Keypad and Matrix Scanning in CircuitPython

Created by Dan Halbert

https://learn.adafruit.com/key-pad-matrix-scanning-in-circuitpython

Last updated on 2023-08-29 04:42:26 PM EDT

Table of Contents

Overview 3

Keys and Events 4

- One-Button Example
- Events and EventQueues
- MacroPad Example
- MacroPad HID Example

KeyMatrix 9

- Keypad Matrix Example
- Which Way are the Diodes?

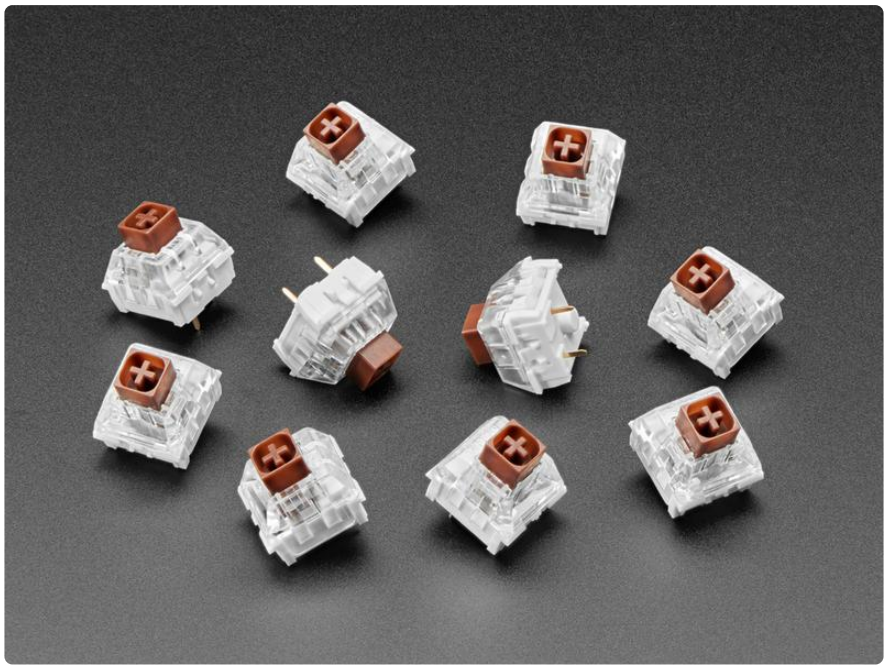
ShiftRegisterKeys 12

- PyBadge or PyGamer ShiftRegisterKeys Example
- SNES Controller Example

Advanced Features 15

- Scanning Interval and Debouncing
- Queue Size and Queue Overflow
- Avoiding Storage Allocation
- Comparing Events and Using Events as Dictionary Keys

Overview



The CircuitPython `keypad` module scans a set of one or more keys or buttons in the background, while your program is doing other things, and gives you debounced key-pressed and key-released events. The module provides three different kinds of scanners, which cover common ways of connecting keys to pins.

- `Keys`: Each key or button is connected to a separate pin.
- `KeyMatrix`: The keys are wired in a row-column matrix, with or without diodes. This is how most typing keyboards, music keyboards, and telephone-style keypads are wired.
- `ShiftRegisterKeys`: The keys are connected to an external parallel-in serial-out shift register. The keys are read by clocking data from the shift register into a single input pin.

Each time a key is pressed or released, a scanner will record an `Event` in an `EventQueue`. Each event gives the key number, and says whether the key has been pressed or released. Your program can read the events sequentially.

The background scanning does debouncing as well, so you don't have to worry about handling that yourself.

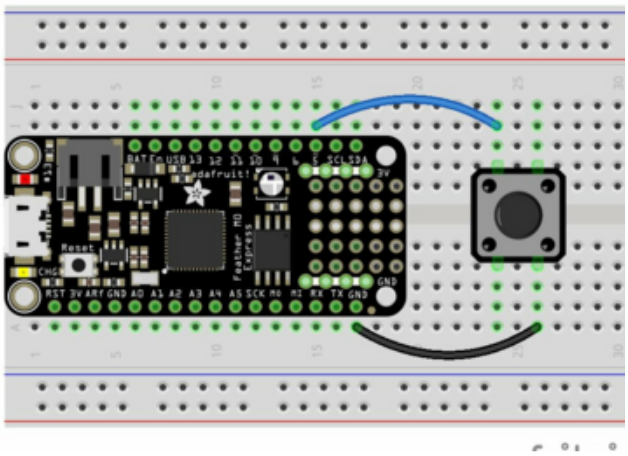
This Guide will explain how each of the scanners works, and provide some simple examples.

