Tap Dance: A Single Key Can Do 3, 5, or 100 Different Things

Introduction

Hit the semicolon key once, send a semicolon. Hit it twice, rapidly – send a colon. Hit it three times, and your keyboard's LEDs do a wild dance. That's just one example of what Tap Dance can do. It's one of the nicest community-contributed features in the firmware, conceived and created by algernon in #451. Here's how algernon describes the feature:

With this feature one can specify keys that behave differently, based on the amount of times they have been tapped, and when interrupted, they get handled before the interrupter.

Explanatory Comparison with `ACTION_FUNCTION_TAP`

`ACTION_FUNCTION_TAP` can offer similar functionality to Tap Dance, but it's worth noting some important differences. To do this, let's explore a certain setup! We want one key to send `Space` on single-tap, but `Enter` on double-tap.

With `ACTION_FUNCTION_TAP`, it is quite a rain-dance to set this up, and has the problem that when the sequence is interrupted, the interrupting key will be sent first. Thus, `SPC a` will result in `a SPC` being sent, if `SPC` and `a` are both typed within `TAPPING_TERM`. With the Tap Dance feature, that'll come out correctly as `SPC a` (even if both `SPC` and `a` are typed within the `TAPPING_TERM`.

To achieve this correct handling of interrupts, the implementation of Tap Dance hooks into two parts of the system: `process_record_quantum()`, and the matrix scan. These two parts are explained below, but for now the point to note is that we need the latter to be able to time out a tap sequence even when a key is not being pressed. That way, `SPC` alone will time out and register after `TAPPING_TERM` time.

How to Use Tap Dance

But enough of the generalities; lets look at how to actually use Tap Dance!

First, you will need `TAP_DANCE_ENABLE=yes` in your `rules.mk`, because the feature is disabled by default. This adds a little less than 1k to the firmware size.

Optionally, you might want to set a custom `TAPPING_TERM` time by adding something like this in you `config.h`:

```
#define TAPPING_TERM 175
```

The `TAPPING_TERM` time is the maximum time allowed between taps of your Tap Dance key, and is measured in milliseconds. For example, if you used the above

`#define` statement and set up a Tap Dance key that sends `Space` on single-tap and `Enter` on double-tap, then this key will send `ENT` only if you tap this key twice in less than 175ms. If you tap the key, wait more than 175ms, and tap the key again you'll end up sending `SPC SPC` instead.

Next, you will want to define some tap-dance keys, which is easiest to do with the `TD()` macro, that - similar to `F()` - takes a number, which will later be used as an index into the `tap_dance_actions` array.

After this, you'll want to use the `tap_dance_actions` array to specify what actions shall be taken when a tap-dance key is in action. Currently, there are five possible options:

- `ACTION_TAP_DANCE_DOUBLE(kc1, kc2)`: Sends the `kc1` keycode when tapped once, `kc2` otherwise. When the key is held, the appropriate keycode is registered: `kc1` when pressed and held, `kc2` when tapped once, then pressed and held.
- `ACTION_TAP_DANCE_LAYER_MOVE(kc, layer)`: Sends the `kc` keycode when tapped once, or moves to `layer`. (this functions like the `TO` layer keycode).
  - This is the same as `ACTION_TAP_DANCE_DUAL_ROLE`, but renamed to something that is clearer about its functionality. Both names will work.
- `ACTION_TAP_DANCE_LAYER_TOGGLE(kc, layer)`: Sends the `kc` keycode when tapped once, or toggles the state of `layer`. (this functions like the `TG` layer keycode).
- `ACTION_TAP_DANCE_FN(fn)`: Calls the specified function - defined in the user keymap - with the final tap count of the tap dance action.
- `ACTION_TAP_DANCE_FN_ADVANCED(on_each_tap_fn, on_dance_finished_fn, on_dance_reset_fn)`: Calls the first specified function - defined in the user keymap - on every tap, the second function when the dance action finishes (like the previous option), and the last function when the tap dance action resets.
- `ACTION_TAP_DANCE_FN_ADVANCED_TIME(on_each_tap_fn, on_dance_finished_fn, on_dance_reset_fn, tap_specific_tapping_term)`: This functions identically to the `ACTION_TAP_DANCE_FN_ADVANCED` function, but uses a custom tapping term for it, instead of the predefined `TAPPING_TERM`.

