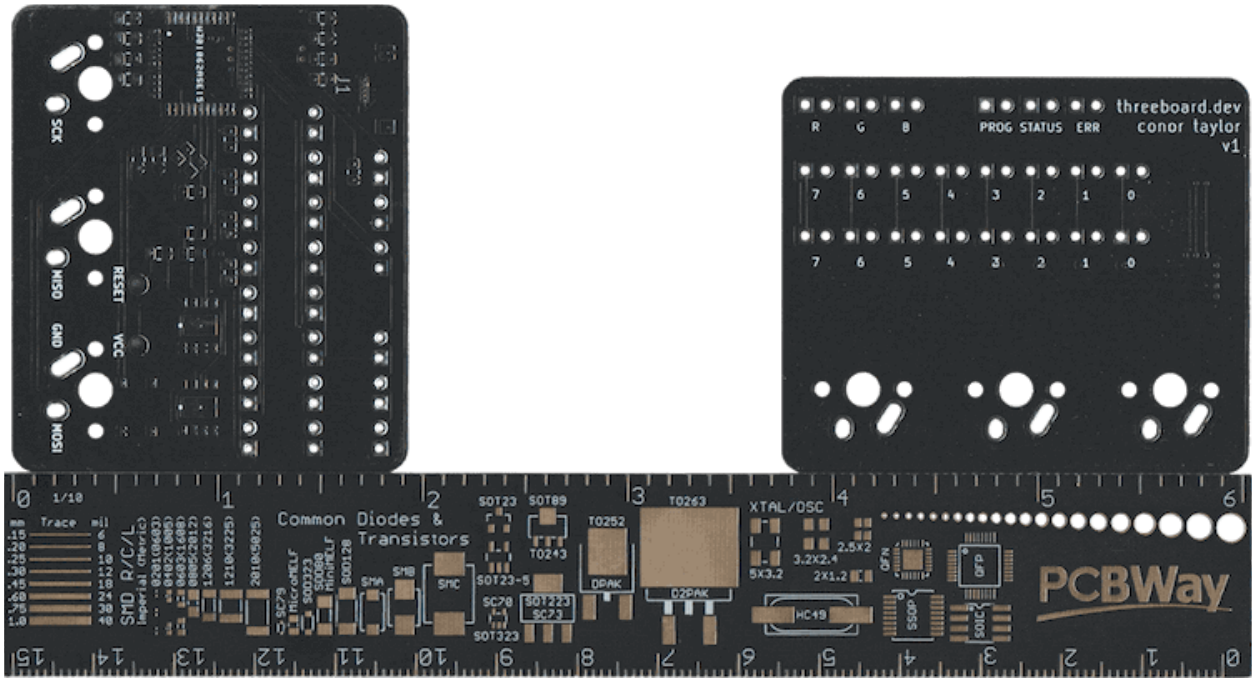




The threeboard includes a reset push button to help flash firmware to the MCU during development. The reset switch (shown above in blue) is connected to the RESET pin in the MCU, pulled up to VCC with a 10kΩ pull-up resistor (shown in green). Additionally, the HWB pin is forced low to instruct the MCU to execute the USB bootloader on reset, which allows flashing new firmware on reset. To force it low, it's tied to ground using a 10kΩ pull-down resistor (shown in red) to prevent the pin from floating.

As specified in the atmega32u4 datasheet, section 2.2.12, the UCap pin must be connected to a 1µF capacitor to regulate the USB output supply voltage.