Keys and Events

The `Keys` scanner is the simplest. Each key or button is connected to its own pin on your microcontroller board. When you create a `Keys` scanner, you give it a sequence of pins, and say whether the pins become `True` or `False` (high or low logic value) when pressed. You also say whether internal pull-up or pull-down resistors need to be enabled on the pins to force a stable voltage level when the keys or buttons are not pressed.

One-Button Example

Here's a really simple example, with just one button:



One terminal of a button is connected to pin D5 on this Feather board, and the other terminal is grounded. D5 will need to be pulled up, because there is no external pull-up resistor. When the button is not pressed, D5 will be high (`True`), and when it is pressed, D5 will be low (`False`).

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import keypad

keys = keypad.Keys((board.D5,), value_when_pressed=False, pull=True)

while True:
    event = keys.events.get()
    # event will be None if nothing has happened.
    if event:
        print(event)
```

The program creates a `Keys` object, and passes it a tuple of one pin (D5). That pin will be `key_number` 0.

We set `value_when_pressed` to `False`, because the pin goes low when the button is pressed. We set `pull` to `True` because we haven't provided an external resistor. When `pull` is `True`, it will enable a pull-up or a pull-down appropriately:

- If `value_when_pressed` is `True`, enable an internal pull-down resistor, to hold the pin low when not pressed.
- If `value_when_pressed` is `False`, enable an internal pull-up resistor, to hold the pin high when not pressed.

When you press the button, you'll get an `Event` saying that `key_number` 0 has been pressed (see below). When you release, the button you'll get a released event. The output will look like this:

```
&lt;Event: key_number 0 pressed&gt;
&lt;Event: key_number 0 released&gt;
&lt;Event: key_number 0 pressed&gt;
&lt;Event: key_number 0 released&gt;
```

Events and EventQueues

We glossed over how events are fetched in the example above. When you press or release a key, the scanner notices that a key has changed state. It records an `Event` in the `EventQueue` attached to the scanner. You get the `EventQueue` by accessing the `events` property of the scanner. In the example above the `EventQueue` is available from `keys.events`.

To fetch the next `Event` from the `EventQueue`, you use `events.get()`. If nothing has happened (no press or release of any key) since you last asked, `events.get()` returns `None`.

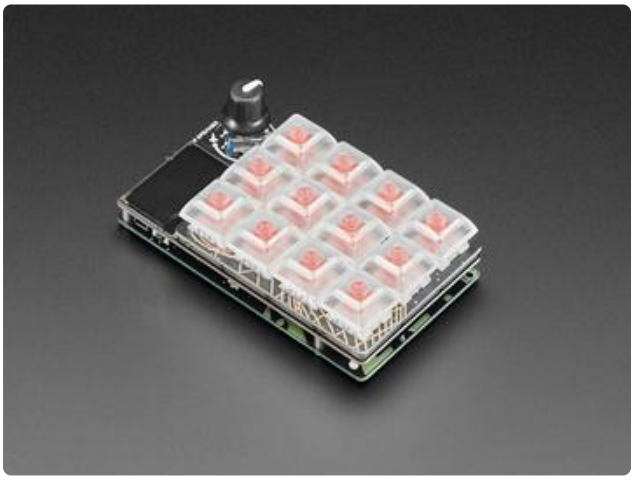
Every `Event` has four properties:

- `key_number`: the number of the key that changed. Keys are numbered starting at 0.
- `pressed`: `True` if the event is a transition from released to pressed.
- `released`: `True` if the event is a transition from pressed to released. `released` is always the opposite of `pressed`; it's provided for convenience and clarity, in case you want to test for key-release events explicitly.
- `timestamp`: The time in milliseconds when the event occurred. The timestamp is the value of `supervisor.ticks_ms()` at the time of the event. You can use

the timestamp to determine, for instance, how long a key was pressed down, or the interval between presses,

You only get a `pressed` or `released` event once per transition. If you keep holding down a key, you won't get multiple `pressed` events.

MacroPad Example



The Adafruit MacroPad has 12 keys, with one key per pin. This program illuminates a key when it's pressed, and turns off the illumination when it's released.

Note that the key pin names are `KEY1` to `KEY12`, but the `key_number` values are `0` to `11`.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import keypad
import neopixel
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keycode import Keycode

KEY_PINS = (
    board.KEY1,
    board.KEY2,
    board.KEY3,
    board.KEY4,
    board.KEY5,
    board.KEY6,
    board.KEY7,
    board.KEY8,
    board.KEY9,
    board.KEY10,
    board.KEY11,
    board.KEY12,
)

KEYCODES = (
    Keycode.SEVEN,
    Keycode.EIGHT,
    Keycode.NINE,
    Keycode.FOUR,
    Keycode.FIVE,
    Keycode.SIX,
    Keycode.ONE,
    Keycode.TWO,
```