The first option is enough for a lot of cases, that just want dual roles. For example, `ACTION_TAP_DANCE_DOUBLE(KC_SPC, KC_ENT)` will result in `Space` being sent on single-tap, `Enter` otherwise.

!> Keep in mind that only basic keycodes are supported here. Custom keycodes are not supported.

Similar to the first option, the second option is good for simple layer-switching cases.

For more complicated cases, use the third or fourth options (examples of each are listed below).

Finally, the fifth option is particularly useful if your non-Tap-Dance keys start behaving weirdly after adding the code for your Tap Dance keys. The likely problem is that you changed the `TAPPING_TERM` time to make your Tap Dance keys easier for you to use, and that this has changed the way your other keys handle interrupts.

Implementation Details

Well, that's the bulk of it! You should now be able to work through the examples below, and to develop your own Tap Dance functionality. But if you want a deeper understanding of what's going on behind the scenes, then read on for the explanation of how it all works!

The main entry point is `process_tap_dance()`, called from `process_record_quantum()`, which is run for every keypress, and our handler gets to run early. This function checks whether the key pressed is a tap-dance key. If it is not, and a tap-dance was in action, we handle that first, and enqueue the newly pressed key. If it is a tap-dance key, then we check if it is the same as the already active one (if there's one active, that is). If it is not, we fire off the old one first, then register the new one. If it was the same, we increment the counter and reset the timer.

This means that you have `TAPPING_TERM` time to tap the key again; you do not have to input all the taps within a single `TAPPING_TERM` timeframe. This allows for longer tap counts, with minimal impact on responsiveness.

Our next stop is `matrix_scan_tap_dance()`. This handles the timeout of tap-dance keys.

For the sake of flexibility, tap-dance actions can be either a pair of keycodes, or a user function. The latter allows one to handle higher tap counts, or do extra things, like blink the LEDs, fiddle with the backlighting, and so on. This is accomplished by using an union, and some clever macros.

Examples

Simple Example

Here's a simple example for a single definition:

1. In your `rules.mk`, add `TAP_DANCE_ENABLE = yes`
2. In your `config.h` (which you can copy from `qmk_firmware/keyboards/planck/config.h` to your keymap directory), add `#define TAPPING_TERM 200`
3. In your `keymap.c` file, define the variables and definitions, then add to your keymap:

```
//Tap Dance Declarations
enum {
  TD_ESC_CAPS = 0
```

```
};

//Tap Dance Definitions
qk_tap_dance_action_t tap_dance_actions[] = {
  //Tap once for Esc, twice for Caps Lock
  [TD_ESC_CAPS]  = ACTION_TAP_DANCE_DOUBLE(KC_ESC, KC_CAPS)
// Other declarations would go here, separated by commas, if you have them
};

//In Layer declaration, add tap dance item in place of a key code
TD(TD_ESC_CAPS)
```

Complex Examples

This section details several complex tap dance examples. All the enums used in the examples are declared like this:

```
// Enums defined for all examples:
enum {
 CT_SE = 0,
 CT_CLN,
 CT_EGG,
 CT_FLSH,
 X_TAP_DANCE
};
```

Example 1: Send : on Single Tap, ; on Double Tap

```
void dance_cln_finished (qk_tap_dance_state_t *state, void *user_data) {
  if (state->count == 1) {
    register_code (KC_RSFT);
    register_code (KC_SCLN);
  } else {
    register_code (KC_SCLN);
  }
}

void dance_cln_reset (qk_tap_dance_state_t *state, void *user_data) {
  if (state->count == 1) {
    unregister_code (KC_RSFT);
    unregister_code (KC_SCLN);
  } else {
    unregister_code (KC_SCLN);
  }
}

//All tap dance functions would go here. Only showing this one.
```

```
qk_tap_dance_action_t tap_dance_actions[] = {
 [CT_CLN] = ACTION_TAP_DANCE_FN_ADVANCED (NULL, dance_cln_finished, dance_cln_reset)
};
```