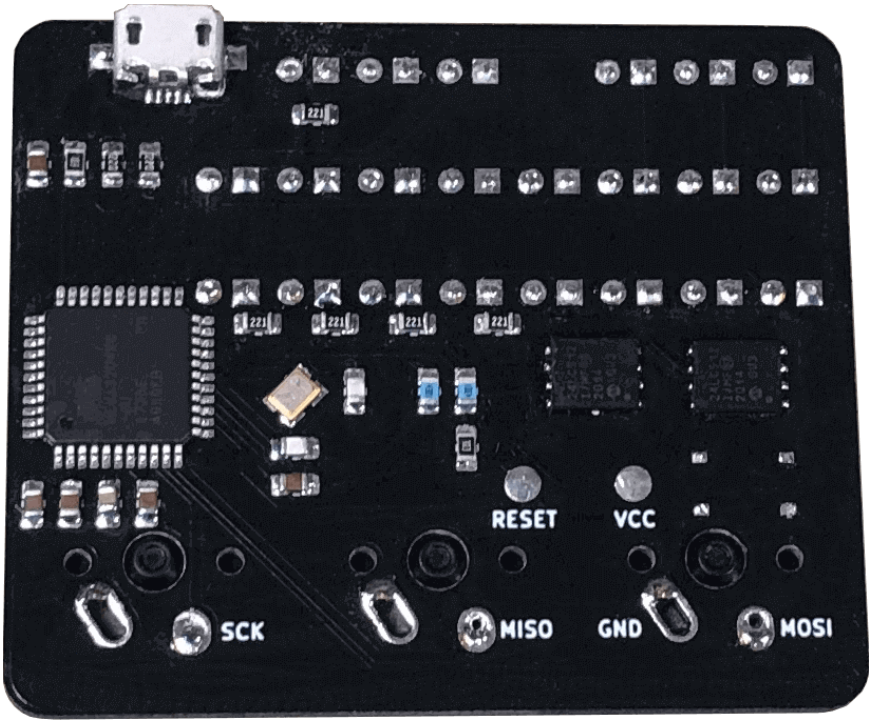
Manufacturing

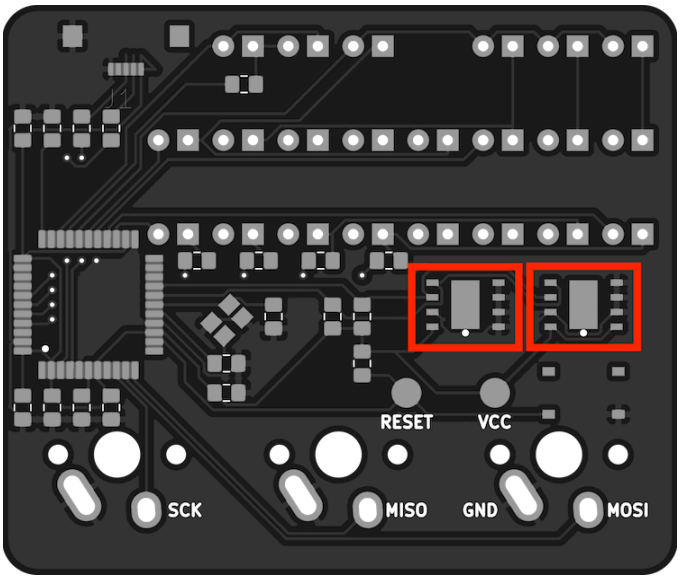


The threeboard project repository contains all of the files needed to manufacture a threeboard PCB, in the hardware subdirectory. These files can be opened using KiCad, a common open-source PCB design tool. The PCBs must be manufactured by a professional PCB manufacturing company. I recommend PCBWay as I've had great experiences using them to manufacture each iteration of the threeboard development boards. To make manufacturing easier, the threeboard repository contains a prebuilt `gerber.zip` package which is all that's needed to be provided to a PCB manufacturer such as PCBWay to manufacture threeboard PCBs.

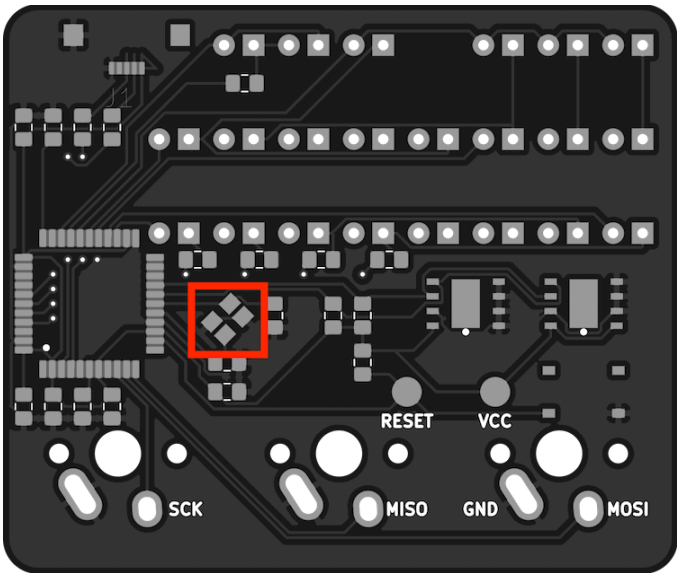
Soldering

The components required to assemble the threeboard are all listed in the component list in the appendix. These all need to be soldered to the threeboard's PCB, and the microcontroller flashed with the threeboard firmware before it's usable as a keyboard. All components can be soldered by hand and without a microscope, and although the two external EEPROMs and quartz crystal are much easier to solder using a hot air gun, it's possible to solder everything using just a soldering iron.

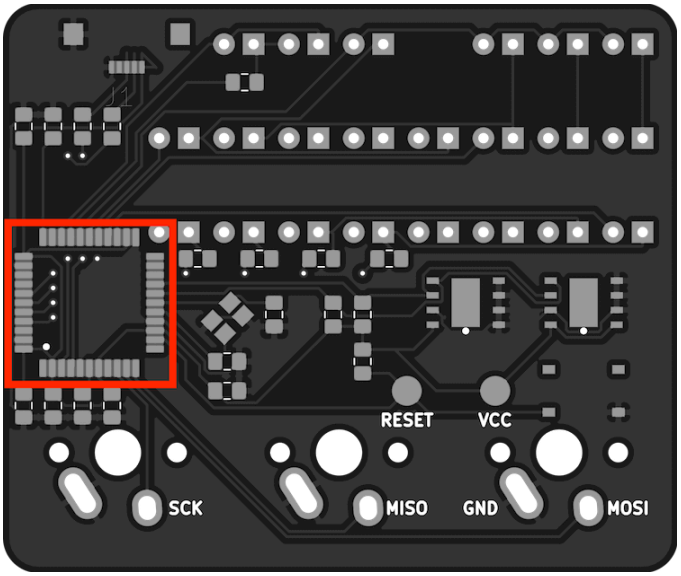




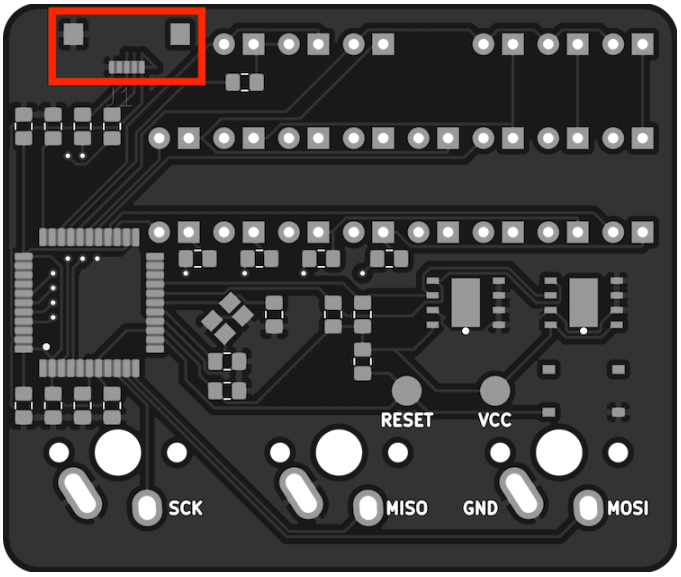
The two 24LC512-I/MF EEPROM devices should be soldered first as they're the trickiest to hand-solder with a soldering iron. If they're being soldered with a hot air gun, soldering them first prevents other nearby components from being damaged or displaced by the hot air. Solder the EEPROMs into the highlighted footprints, oriented so that their visual positioning indicators are positioned in the lower right corner.