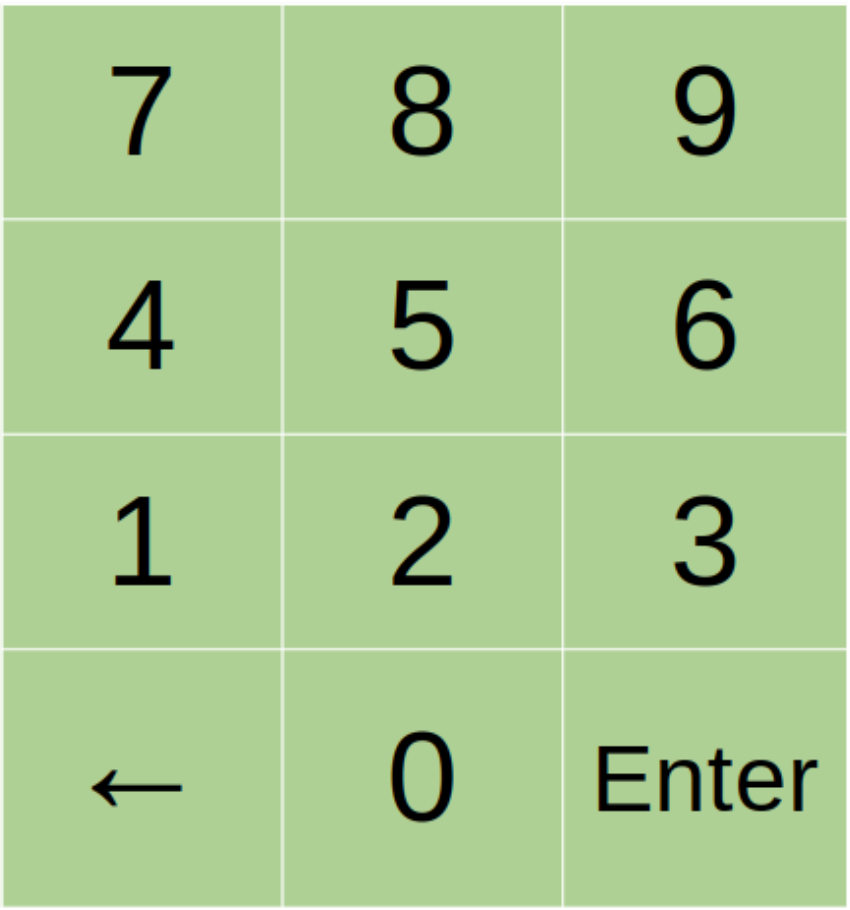
```
    Keycode.THREE,
    Keycode.BACKSPACE,
    Keycode.ZERO,
    Keycode.ENTER,
)

ON_COLOR = (0, 0, 255)
OFF_COLOR = (0, 20, 0)

keys = keypad.Keys(KEY_PINS, value_when_pressed=False, pull=True)
neopixels = neopixel.NeoPixel(board.NEOPIXEL, 12, brightness=0.4)
neopixels.fill(OFF_COLOR)
kbd = Keyboard(usb_hid.devices)

while True:
    event = keys.events.get()
    if event:
        key_number = event.key_number
        # A key transition occurred.
        if event.pressed:
            kbd.press(KEYCODES[key_number])
            neopixels[key_number] = ON_COLOR

        if event.released:
            kbd.release(KEYCODES[key_number])
            neopixels[key_number] = OFF_COLOR
```