Example 2: Send "Safety Dance!" After 100 Taps

```
void dance_egg (qk_tap_dance_state_t *state, void *user_data) {
  if (state->count >= 100) {
    SEND_STRING ("Safety dance!");
    reset_tap_dance (state);
  }
}

qk_tap_dance_action_t tap_dance_actions[] = {
 [CT_EGG] = ACTION_TAP_DANCE_FN (dance_egg)
};
```

Example 3: Turn LED Lights On Then Off, One at a Time

```
// on each tap, light up one led, from right to left
// on the forth tap, turn them off from right to left
void dance_flsh_each(qk_tap_dance_state_t *state, void *user_data) {
  switch (state->count) {
  case 1:
    ergodox_right_led_3_on();
    break;
  case 2:
    ergodox_right_led_2_on();
    break;
  case 3:
    ergodox_right_led_1_on();
    break;
  case 4:
    ergodox_right_led_3_off();
    _delay_ms(50);
    ergodox_right_led_2_off();
    _delay_ms(50);
    ergodox_right_led_1_off();
  }
}

// on the fourth tap, set the keyboard on flash state
void dance_flsh_finished(qk_tap_dance_state_t *state, void *user_data) {
  if (state->count >= 4) {
    reset_keyboard();
    reset_tap_dance(state);
```

```
  }
}

// if the flash state didn't happen, then turn off LEDs, left to right
void dance_flsh_reset(qk_tap_dance_state_t *state, void *user_data) {
  ergodox_right_led_1_off();
  _delay_ms(50);
  ergodox_right_led_2_off();
  _delay_ms(50);
  ergodox_right_led_3_off();
}

//All tap dances now put together. Example 3 is "CT_FLASH"
qk_tap_dance_action_t tap_dance_actions[] = {
  [CT_SE]  = ACTION_TAP_DANCE_DOUBLE (KC_SPC, KC_ENT)
 ,[CT_CLN] = ACTION_TAP_DANCE_FN_ADVANCED (NULL, dance_cln_finished, dance_cln_reset)
 ,[CT_EGG] = ACTION_TAP_DANCE_FN (dance_egg)
 ,[CT_FLSH] = ACTION_TAP_DANCE_FN_ADVANCED (dance_flsh_each, dance_flsh_finished, dance_flsh
};
```

Example 4: 'Quad Function Tap-Dance'

By DanielGGordon

Allow one key to have 4 (or more) functions, depending on number of presses, and if the key is held or tapped. Below is a specific example: * Tap = Send `x` * Hold = Send `Control` * Double Tap = Send `Escape` * Double Tap and Hold = Send `Alt`

Setup

You will need a few things that can be used for 'Quad Function Tap-Dance'.

You'll need to add these to the top of your `keymap.c` file, before your keymap.

```
typedef struct {
  bool is_press_action;
  int state;
} tap;

enum {
  SINGLE_TAP = 1,
  SINGLE_HOLD = 2,
  DOUBLE_TAP = 3,
  DOUBLE_HOLD = 4,
  DOUBLE_SINGLE_TAP = 5, //send two single taps
  TRIPLE_TAP = 6,
  TRIPLE_HOLD = 7
```

```
};

//Tap dance enums
enum {
  X_CTL = 0,
  SOME_OTHER_DANCE
};

int cur_dance (qk_tap_dance_state_t *state);

//for the x tap dance. Put it here so it can be used in any keymap
void x_finished (qk_tap_dance_state_t *state, void *user_data);
void x_reset (qk_tap_dance_state_t *state, void *user_data);
```