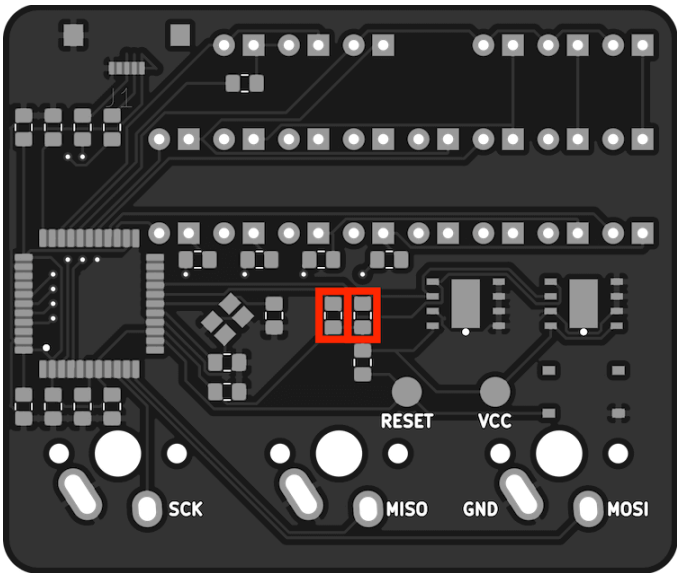
The quartz crystal should be soldered next as it's also tricky to hand-solder, and can also be hot air soldered.



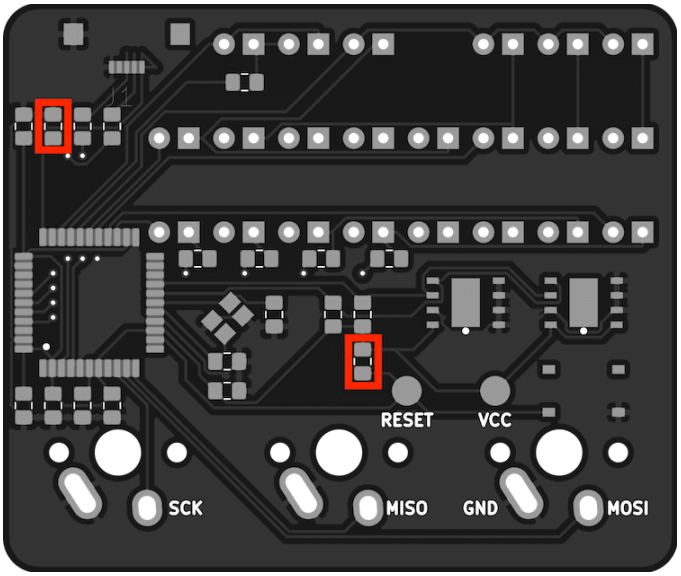
The ATMEGA32U4-AUR MCU should be soldered in the highlighted area, with its visual positioning indicator in the bottom left corner.



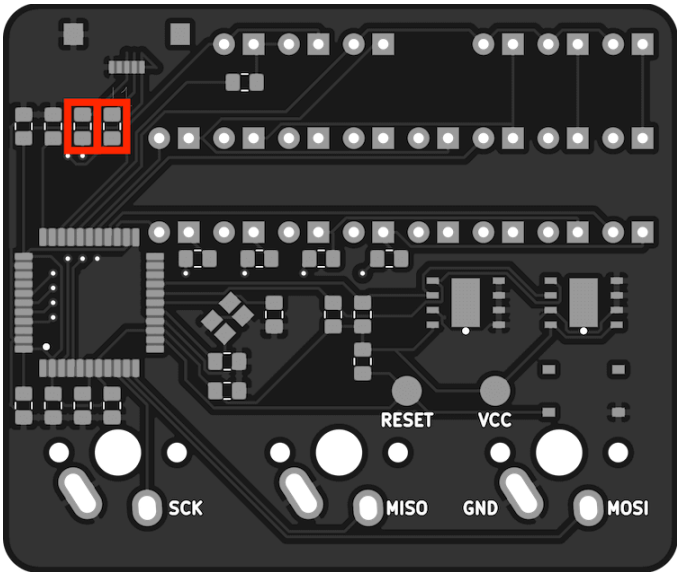
Solder the USB3070-30-A USB socket in the highlighted area. The two larger pads on either side are not connected to anything, and just add stability to the connector, which otherwise is quite fragile. These pads can be soldered or glued. I recommend using a small amount of glue between these two pads to secure the connector. The 5 small USB pins can then be soldered once the connector is aligned and secure.



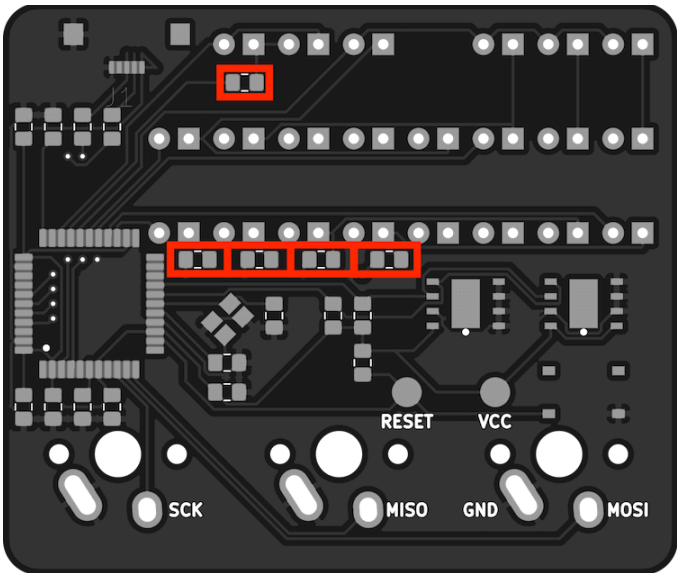
Solder the two RMCF0805FT4K70 4.7kΩ resistors – the pull-up resistors for the EEPROMs – in the highlighted areas.