Here's some sample output from the program. In this case I pressed down key number 0, held it down, and then pressed key 1. Then I released key 0 and then released key 1. Then I pressed key 2.

```
&lt;Event: key_number 0 pressed&gt;
&lt;Event: key_number 1 pressed&gt;
&lt;Event: key_number 0 released&gt;
&lt;Event: key_number 1 released&gt;
&lt;Event: key_number 2 pressed&gt;
&lt;Event: key_number 2 released&gt;
```

MacroPad HID Example

You can use keypad with CircuitPython's HID capability to send keypresses to a host computer. The example below is a number pad implemented on a MacroPad with backspace and enter keys.



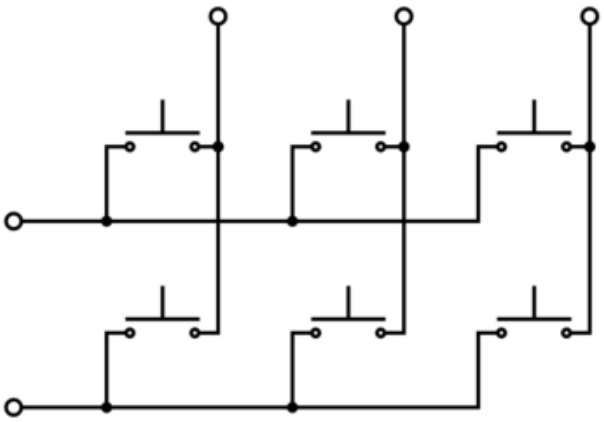

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import keypad
import neopixel
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keycode import Keycode

KEY_PINS = (
    board.KEY1,
    board.KEY2,
    board.KEY3,
    board.KEY4,
    board.KEY5,
    board.KEY6,
    board.KEY7,
    board.KEY8,
    board.KEY9,
    board.KEY10,
    board.KEY11,
    board.KEY12,
)

KEYCODES = (
    Keycode.SEVEN,
    Keycode.EIGHT,
    Keycode.NINE,
    Keycode.FOUR,
    Keycode.FIVE,
```

```
    Keycode.SIX,
    Keycode.ONE,
    Keycode.TWO,
    Keycode.THREE,
    Keycode.BACKSPACE,
    Keycode.ZERO,
    Keycode.ENTER,
)

ON_COLOR = (0, 0, 255)
OFF_COLOR = (0, 20, 0)

keys = keypad.Keys(KEY_PINS, value_when_pressed=False, pull=True)
neopixels = neopixel.NeoPixel(board.NEOPIXEL, 12, brightness=0.4)
neopixels.fill(OFF_COLOR)
kbd = Keyboard(usb_hid.devices)

while True:
    event = keys.events.get()
    if event:
        key_number = event.key_number
        # A key transition occurred.
        if event.pressed:
            kbd.press(KEYCODES[key_number])
            neopixels[key_number] = ON_COLOR

        if event.released:
            kbd.release(KEYCODES[key_number])
            neopixels[key_number] = OFF_COLOR
```

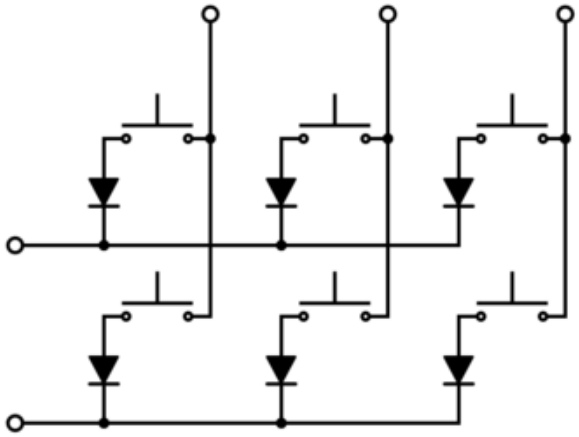
KeyMatrix

The `KeyMatrix` scanner scans keys that are wired in a row-column matrix. Each key is wired to a single row line and a single column line. Each row and column line goes to a pin. So for instance a 3x4 matrix of 12 keys will have 3 column lines and 4 row lines. We only need 7 pins (3+4) to scan this matrix instead of 12.

Optionally, there is diode attached to each key as well. This prevents false presses when you press several keys at once.



Here is a simple 2x3 matrix without diodes. There are 2 rows and 3 columns.



Here is 2x3 matrix with diodes. In this case, the columns are connected to the diode anodes, and the rows are connected to the cathodes. It could be the other way around.

Keypad Matrix Example

In this example, a `KeypadMatrix` is created and then monitored.

This example is for a 3x4 matrix, such as the Adafruit 3x4 Matrix Keypad () (pictured), the Adafruit Membrane 3x4 Keypad (), or the 3x4 Phone-style Matrix Keypad (). For details about wiring these keypads, see the Matrix Keypad Guide ().

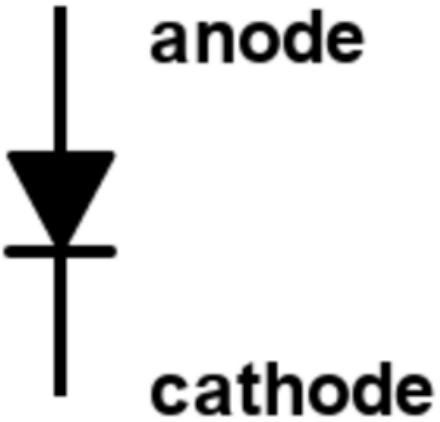
```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import keypad
import board