Now, at the bottom of your `keymap.c` file, you'll need to add the following:

```
/* Return an integer that corresponds to what kind of tap dance should be executed.
 *
 * How to figure out tap dance state: interrupted and pressed.
 *
 * Interrupted: If the state of a dance dance is "interrupted", that means that another key
 *  under the tapping term. This is typically indicitive that you are trying to "tap" the ke
 *
 * Pressed: Whether or not the key is still being pressed. If this value is true, that means
 *  has ended, but the key is still being pressed down. This generally means the key is bei
 *
 * One thing that is currenlty not possible with qmk software in regards to tap dance is to
 *  feature. In general, advanced tap dances do not work well if they are used with commonly
 *  For example "A". Tap dances are best used on non-letter keys that are not hit while typ
 *
 * Good places to put an advanced tap dance:
 *  z,q,x,j,k,v,b, any function key, home/end, comma, semi-colon
 *
 * Criteria for "good placement" of a tap dance key:
 *  Not a key that is hit frequently in a sentence
 *  Not a key that is used frequently to double tap, for example 'tab' is often double tappe
 *    in a web form. So 'tab' would be a poor choice for a tap dance.
 *  Letters used in common words as a double. For example 'p' in 'pepper'. If a tap dance fu
 *    letter 'p', the word 'pepper' would be quite frustating to type.
 *
 * For the third point, there does exist the 'DOUBLE_SINGLE_TAP', however this is not fully
 *
 */
int cur_dance (qk_tap_dance_state_t *state) {
  if (state->count == 1) {
    if (state->interrupted || !state->pressed)  return SINGLE_TAP;
    //key has not been interrupted, but they key is still held. Means you want to send a 'HU
```

```
    else return SINGLE_HOLD;
  }
  else if (state->count == 2) {
    /*
     * DOUBLE_SINGLE_TAP is to distinguish between typing "pepper", and actually wanting a d
     * action when hitting 'pp'. Suggested use case for this return value is when you want t
     * keystrokes of the key, and not the 'double tap' action/macro.
    */
    if (state->interrupted) return DOUBLE_SINGLE_TAP;
    else if (state->pressed) return DOUBLE_HOLD;
    else return DOUBLE_TAP;
  }
  //Assumes no one is trying to type the same letter three times (at least not quickly).
  //If your tap dance key is 'KC_W', and you want to type "www." quickly - then you will nee
  //an exception here to return a 'TRIPLE_SINGLE_TAP', and define that enum just like 'DOUBL
  if (state->count == 3) {
    if (state->interrupted || !state->pressed)  return TRIPLE_TAP;
    else return TRIPLE_HOLD;
  }
  else return 8; //magic number. At some point this method will expand to work for more pres
}

//instanalize an instance of 'tap' for the 'x' tap dance.
static tap xtap_state = {
  .is_press_action = true,
  .state = 0
};

void x_finished (qk_tap_dance_state_t *state, void *user_data) {
  xtap_state.state = cur_dance(state);
  switch (xtap_state.state) {
    case SINGLE_TAP: register_code(KC_X); break;
    case SINGLE_HOLD: register_code(KC_LCTRL); break;
    case DOUBLE_TAP: register_code(KC_ESC); break;
    case DOUBLE_HOLD: register_code(KC_LALT); break;
    case DOUBLE_SINGLE_TAP: register_code(KC_X); unregister_code(KC_X); register_code(KC_X);
    //Last case is for fast typing. Assuming your key is `f`:
    //For example, when typing the word `buffer`, and you want to make sure that you send `f
    //In order to type `ff` when typing fast, the next character will have to be hit within
  }
}

void x_reset (qk_tap_dance_state_t *state, void *user_data) {
  switch (xtap_state.state) {
    case SINGLE_TAP: unregister_code(KC_X); break;
    case SINGLE_HOLD: unregister_code(KC_LCTRL); break;
```

```
    case DOUBLE_TAP: unregister_code(KC_ESC); break;
    case DOUBLE_HOLD: unregister_code(KC_LALT);
    case DOUBLE_SINGLE_TAP: unregister_code(KC_X);
  }
  xtap_state.state = 0;
}

qk_tap_dance_action_t tap_dance_actions[] = {
  [X_CTL]     = ACTION_TAP_DANCE_FN_ADVANCED(NULL,x_finished, x_reset)
};
```

And then simply use `TD(X_CTL)` anywhere in your keymap.