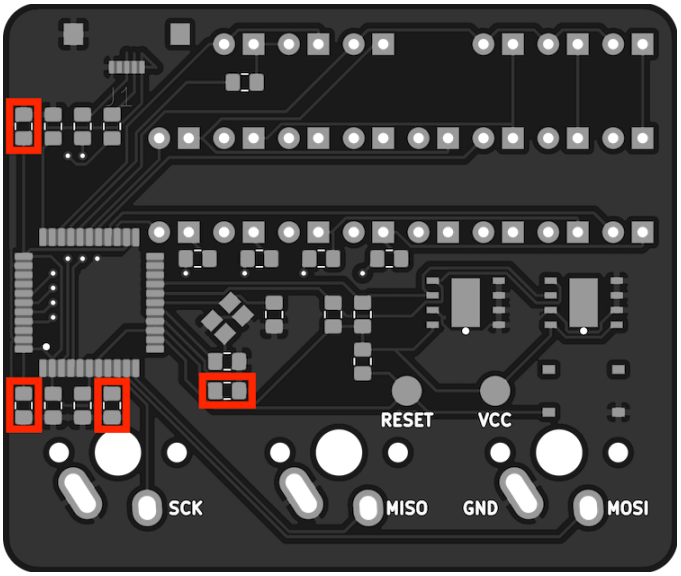
The two RMCF0805JT10K0 10kΩ resistors are used as a pull-up resistor for the MCUs RESET pin and a pull-down resistor to inform the MCU to check for USB activity during boot. They should be soldered in the highlighted areas.



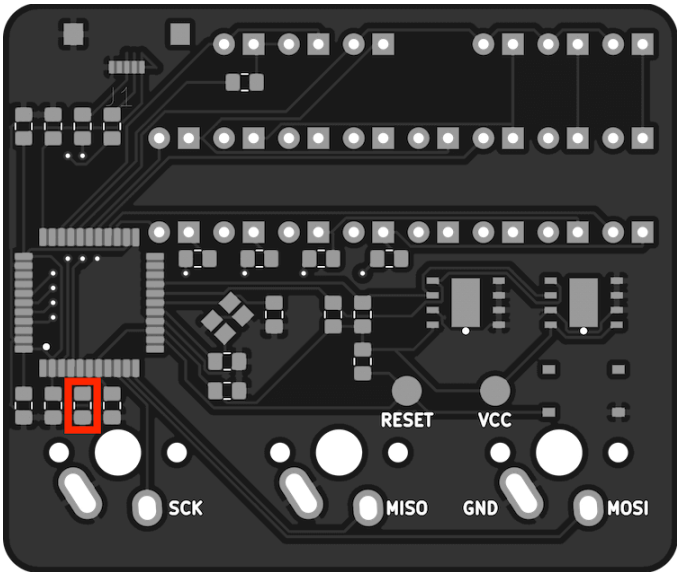
Solder the two RMCF0805JT22R0 22Ω resistors in the highlighted areas. These are serially connected between the USB data lines and the MCU.



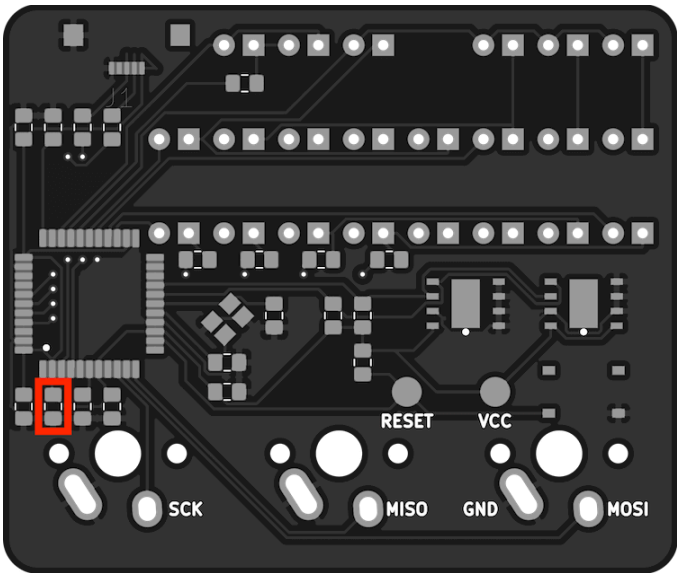
There are five 220Ω resistors needed to protect the LED banks. They should be soldered in the highlighted areas.



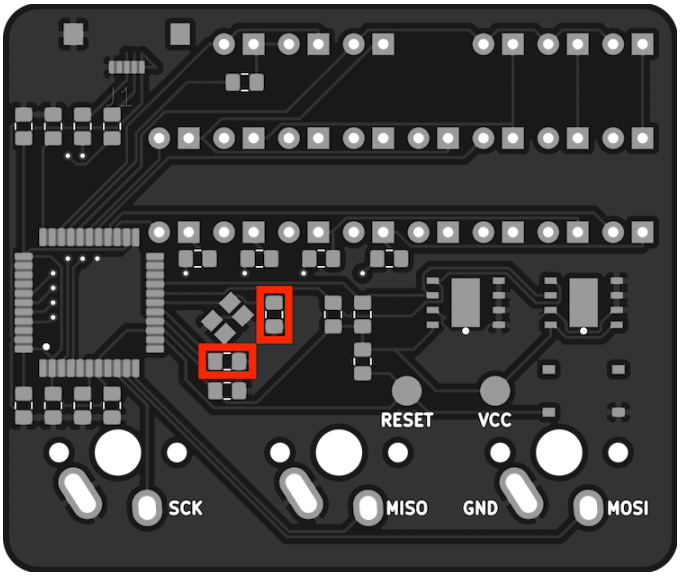
Solder the four CC0805ZRY5V9BB104 0.1uF decoupling capacitors in the areas highlighted.



