KMK FW - Sequences

Sequences are used for sending multiple keystrokes in a single action, and can be used for things like Unicode characters (even emojis! 🇨🇦), *Lorem ipsum* generators, triggering side effects (think lighting, speakers, microcontroller-optimized cryptocurrency miners, whatever). If you are still unsure of what this is, most other vendors call these "Macros", but can do much more if you wish.

Sending strings

The most basic sequence is `send_string`. It can be used to send any standard English alphabet character, and an assortment of other "standard" keyboard keys (return, space, exclamation points, etc.)

```
from kmk.handlers.sequences import send_string

WOW = send_string("Wow, KMK is awesome!")

keyboard.keymap = [<other keycodes>, WOW, <other keycodes>]
```

Key sequences

If you need to add modifier keys to your sequence, instead of `send_string` use `simple_key_sequence`. While it's not as visually clean as `send_string`, you can use it to add things like copying/pasting, tabbing between fields, etc.

```
from kmk.handlers.sequences import simple_key_sequence

PASTE_WITH_COMMENTARY = simple_key_sequence(
        (
                KC.L,
                KC.O,
                KC.O,
                KC.K,
                KC.SPC,
                KC.A,
                KC.T,
                KC.SPC,
                KC.T,
```

```
                KC.H,
                KC.I,
                KC.S,
                KC.COLN,
                KC.SPC,
                KC.LCTL(KC.V),
        )
)

keyboard.keymap = [<other keycodes>, PASTE_WITH_COMMENTARY, <other keycodes>]
```

The above example will type out "look at this: " and then paste the contents of your clipboard.

Sleeping within a sequence

If you need to wait during a sequence, you can use `KC.MACRO_SLEEP_MS(ms)` to wait a length of time, in milliseconds.

```
from kmk.handlers.sequences import simple_key_sequence

COUNTDOWN_TO_PASTE = simple_key_sequence(
        (
                KC.N3,
                KC.ENTER,
                KC.MACRO_SLEEP_MS(1000),
                KC.N2,
                KC.ENTER,
                KC.MACRO_SLEEP_MS(1000),
                KC.N1,
                KC.ENTER,
                KC.MACRO_SLEEP(1000),
                KC.LCTL(KC.V),
        )
)

keyboard.keymap = [<other keycodes>, COUNTDOWN_TO_PASTE, <other keycodes>]
```

This example will type out the following, waiting one second (1000 ms) between numbers:

```
3
2
1
```

and then paste the contents of your clipboard.

Alt Tab with delay

If alt tab isn't working because it requires a delay, adding a delay and triggering down and up on ALT manually may fix the issue.

```
from kmk.handlers.sequences import simple_key_sequence

NEXT = simple_key_sequence(
    (
        KC.LALT(no_release=True),
        KC.MACRO_SLEEP_MS(30),
        KC.TAB,
        KC.MACRO_SLEEP_MS(30),
        KC.LALT(no_press=True),
    )
)
```

Unicode

Before trying to send Unicode sequences, make sure you set your `UnicodeMode`. You can set an initial value in your keymap by setting `keyboard.unicode_mode`.

Keys are provided to change this mode at runtime - for example, `KC.UC_MODE_LINUX`.

Unicode Modes:

On Linux, Unicode uses `Ctrl-Shift-U`, which is supported by `ibus` and GTK+3. `ibus` users will need to add `IBUS_ENABLE_CTRL_SHIFT_U=1` to their environment (`~/profile`, `~/.bashrc`, `~/.zshrc`, or through your desktop environment's configurator).

On Windows, WinCompose is required.

- Linux : `UnicodeMode.LINUX` or `UnicodeMode.IBUS`
- Mac: `UnicodeMode.MACOS` or `UnicodeMode.OSX` or `UnicodeMode.RALT`
- Windows: `UnicodeMode.WINC`

Unicode Examples

To send a simple Unicode symbol

```
from kmk.handlers.sequences import unicode_string_sequence

FLIP = unicode_string_sequence('(ノ☉Д☉)ノ彡┻━━┻')

keyboard.keymap = [<other keycodes>, FLIP, <other keycodes>]
```

If you'd rather keep a lookup table of your sequences (perhaps to bind emojis to keys), that's supported too, through an obnoxiously long-winded method:

```
from kmk.handlers.sequences import compile_unicode_string_sequences as cuss

emoticons = cuss({
        'BEER': r'🍺',
        'HAND_WAVE': r'👋',
})

keymap = [<other keycodes>, emoticons.BEER, emoticons.HAND_WAVE, <other
keycodes>]
```

> The observant will notice dot-notation is supported here despite feeding in a dictionary - the return of `compile_unicode_string_sequences` is a `kmk.types.AttrDict`, which you can think of as a read-only view over a dictionary adding attribute-based (dot-notation) access.

Finally, if you need to send arbitrary Unicode codepoints in raw form, that's supported too, through `unicode_codepoint_sequence`.

```
from kmk.handlers.sequences import unicode_codepoint_sequence

TABLE_FLIP = unicode_codepoint_sequence([
        "28", "30ce", "ca0", "75ca","ca0", "29",
        "30ce", "5f61", "253b", "2501", "253b",
])

keyboard.keymap = [<other keycodes>, TABLE_FLIP, <other keycodes>]
```