If you want to implement this in your userspace, then you may want to check out how DanielGGordon has implemented this in their userspace.

Example 5: Using tap dance for advanced mod-tap and layer-tap keys

Tap dance can be used to emulate `MT()` and `LT()` behavior when the tapped code is not a basic keycode. This is useful to send tapped keycodes that normally require `Shift`, such as parentheses or curly braces—or other modified keycodes, such as `Control + X`.

Below your layers and custom keycodes, add the following:

```
// tapdance keycodes
enum td_keycodes {
  ALT_LP // Our example key: `LALT` when held, `(` when tapped. Add additional keycodes for
};

// define a type containing as many tapdance states as you need
typedef enum {
  SINGLE_TAP,
  SINGLE_HOLD,
  DOUBLE_SINGLE_TAP
} td_state_t;

// create a global instance of the tapdance state type
static td_state_t td_state;

// declare your tapdance functions:

// function to determine the current tapdance state
int cur_dance (qk_tap_dance_state_t *state);

// `finished` and `reset` functions for each tapdance keycode
void altlp_finished (qk_tap_dance_state_t *state, void *user_data);
void altlp_reset (qk_tap_dance_state_t *state, void *user_data);
```

Below your `LAYOUT`, define each of the tapdance functions:

```
// determine the tapdance state to return
int cur_dance (qk_tap_dance_state_t *state) {
  if (state->count == 1) {
    if (state->interrupted || !state->pressed) { return SINGLE_TAP; }
    else { return SINGLE_HOLD; }
  }
  if (state->count == 2) { return DOUBLE_SINGLE_TAP; }
  else { return 3; } // any number higher than the maximum state value you return above
}

// handle the possible states for each tapdance keycode you define:

void altlp_finished (qk_tap_dance_state_t *state, void *user_data) {
  td_state = cur_dance(state);
  switch (td_state) {
    case SINGLE_TAP:
      register_code16(KC_LPRN);
      break;
    case SINGLE_HOLD:
      register_mods(MOD_BIT(KC_LALT)); // for a layer-tap key, use `layer_on(_MY_LAYER)` he
      break;
    case DOUBLE_SINGLE_TAP: // allow nesting of 2 parens `((` within tapping term
      tap_code16(KC_LPRN);
      register_code16(KC_LPRN);
  }
}

void altlp_reset (qk_tap_dance_state_t *state, void *user_data) {
  switch (td_state) {
    case SINGLE_TAP:
      unregister_code16(KC_LPRN);
      break;
    case SINGLE_HOLD:
      unregister_mods(MOD_BIT(KC_LALT)); // for a layer-tap key, use `layer_off(_MY_LAYER)`
      break;
    case DOUBLE_SINGLE_TAP:
      unregister_code16(KC_LPRN);
  }
}

// define `ACTION_TAP_DANCE_FN_ADVANCED()` for each tapdance keycode, passing in `finished`
qk_tap_dance_action_t tap_dance_actions[] = {
  [ALT_LP] = ACTION_TAP_DANCE_FN_ADVANCED(NULL, altlp_finished, altlp_reset)
};
```

Wrap each tapdance keycode in `TD()` when including it in your keymap, e.g. `TD(ALT_LP)`.

Example 6: Using tap dance for momentary-layer-switch and layer-toggle keys

Tap Dance can be used to mimic MO(layer) and TG(layer) functionality. For this example, we will set up a key to function as `KC_QUOT` on single-tap, as `MO(_MY_LAYER)` on single-hold, and `TG(_MY_LAYER)` on double-tap.