km = keypad.KeyMatrix(
    row_pins=(board.A0, board.A1, board.A2, board.A3),
    column_pins=(board.D0, board.D1, board.D2),
)

while True:
    event = km.events.get()
    if event:
        print(event)
```

Here is some output from this program. Remember that the key numbers start from 0, so they probably don't correspond to the labels on the keys themselves.

```
&lt;Event: key_number 0 pressed&gt;
&lt;Event: key_number 0 released&gt;
&lt;Event: key_number 1 pressed&gt;
&lt;Event: key_number 1 released&gt;
&lt;Event: key_number 3 pressed&gt;
&lt;Event: key_number 3 released&gt;
```

Which Way are the Diodes?

The 3x4 keypads above don't have any diodes, so we don't need specify diode orientation. But if you're building a typing or music keyboard, you probably have diodes, and need to specify their orientation. To do this, you set the `columns_to_anodes` parameter when you construct the `KeyMatrix`.



If the column lines are connected to the diode anodes and the rows are connected to the cathodes, set `columns_to_anodes` to `True`, which is the default. If the reverse is true, set `columns_to_anodes` to `False`.

Here's an example of specifying `columns_to_anodes` when the diode directions match the schematic at the top of this page:

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import keypad
import board

km = keypad.KeyMatrix(
    row_pins=(board.D0, board.D1, board.D2, board.D3),
    column_pins=(board.D4, board.D5, board.D6),
    columns_to_anodes=True,
)

while True:
    event = km.events.get()
    if event:
        print(event)
```