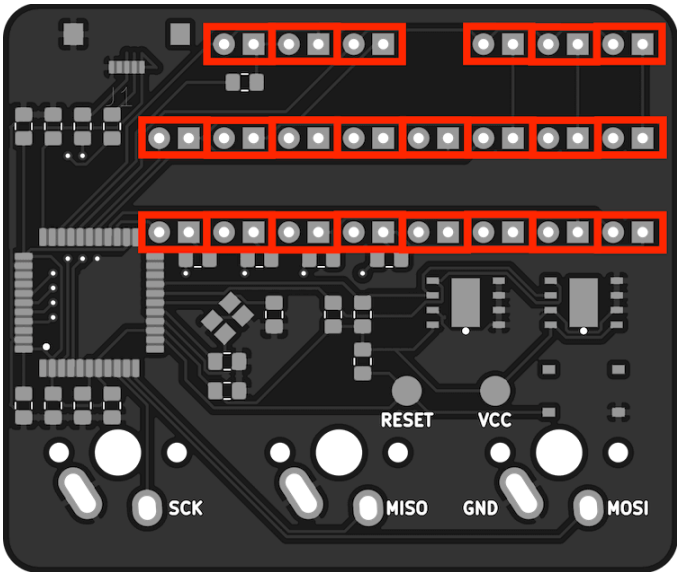
A 1uF capacitor is needed to regulate the USB output supply voltage. Solder the CC0805KKX7R7BB105 capacitor in the highlighted area.



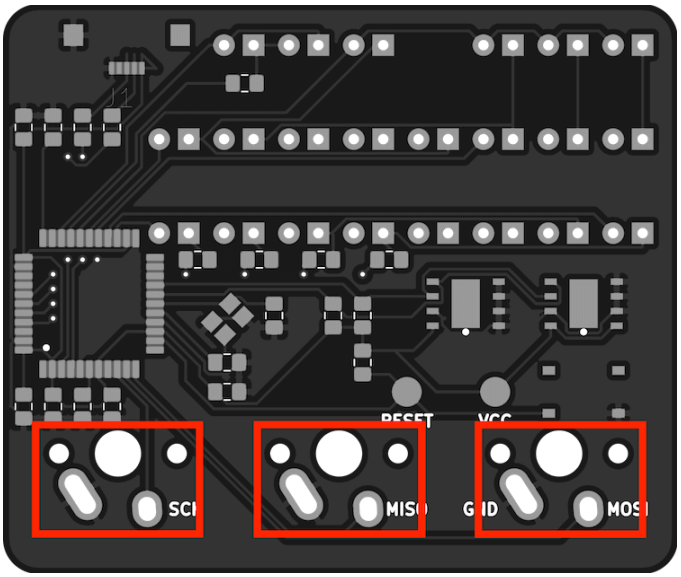
Solder the CC0805ZRY5V6BB475 4.7µF decoupling capacitor in the highlighted area.



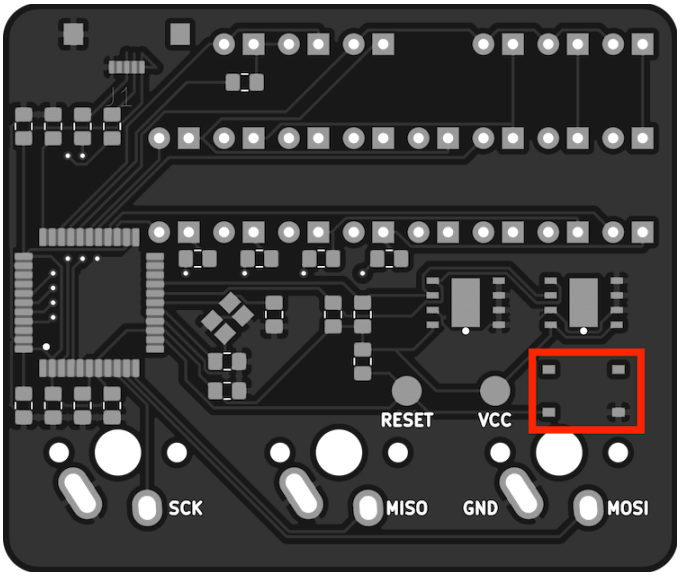
The two C0805C220J5GACTU 22pF external clock crystal capacitors should be soldered as shown.



Solder the 22 LTL-1CHE LEDs in the locations shown. Make sure to wire the cathode of the LED to the square through-hole.



Solder the three Cherry MX keyswitches as shown.



If needed, a PTS526 SK08 SMTR2 LFS push-button RESET switch can be soldered in the highlighted area. This is only needed when flashing threeboard firmware to the threeboard using a USB bootloader. It may only be needed once, or not needed at all if flashing using SPI.

Flashing firmware

A freshly constructed threeboard will need to have a bootloader installed before it's possible to flash the threeboard firmware. For simplicity, I suggest installing the Arduino USB bootloader using an Arduino as an ISP programmer and following this guide. The necessary SPI pins are all labelled on the threeboard: `RESET`, `VCC`, `SCK`, `MISO`, `GND` and `MOSI`. These should be connected to the corresponding pins on the ISP programmer to flash the bootloader.