The first step is to include the following code towards the beginning of your `keymap.c`:

```
typedef struct {
  bool is_press_action;
  int state;
} tap;

//Define a type for as many tap dance states as you need
enum {
  SINGLE_TAP = 1,
  SINGLE_HOLD = 2,
  DOUBLE_TAP = 3
};

enum {
  QUOT_LAYR = 0     //Our custom tap dance key; add any other tap dance keys to this enum
};

//Declare the functions to be used with your tap dance key(s)

//Function associated with all tap dances
int cur_dance (qk_tap_dance_state_t *state);

//Functions associated with individual tap dances
void ql_finished (qk_tap_dance_state_t *state, void *user_data);
void ql_reset (qk_tap_dance_state_t *state, void *user_data);

//Declare variable to track which layer is active
int active_layer;
```

The above code is similar to that used in previous examples. The one point to note is that you need to declare a variable to keep track of what layer is currently the active layer. We'll see why shortly.

Towards the bottom of your `keymap.c`, include the following code:

```
//Update active_layer
```

```
uint32_t layer_state_set_user(uint32_t state) {
  switch (biton32(state)) {
    case 1:
      active_layer = 1;
      break;
    case 2:
      active_layer = 2;
      break;
    case 3:
      active_layer = 3;
      break;
    default:
      active_layer = 0;
      break;
  }
  return state;
}

//Determine the current tap dance state
int cur_dance (qk_tap_dance_state_t *state) {
  if (state->count == 1) {
    if (!state->pressed) {return SINGLE_TAP;}
    else return SINGLE_HOLD;
  } else if (state->count == 2) {return DOUBLE_TAP;}
  else return 8;
}

//Initialize tap structure associated with example tap dance key
static tap ql_tap_state = {
  .is_press_action = true,
  .state = 0
};

//Functions that control what our tap dance key does
void ql_finished (qk_tap_dance_state_t *state, void *user_data) {
  ql_tap_state.state = cur_dance(state);
  switch (ql_tap_state.state) {
    case SINGLE_TAP: tap_code(KC_QUOT); break;
    case SINGLE_HOLD: layer_on(_MY_LAYER); break;
    case DOUBLE_TAP:
      if (active_layer==_MY_LAYER) {layer_off(_MY_LAYER);}
      else layer_on(_MY_LAYER);
  }
}

void ql_reset (qk_tap_dance_state_t *state, void *user_data) {
```

```
  if (ql_tap_state.state==SINGLE_HOLD) {layer_off(_MY_LAYER);}
  ql_tap_state.state = 0;
}

//Associate our tap dance key with its functionality
qk_tap_dance_action_t tap_dance_actions[] = {
  [QUOT_LAYR] = ACTION_TAP_DANCE_FN_ADVANCED_TIME(NULL, ql_finished, ql_reset, 275)
};
```

The is where the real logic of our tap dance key gets worked out. Since `layer_state_set_user()` is called on any layer switch, we use it to update `active_layer`. Our example is assuming that your `keymap.c` includes 4 layers, so adjust the switch statement here to fit your actual number of layers.

The use of `cur_dance()` and `ql_tap_state` mirrors the above examples.

The `case:SINGLE_TAP` in `ql_finished` is similar to the above examples. The `case:SINGLE_HOLD` works in conjunction with `ql_reset()` to switch to `_MY_LAYER` while the tap dance key is held, and to switch away from `_MY_LAYER` when the key is released. This mirrors the use of `MO(_MY_LAYER)`. The `case:DOUBLE_TAP` works by checking whether `_MY_LAYER` is the active layer, and toggling it on or off accordingly. This mirrors the use of `TG(_MY_LAYER)`.

`tap_dance_actions[]` works similar to the above examples. Note that I used `ACTION_TAP_DANCE_FN_ADVANCED_TIME()` instead of `ACTION_TAP_DANCE_FN_ADVANCED()`. This is because I like my `TAPPING_TERM` to be short (~175ms) for my non-tap-dance keys but find that this is too quick for me to reliably complete tap dance actions - thus the increased time of 275ms here.

Finally, to get this tap dance key working, be sure to include `TD(QUOT_LAYR)` in your `keymaps[]`.