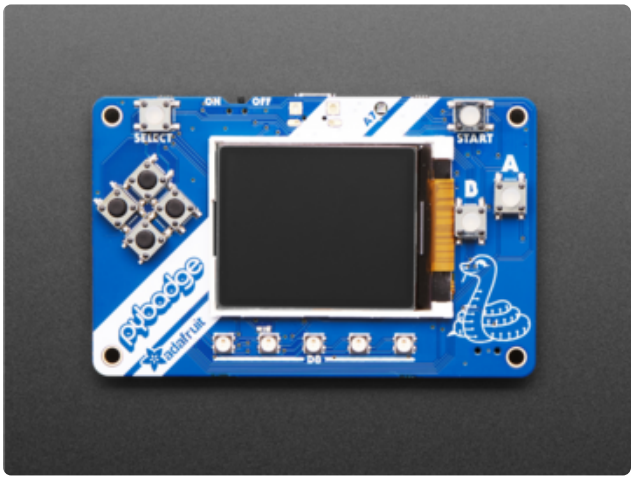
ShiftRegisterKeys

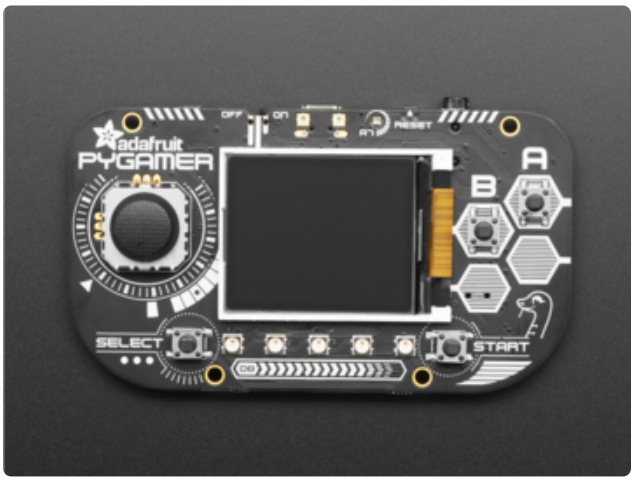
The `ShiftRegisterKeys` scanner scans keys that are attached to a parallel-in serial out shift register. The key states are loaded into the shift register all at once, and then clocked out of the shift register into an input pin that is read serially.

At this time we support basic clocked/latched shift registers such as 74x165 chips. These are very inexpensive, but require external pull-up/down resistors. We do not support shift-register-to-matrix or I2C/SPI expanders.

Several Adafruit products, such as the PyBadge () and the PyGamer (), use a shift register to monitor their buttons. Other products such as SNES game controllers also use shift registers.

PyBadge or PyGamer ShiftRegisterKeys Example



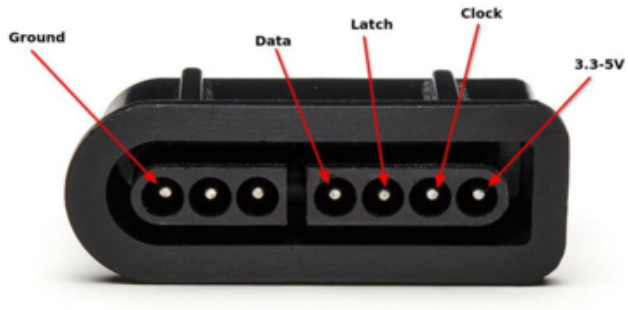


This example reads the buttons on a PyBadge or PyGamer. They both work the same way. (The joystick on the PyGamer is an analog joystick, and is read using `AnalogIn`; it can't be read with `keypad`).

We don't know how many keys are connected to the shift register. There could also be multiple shift registers chained together. So we have to specify a `key_count` in the example below, which is the total length of the shift register or registers.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import keypad
import board

k = keypad.ShiftRegisterKeys(
    clock=board.BUTTON_CLOCK,
    data=board.BUTTON_OUT,
    latch=board.BUTTON_LATCH,
    key_count=8,
    value_when_pressed=True,
)

while True:
    event = k.events.get()
    if event:
        print(event)
```

SNES Controller Example



Controllers for the venerable SNES game system also use shift registers. Adafruit stocks an SNES controller (), and they are readily available elsewhere as well. The controller is pictured on the left, with extra labelling for some of the buttons.

These controllers have 12 buttons, so they use two 8-bit shift registers chained together, with the last 4 bits ignored. So we'll set `key_count` to 12 below.

The pinout for the connector is shown on the left. Note there is a group of 3 and a group of 4 pins, and one end is rounded.

Which Way to Latch?

The PyBadge and PyGamer use 74HC165 shift registers. These shift registers latch the input when the latch pin is high (True). That's the default for

`ShiftRegisterKeys`. But the SNES controller uses CD4021 shift registers. They latch on low, so we need to specify an extra argument to specify how to latch: we set `value_to_latch` to `False` in the code below (the default is `True`).

SNES Key Names

The program below prints out the SNES key names by looking up the key numbers in the `SNES_KEY_NAMES` tuple. Key number 0 is "B", 1 is "Y", etc.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import keypad
import board

SNES_KEY_NAMES = (
    "B",
    "Y",
    "SELECT",
    "START",
    "UP",
    "DOWN",
    "LEFT",
    "RIGHT",
    "A",
    "X",
    "L",
    "R",
)

shift_k = keypad.ShiftRegisterKeys(
    clock=board.D5,
    latch=board.D6,
    value_to_latch=False,
    data=board.D7,
    key_count=12,
    value_when_pressed=False,
)

while True:
    event = shift_k.events.get()
    if event:
        print(
            SNES_KEY_NAMES[event.key_number],
            "pressed" if event.pressed else "released",
        )
```