A threeboard hex file is required to flash the firmware to the device. This is produced by running `bazel build //src:threeboard_hex` from the `threeboard/firmware` directory. Once the threeboard hex file has been successfully built, it can be flashed to the device using the following command:

```
avrdude -v -patmega32u4 -cavr109 -P /dev/cu.usbmodemXXX -b57600 -D
-Uflash:w:bazel-bin/src/threeboard_hex.hex:i
```

APPENDIX

threeboard usage table

Layer	Mode	Key combo	Action	LED byte 1	LED byte 2	
					4 MSB	4 LSB
DFLT	DFLT	**X**	`keycode++`	`keycode`	`modcode`	
		Y	`modcode++`			
		Z	Send `(keycode, modcode)` over USB			
		XY				
		XZ	Clear `keycode`			
		YZ	Clear `modcode`			
		XYZ	Go to layer `R`			
R	DFLT	**X**	`shortcut_id++`	`shortcut_id`	`modcode`	
		Y	`modcode++`			
		Z	Send `(shortcut[shortcut_id], mod_code)` over USB			
		XY	Enter `PROG` mode			
		XZ	Clear `shortcut_id`			
		YZ	Clear `modcode`			
		XYZ	Go to layer `G`			
	PROG	**X**	`shortcut[shortcut_id]++`	`shortcut[shortcut_id]`	`shortcut_id`	
		Y	`shortcut_id++`			
		Z				
		XY				
		XZ	Clear `shortcut[shortcut_id]`			
		YZ	Clear `shortcut_id`			
		XYZ	Exit `PROG` mode			
G	DFLT	**X**	`shortcut_id++`	`shortcut_id`	`shortcut[shortcut_id].length()`	`word_mod_code`
		Y	`word_mod_code++`			
		Z	Send `(shortcut[shortcut_id], word_mod_code)` over USB			
		XY	Enter `PROG` mode			
		XZ	Clear `shortcut_id`			
		YZ	Clear `word_mod_code`			
		XYZ	Go to layer `B`			

	PROG	**X**	`key_code++`	`key_code`	`shortcut[shortcut_id].length()`
		Y			
		Z	Append `key_code`		
		XY			
		XZ	Clear `key_code`		
		YZ	Clear shortcut at `shortcut_id`		
		XYZ	Exit `PROG` mode		
B	DFLT	**X**	`shortcut_id++`	`shortcut_id`	`shortcut[shortcut_id].length()`
		Y			
		Z	Send `(shortcut[shortcut_id])` over USB		
		XY	Enter `PROG` mode		
		XZ	Clear `shortcut_id`		
		YZ			
		XYZ	Go to layer `DFLT`		
	PROG	**X**	`key_code++`	`keycode`	`modcode`
		Y	`mod_code++`		
		Z	Commit `(key_code, mod_code)`		
		XY	Delete shortcut at `shortcut_id`		
		XZ	Clear `key_code`		
		YZ	Clear `mod_code`		
		XYZ	Exit `PROG` mode		

Component list

ATMEGA32U4-AUR	Atmega32u4 8-bit microcontroller in 44 pin TQFP (10mm x 10mm) package.
C0805C220J5GACTU	22pF ±5% 50V Ceramic Capacitor, SMD 0805.
CC0805ZRY5V9BB104	0.1µF -20%, +80% 50V Ceramic Capacitor, SMD 0805.
CC0805KKX7R7BB105	1µF ±10% 16V Ceramic Capacitor, SMD 0805.
CC0805ZRY5V6BB475	4.7µF -20%, +80% 10V Ceramic Capacitor, SMD 0805.
RMCF0805JT22R0	22Ω 5% 1/8W resistor, SMD 0805.
RMCF0805JT220R	220Ω 5% 1/8W resistor, SMD 0805.
RMCF0805FT4K70	4.7kΩ 1% 1/8W 0805.
RMCF0805JT10K0	10kΩ 5% 1/8W resistor, SMD 0805.
24LC512-I/MF	512KBIT EEPROM I2C 8DFN.
USB3070-30-A	Micro USB B female socket.
PTS526 SK08 SMTR2 LFS	Tactile Switch SPST-NO Top Actuated Surface Mount, 5.2mm x 5.2mm.
LTL-1CHE	Diffused red LED.
MX1A-E1NW	Cherry MX “Blue” mechanical key switch SPST-NO 0.01A 12V.
FA-238 16.0000MB-C3	16MHz ±50ppm Crystal 18pF 80 Ohms 4-SMD.

Firmware build instructions

1. Install Bazel using the instructions for your OS/distro as described in the Bazel installation guide.
2. Install the packages that the threeboard firmware depends on:

   ```
   sudo apt-get install build-essential gcc-avr avr-libc git
   ```

3. If you don't already have a C++17 compatible compiler, install GCC version 8:

   ```
   sudo apt-get install g++8
   ```

4. Clone the threeboard git repository:

   ```
   git clone git@github.com:taylorconor/threeboard.git
   ```

5. Navigate to the Bazel workspace root of the threeboard project:

   ```
   cd threeboard/firmware
   ```

6. Compile the threeboard firmware into a `.hex` file that can be flashed to hardware:

   ```
   bazel build //src:threeboard_hex
   ```

   The firmware hex file will be written to `threeboard/firmware/bazel-bin/src/threeboard_hex.hex`.

Simulator build instructions

1. Complete all of the steps in the firmware build instructions guide if you haven't already.
2. Install additional simulator dependencies:

   ```
   sudo apt-get install libelf-dev libncursesw5-dev freeglut3-dev
   ```

3. Clone the custom simavr fork, which includes additional threeboard compatibility functionality:

   ```
   git clone git@github.com:taylorconor/simavr.git
   ```

4. Check out the threeboard compatibility branch:

   ```
   cd simavr && git checkout threeboard_compat
   ```

5. Install simavr:

   ```
   sudo make install RELEASE=1
   ```

6. Navigate to the `threeboard/firmware` directory of your threeboard project clone.
7. Run the threeboard simulator:

   ```
   cd simavr && git checkout threeboard_compat
   ```

How does a USB keyboard work?

This section describes on a very high level how most USB 2.0 keyboards interact with their connected computer (the *host*) via USB, from connection to device setup to keypress handling. It is not intended to be a definitive explanation, just an overview. For more detailed explanations, I recommend the USB in a NutShell series of articles, parts of which have been included in this section. The full 650 page USB 2.0 specification can also be useful as a reference.

Overview

USB stands for Universal Serial Bus, a name which identifies the three main properties of USB:

- Universal: Designed to be able to support any kind of peripheral device.
- Serial: There is only one stream of data being sent over the wire at a time (as opposed to a *parallel bus* that supports multiple concurrent streams).
- Bus: A network topology in which all of the various devices in the network are connected to a single shared data line.

USB 2.0 is a single host, multiple device protocol. There must be exactly one host device coordinating all communication on the wire. In most cases, the host is a PC or laptop, and the devices are any connected peripherals like keyboards, mice, webcams etc. Each device is assigned a unique 7-bit address when it first connects to the host (a process known as *device enumeration*), so up to 127 devices can be connected to a single USB bus at a time. Most computers will have multiple USB busses, e.g. one per external USB port.

The USB protocol defines several data transfer modes. Which mode a device chooses depends on the intended use of the device:

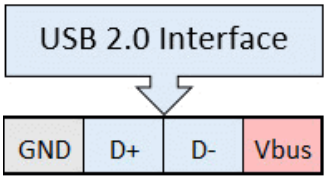
- Control: Used for device enumeration and setup, for example for informing the host about the name and manufacturer of the device, and for selecting which other mode the device will use as its main mode of operation.
- Interrupt: Non-periodic, low-latency, low data throughput communication. An Interrupt request is queued by the device until the host polls it to ask for data. This is typically used for rare but time-sensitive events, such as mouse clicks or keyboard keypresses.
- Bulk: Used for large, potentially bursty data transfers. Bulk transfers provide error correction on the payload, and error detection/re-transmission mechanisms, ensuring data is transmitted and received without error. This transfer mode is commonly used in USB memory sticks.
- Isochronous: This allows a device to reserve a defined amount of bandwidth with guaranteed latency, and perform periodic transfers. This is useful in audio or video applications where congestion may cause loss of data or framerate, such as webcams.

USB keyboards spend most of their time transferring data in the interrupt mode because the host needs to react quickly to keypress events to avoid perceivable lag.

The USB protocol identifies the functionality of a device using *device classes*, defined by a class code sent to the host. This allows the host to e.g. load drivers for the device and to support new devices from different manufacturers, making USB universal. The USB standard specifies some standard device classes, but manufacturers can implement their own using the wildcard

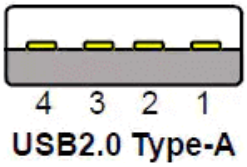
"vendor-specific" class type. One standard-defined class is the Human Interface Device (HID) class; this class includes devices intended to be used by humans to interact with a computer, such as keyboards and mice. The threeboard implements this USB HID class to identify itself as a keyboard.

Connector wiring

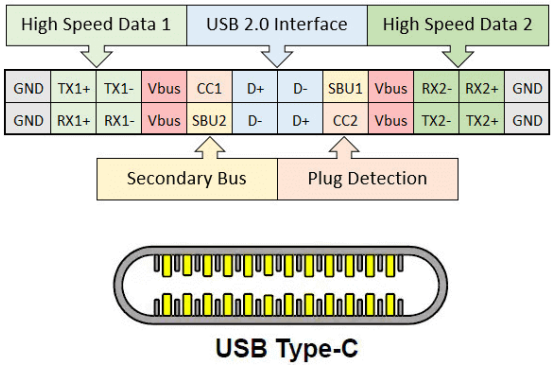


The wiring of a USB 2.0-only connector (i.e. a connector capable of up to a 480 Mbit/s "high speed" data rate) is quite simple, with only 4 pins needed:

1. V_{BUS}: 5V power line.
2. D-: Part of differential pair data line.
3. D+: Part of differential pair data line.
4. Ground.



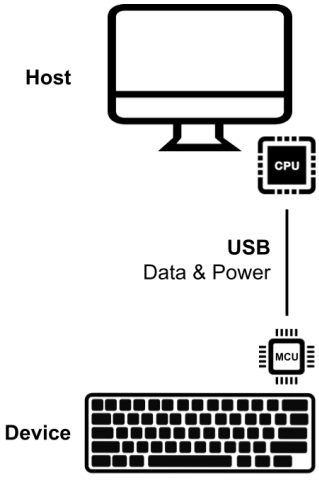
USB 3.0 requires an additional 5 pins for a total of 9, but keeps the same connector types. Sometimes these have blue inserts to indicate USB 3.0 compatibility, but retain backwards compatibility with USB 2.0. Type A connectors are only used on hosts (e.g. computers), and Type B connectors are only used on devices (e.g. phones, keyboards), but they have the same wiring and pin configuration. Newer Type C connectors (often called "USB-C") are omnidirectional and are the only connector type that supports USB 4.0. These connectors are radically different, with 24 pins, but still maintain backwards compatibility with all previous USB standards. This is possible because USB-C contains the 4-pins USB 2.0 configuration as a subset of its 24 pins.



The *differential pair* data line is an implementation of differential signalling. Two wires are used but the data line remains serial.

The remainder of this section assumes the USB 2.0 standard. Although as discussed, keyboards implementing USB 2.0 can still have USB-C connectors by using its USB 2.0 subset.