Here's some sample output from the program above:

```
START pressed
START released
B pressed
B released
B pressed
B released
X pressed
X released
Y pressed
Y released
L pressed
```

```
R pressed
R released
R pressed
R released
L released
DOWN pressed
LEFT pressed
B pressed
DOWN released
LEFT released
B released
LEFT pressed
LEFT released
LEFT pressed
LEFT released
```

Advanced Features

Scanning Interval and Debouncing

By default, all the scanners in the `keypad` module scan their inputs at 20 millisecond intervals. This is enough time to debounce most mechanical keys, which can take quite a few milliseconds to settle down. If you have keys that need more (or less) debouncing, you can set the optional `interval` parameter when you create a scanner. The `interval` is given in floating-point seconds. For example:

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import keypad
import board

km = keypad.KeyMatrix(
    row_pins=(board.A0, board.A1, board.A2, board.A3),
    column_pins=(board.D0, board.D1, board.D2),
    # Allow 50 msecs to debounce.
    interval=0.050,
)

while True:
    event = km.events.get()
    if event:
        print(event)
```

Queue Size and Queue Overflow

The `EventQueue` for each scanner is of fixed size. The default size is 64 events. You can change this value by setting the optional `max_events` parameter:

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT
```

```
import keypad
import board

k = keypad.Keys(
    (board.D8, board.D9),
    value_when_pressed=False,
    pull=True,
    # Increase event queue size to 128 events.
    max_events=128,
)

while True:
    event = k.events.get()
    if event:
        print(event)
```

If your program doesn't keep up while reading events, the event queue can become full. New events will be discarded. If this happens, the `EventQueue.overflowed` property is set to `True` on the event queue. You can clear the event queue with the `EventQueue.clear()` method; that also sets the `overflowed` flag to `False`. Since you have lost events, you probably also want to forget the existing state of the keys and start over. you can do that by calling `reset()` on the scanner. `scanner.reset()` clears any key-pressed information, so it will immediately generate key-pressed events for any keys that were pressed at the time you reset. For example:

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import keypad
import board

k = keypad.Keys(
    (board.D8, board.D9),
    value_when_pressed=False,
    pull=True,
)

while True:
    # Check if we lost some events.
    if k.events.overflowed:
        k.events.clear()  # Empty the event queue.
        k.reset()  # Forget any existing presses. Start over.

    event = k.events.get()
    if event:
        print(event)
```

You don't have to check for queue overflow. For most programs, it will never happen, or it's not important if it happens: the user will adjust.

Avoiding Storage Allocation

`EventQueue.get()` creates a new `Event` every time it returns an event. These Events probably become unused quite quickly, and add to the "garbage" that must be collected periodically. You can avoid generating new `Event` objects each time by

using the `EventQueue.get_into(event)` method, which writes into an existing `Event`. It returns `True` if it wrote into the `Event`, and `False` otherwise (when `EventQueue.get()` would have returned `None`).

Here's an example of using `get_into()`:

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board
import keypad

keys = keypad.Keys((board.D8,), value_when_pressed=False, pull=True)

# Create an event we will reuse over and over.
event = keypad.Event()

while True:
    if keys.events.get_into(event):
        print(event)
```

Comparing Events and Using Events as Dictionary Keys

Event implements `__eq__()` and `__hash__()`, so you can test if two Events are equivalent, and you can store them in a dictionary. For example, here's a program that translates button pushes on pins D8 and D9 into the strings `"LEFT"` and `"RIGHT"`. And if you push the button on D10, the program stops.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import board

from keypad import Keys, Event

keys = Keys((board.D8, board.D9, board.D10), value_when_pressed=False, pull=True)

LEFT_EVENT = Event(0, True)  # Button 0 (D8) pressed
RIGHT_EVENT = Event(1, True)  # Button 1 (D9) pressed
STOP_EVENT = Event(2, True)  # Button 2 (D10) pressed

DIRECTION = {
    LEFT_EVENT: "LEFT",
    RIGHT_EVENT: "RIGHT",
}

while True:
    event = keys.events.get()
    if event:
        if event == STOP_EVENT:
            print("stop")
            break

        # Look up the event. If not found, direction is None.
        direction = DIRECTION.get(event)
```

```
        if direction:
            print(direction)
```