Hardware and firmware

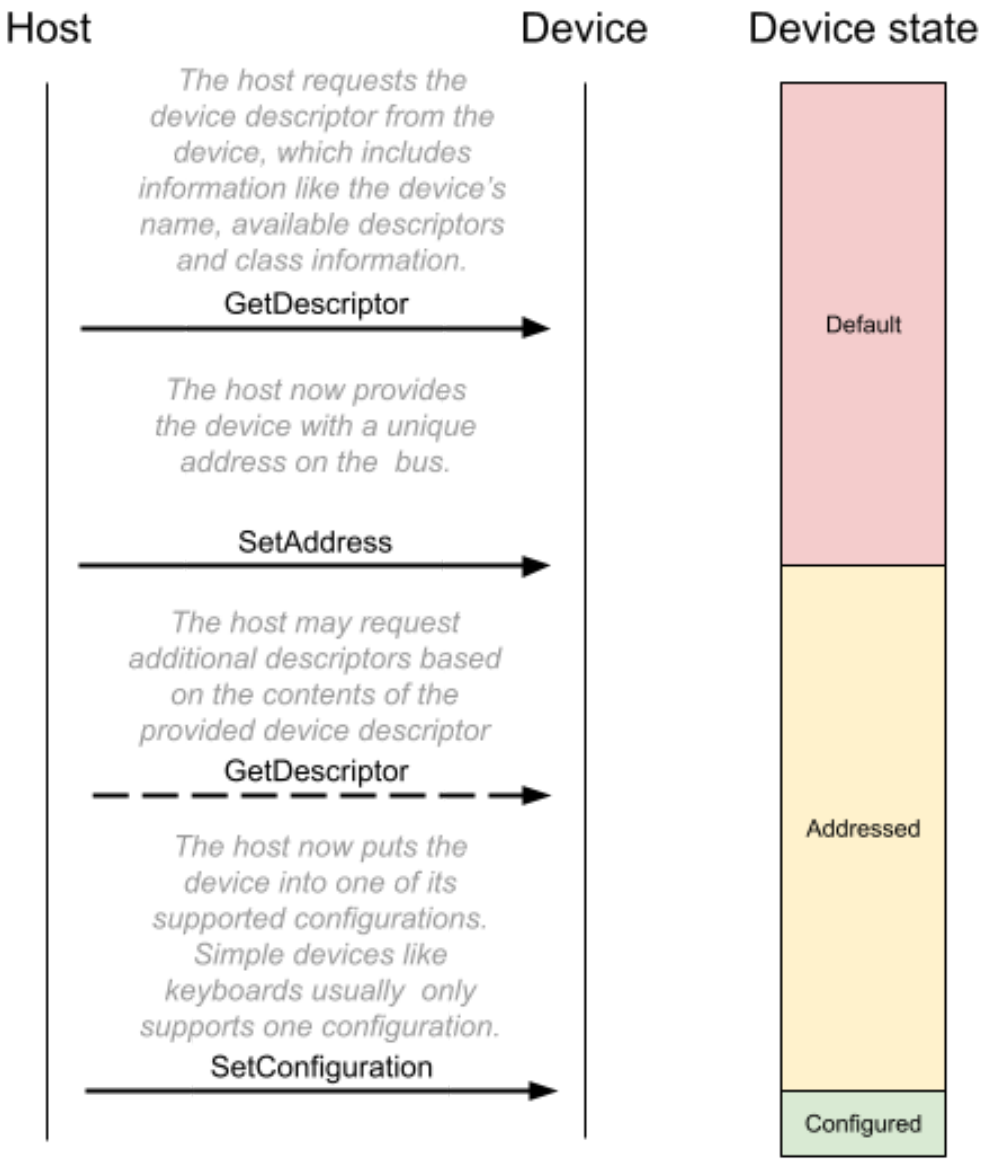


It's important to think of a USB keyboard as a computer in its own right. To be able to fully implement the USB protocol, keyboards generally contain a 5V microcontroller with clock speeds up to 80MHz, as much as 100 KB of RAM, and hardware support for the USB protocol. These microcontrollers use RISC instruction sets, such as ARM or AVR. Firmware runs on the keyboards microcontroller, implementing everything from checking for keypresses and lighting status LEDs to implementing the USB protocol and communicating with the host. QMK is a popular ARM and AVR-compatible open-source keyboard firmware. The threeboard uses its own custom-built firmware, available in the threeboard repository.

Device enumeration

USB device enumeration is the process where hosts detect, identify and in some cases load drivers for a USB device. This section won't go into any of the electrical details of how hosts detect when a device has been plugged in, instead it will focus on how devices use the USB protocol to identify themselves to the host. However one important electrical feature of this process is how the host determines the speed of the device; the USB D+ is connected to V_{BUS} with a pull-up resistor, indicating full-speed 12 Mbps mode. Connecting USB D- instead would specify the low-speed 1.5 Mbps operating mode.

When a USB device is being enumerated, it cycles through several configuration states: *default*, *addressed* and *configured*. Once a device has been plugged in and is receiving power from the host, it enters the *default* state. In this state, it has not even been assigned an address yet. The process from *default* to *configured* (i.e. usable) state is visualised below:



The threeboard's enumeration values (such as its device descriptor and additional HID descriptors) are all specified in usb/internal/descriptors.h.

Sending keypress events

During enumeration, a USB keyboard will have provided an *endpoint descriptor* to the host. This is used to describe all of the USB *endpoints* a device supports. Endpoints are essentially isolated data buffers on the USB device, with important properties including a transfer direction (IN, meaning device to host, and OUT, meaning host to device) and a transfer type (control, interrupt, bulk, isochronous). USB devices all must provide at least one *control endpoint* (endpoint 0), which is the endpoint used during enumeration. Devices may provide multiple additional endpoints e.g. for interrupting the host with keypress data.

USB keyboards may be configured differently depending on their functionality, but in the most basic case (and in the case of the threeboard), a USB keyboard provides only one additional endpoint, an IN endpoint 1, with an interrupt transfer type. The host polls the device as frequently as every 1ms (this polling period is configurable in the endpoint descriptor) to check if the device has an interrupt in its endpoint buffer waiting to be sent to the host. A USB HID-compliant keyboard device will put a *keyboard input report* message in its IN buffer, which contains up to 6 keypress codes and a *modifier code*, identifying all of the modifier keys being pressed. The *keyboard input report* is 8 bytes long and contains one modifier keycode byte, one reserved byte, and 6 keycode bytes. A table mapping these bytes to key codes is defined in section 10 of the HID usage tables document.

An example host polling sequence is visualised below: