LwBTN

Tilen MAJERLE

Dec 08, 2023

CONTENTS

Welcome to the documentation for version .

Download library Getting started Open Github Donate

CHAPTER

ONE

FEATURES

- Written in C (C11)
- Platform independent, requires user to provide millisecond timing source
- No dynamic memory allocation
- Callback driven event management
- Easy to use and maintain
- User friendly MIT license

CHAPTER

TWO

REQUIREMENTS

- C compiler
- Few kB of non-volatile memory

CHAPTER

THREE

CONTRIBUTE

Fresh contributions are always welcome. Simple instructions to proceed:

1. Fork Github repository
2. Respect C style & coding rules used by the library
3. Create a pull request to `develop` branch with new features or bug fixes

Alternatively you may:

1. Report a bug
2. Ask for a feature request

CHAPTER

FOUR

LICENSE

```
MIT License

Copyright (c) 2023 Tilen MAJERLE

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER

FIVE

TABLE OF CONTENTS

5.1 Getting started

Getting started may be the most challenging part of every new library. This guide is describing how to start with the library quickly and effectively

5.1.1 Download library

Library is primarly hosted on Github.

You can get it by:

- Downloading latest release from releases area on Github
- Cloning `main` branch for latest stable version
- Cloning `develop` branch for latest development

Download from releases

All releases are available on Github releases area.

Clone from Github

First-time clone

This is used when you do not have yet local copy on your machine.

- Make sure `git` is installed.
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Clone repository with one of available options below
  - Run `git clone --recurse-submodules https://github.com/MaJerle/lwbtn` command to clone entire repository, including submodules
  - Run `git clone --recurse-submodules --branch develop https://github.com/MaJerle/lwbtn` to clone *development* branch, including submodules
  - Run `git clone --recurse-submodules --branch main https://github.com/MaJerle/lwbtn` to clone *latest stable* branch, including submodules
- Navigate to `examples` directory and run favourite example

Update cloned to latest version

- Open console and navigate to path in the system where your repository is located. Use command `cd your_path`
- Run `git pull origin main` command to get latest changes on `main` branch
- Run `git pull origin develop` command to get latest changes on `develop` branch
- Run `git submodule update --init --remote` to update submodules to latest version

Note: This is preferred option to use when you want to evaluate library and run prepared examples. Repository consists of multiple submodules which can be automatically downloaded when cloning and pulling changes from root repository.

5.1.2 Add library to project

At this point it is assumed that you have successfully download library, either cloned it or from releases page. Next step is to add the library to the project, by means of source files to compiler inputs and header files in search path

- Copy `lwbtn` folder to your project, it contains library files
- Add `lwbtn/src/include` folder to *include path* of your toolchain. This is where *C/C++* compiler can find the files during compilation process. Usually using `-I` flag
- Add source files from `lwbtn/src/` folder to toolchain build. These files are built by *C/C++* compiler. CMake configuration comes with the library, allows users to include library in the project as **subdirectory** and **library**.
- Copy `lwbtn/src/include/lwbtn/lwbtn_opts_template.h` to project folder and rename it to `lwbtn_opts.h`
- Copy `lwbtn/src/include/lwbtn/lwbtn_types_template.h` to project folder and rename it to `lwbtn_types.h`
- Build the project

5.1.3 Configuration file

Configuration file is used to overwrite default settings defined for the essential use case. Library comes with template config file, which can be modified according to the application needs. and it should be copied (or simply renamed in-place) and named `lwbtn_opts.h`

Note: Default configuration template file location: `lwbtn/src/include/lwbtn/lwbtn_opts_template.h`. File must be renamed to `lwbtn_opts.h` first and then copied to the project directory where compiler include paths have access to it by using `#include "lwbtn_opts.h"`.

List of configuration options are available in the *Configuration* section. If any option is about to be modified, it should be done in configuration file

Listing 1: Template configuration file

```
1 /**
2  * \file            lwbtn_opts_template.h
3  * \brief           LwBTN configuration file
```

(continues on next page)

(continued from previous page)

```
 */

/*
 * Copyright (c) 2023 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of LwBTN - Lightweight button manager.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.0
 */
#ifndef LWBTN_OPTS_HDR_H
#define LWBTN_OPTS_HDR_H

/* Rename this file to "lwbtn_opts.h" for your application */

/*
 * Open "include/lwbtn/lwbtn_opt.h" and
 * copy & replace here settings you want to change values
 */

#endif /* LWBTN_OPTS_HDR_H */
```

Note: If you prefer to avoid using configuration file, application must define a global symbol `LWBTN_IGNORE_USER_OPTS`, visible across entire application. This can be achieved with `-D` compiler option.

5.2 User manual

LwBTN is simple button manager library, with great focus on embedded systems. Motivation behind start of development was linked to several on-going projects including some input reading (button handling), each of them demanding little differences in process.

LwBTN is therefore relatively simple and lightweight, yet it can provide pretty comprehensive processing of your application buttons.

5.2.1 How it works

User must define buttons array and pass it to the library. Next to that, 2 more functions are required:

- Function to read the architecture button state
- Function to receive various button events

User shall later periodically call processing function with current system time as simple parameter and get ready to receive various events.

A simple example for win32 is below:

Listing 2: Win32 example code

```c
#include "lwbtn/lwbtn.h"
#include "windows.h"
#include <stdio.h>
#include <stdlib.h>

static LARGE_INTEGER freq, sys_start_time;
static uint32_t get_tick(void);

/* User defined settings */
const int keys[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
uint32_t last_time_keys[sizeof(keys) / sizeof(keys[0])] = {0};

/* List of buttons to process with assigned custom arguments for callback functions */
static lwbtn_btn_t btns[] = {
    {.arg = (void*)&keys[0]}, {.arg = (void*)&keys[1]}, {.arg = (void*)&keys[2]}, {.arg = (void*)&keys[3]},
    {.arg = (void*)&keys[4]}, {.arg = (void*)&keys[5]}, {.arg = (void*)&keys[6]}, {.arg = (void*)&keys[7]},
    {.arg = (void*)&keys[8]}, {.arg = (void*)&keys[9]},
};

/**
 * \brief           Get input state callback
 * \param           lw: LwBTN instance
 * \param           btn: Button instance
 * \return          `1` if button active, `0` otherwise
 */
uint8_t
prv_btn_get_state(struct lwbtn* lw, struct lwbtn_btn* btn) {
    (void)lw;
```

(continues on next page)

(continued from previous page)

```

    /*
     * Function will return negative number if button is pressed,
     * or zero if button is releases
     */
    return GetAsyncKeyState(*(int*)btn->arg) < 0;
}

/**
 * \brief           Button event
 *
 * \param           lw: LwBTN instance
 * \param           btn: Button instance
 * \param           evt: Button event
 */
void
prv_btn_event(struct lwbtn* lw, struct lwbtn_btn* btn, lwbtn_evt_t evt) {
    const char* s;
    uint32_t color, keepalive_cnt = 0;
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    uint32_t* diff_time_ptr = &last_time_keys[(*(int*)btn->arg) - '0'];
    uint32_t diff_time = get_tick() - *diff_time_ptr;

    /* This is for purpose of test and timing validation */
    if (diff_time > 2000) {
        diff_time = 0;
    }
    *diff_time_ptr = get_tick(); /* Set current date as last one */

    /* Get event string */
    if (0) {
#if LWBTN_CFG_USE_KEEPALIVE
    } else if (evt == LWBTN_EVT_KEEPALIVE) {
        s = "KEEPALIVE";
        color = FOREGROUND_RED;
#endif /* LWBTN_CFG_USE_KEEPALIVE */
    } else if (evt == LWBTN_EVT_ONPRESS) {
        s = "  ONPRESS";
        color = FOREGROUND_GREEN;
    } else if (evt == LWBTN_EVT_ONRELEASE) {
        s = "ONRELEASE";
        color = FOREGROUND_BLUE;
    } else if (evt == LWBTN_EVT_ONCLICK) {
        s = "  ONCLICK";
        color = FOREGROUND_RED | FOREGROUND_GREEN;
    } else {
        s = "  UNKNOWN";
        color = FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE;
    }
#if LWBTN_CFG_USE_KEEPALIVE
    keepalive_cnt = btn->keepalive.cnt;
#endif
```

(continues on next page)

(continued from previous page)

```
    SetConsoleTextAttribute(hConsole, color);
    printf("[%7u][%6u] CH: %c, evt: %s, keep-alive cnt: %3u, click cnt: %3u\r\n",␣
↪(unsigned)get_tick(),
           (unsigned)diff_time, *(int*)btn->arg, s, (unsigned)keepalive_cnt,␣
↪(unsigned)btn->click.cnt);
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_
↪BLUE);
    (void)lw;
}

/**
 * \brief           Example function
 */
int
example_win32(void) {
    uint32_t time_last;
    printf("Application running\r\n");
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&sys_start_time);

    /* Define buttons */
    lwbtn_init_ex(NULL, btns, sizeof(btns) / sizeof(btns[0]), prv_btn_get_state, prv_btn_
↪event);

    time_last = get_tick();
    while (1) {
        /* Process forever */
        lwbtn_process_ex(NULL, get_tick());

        /* Manually read button state */
#if LWBTN_CFG_GET_STATE_MODE == LWBTN_GET_STATE_MODE_MANUAL
        for (size_t i = 0; i < sizeof(btns) / sizeof(btns[0]); ++i) {
            lwbtn_set_btn_state(&btns[i], prv_btn_get_state(NULL, &btns[i]));
        }
#endif /* LWBTN_CFG_GET_STATE_MODE == LWBTN_GET_STATE_MODE_MANUAL */

        /* Check if specific button is active and do some action */
        if (lwbtn_is_btn_active(&btns[0])) {
            if ((get_tick() - time_last) > 200) {
                time_last = get_tick();
                printf("Button is active\r\n");
            }
        }

        /* Artificial sleep to offload win process */
        Sleep(5);
    }
    return 0;
}

/**
 * \brief           Get current tick in ms from start of program
```

(continues on next page)

(continued from previous page)

```
 * \return          uint32_t: Tick in ms
 */
static uint32_t
get_tick(void) {
    LONGLONG ret;
    LARGE_INTEGER now;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&now);
    ret = now.QuadPart - sys_start_time.QuadPart;
    return (uint32_t)((ret * 1000) / freq.QuadPart);
}
```

5.2.2 Input events

During button (or input if you will) lifetime, application can expect some of these events (but not limited to):

- `LWBTN_EVT_ONPRESS` event is sent to application whenever input goes from inactive to active state and minimum debounce time passes by
- `LWBTN_EVT_ONRELEASE` event is sent to application whenever input sent **onpress** event prior to that and when input goes from active to inactive state
- `LWBTN_EVT_KEEPALIVE` event is periodically sent between **onpress** and **onrelease** events
- `LWBTN_EVT_ONCLICK` event is sent after **onrelease** and only if active button state was within allowed window for valid click event.

5.2.3 On-Press event

Onpress event is the first in a row when input is detected active. With nature of embedded systems and various buttons connected to devices, it is necessary to filter out potential noise to ignore unintential multiple presses. This is done by checking line to be at stable level for at least some minimum time, normally called *debounce time*, usually it takes around `20ms`.

Fig. 1: On-Press event trigger after minimum debounce time

5.2.4 On-Release event

Onrelease event is triggered immediately when input goes from active to inactive state, and only if onpress event has been detected prior to that.

Fig. 2: On-Release event trigger

5.2.5 On-Click event

Onclick event is triggered after a combination of multiple events:

- **Onpress** event shall be detected properly, indicating button has been pressed
- **Onrelease** event shall be detected, indicating button has been released
- Time between **onpress** and **onrelease** events has to be within time window

When conditions are met, **onclick** event is sent, either immediately after **onrelease** or after certain timeout after **onrelease** event.

Fig. 3: Sequence for valid click event

A windows-test program demonstration of events is visible below.

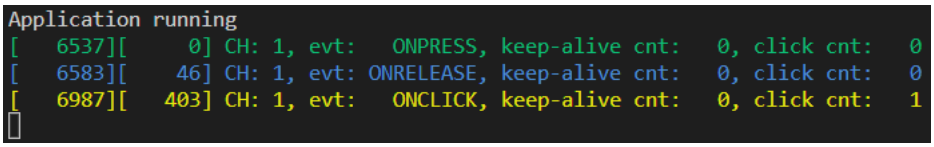

Fig. 4: Click event test program

Second number for each line is a **milliseconds** difference between events. OnClick is reported approximately (windows real-time issue) `400` ms after on-release event.

Tip: Timeout window between last **onrelease** event and **onclick** event is configurable

---

5.2.6 Multi-click events

Multi-click feature is where **timeout** for **onclick** event comes into play. Idea behind timeout feature is to allow multiple presses and to only send **onclick** once for all presses, including the number of detected presses during that time. This let's the application to react only once with known number of presses. This eliminates the problem where in case of **double** click trigger, you also receive **single-click** event, while you do not know yet, if second (or third) event will be triggered after.

Note: Imagine having a button that toggles one light on single click and turns off all lights in a room on double click. With timeout feature and single **onclick** notification, user will only receive the **onclick** once and will, based on the consecutive presses number value, perform appropriate action if it was single or multi click.

Simplified diagram for multi-click, ignoring debounce time indicators, is below. **cp** indicates number of detected **consecutive onclick press** events, to be reported in the final **onclick** event

Fig. 5: Multi-click event example - with 3 consecutive presses

A windows-test program demonstration of events is visible below.

Multi-click event with **onclick** event reported only after second press after minimum timeout of `400ms`.

Note: Number of consecutive clicks can be upper-limited to the desired value.

```
[ 601464][     0] CH: 1, evt:   ONPRESS, keep-alive cnt:   0, click cnt:   0
[ 601494][    30] CH: 1, evt: ONRELEASE, keep-alive cnt:   0, click cnt:   0
[ 601635][   141] CH: 1, evt:   ONPRESS, keep-alive cnt:   0, click cnt:   1
[ 601650][    15] CH: 1, evt: ONRELEASE, keep-alive cnt:   0, click cnt:   1
[ 602056][   406] CH: 1, evt:   ONCLICK, keep-alive cnt:   0, click cnt:   2
```

Fig. 6: Multi-click event test program

When user makes more (or equal) consecutive clicks than maximum, an **onclick** event is sent immediately after **onrelease** event for last detected click.

```
[ 187674][     0] CH: 1, evt:   ONPRESS, keep-alive cnt:   0, click cnt:   0
[ 187705][    31] CH: 1, evt: ONRELEASE, keep-alive cnt:   0, click cnt:   0
[ 187877][   172] CH: 1, evt:   ONPRESS, keep-alive cnt:   0, click cnt:   1
[ 187892][    15] CH: 1, evt: ONRELEASE, keep-alive cnt:   0, click cnt:   1
[ 188032][   140] CH: 1, evt:   ONPRESS, keep-alive cnt:   0, click cnt:   2
[ 188048][    16] CH: 1, evt: ONRELEASE, keep-alive cnt:   0, click cnt:   2
[ 188058][    10] CH: 1, evt:   ONCLICK, keep-alive cnt:   0, click cnt:   3
```

Fig. 7: Max number of onclick events, onclick is sent immediately after onrelease

There is no need to wait timeout expiration since upper clicks limit has been reached.

Tip: It is possible to control the behavior of **onclick** event (when consecutive number reaches maximum set value) timing using *LWBTN_CFG_CLICK_MAX_CONSECUTIVE_SEND_IMMEDIATELY* configuration. When enabled, behavior is as illustrated above. When disabled, **onclick** event it sent in timeout (or in case of new onpress), even if max allowed clicks has been reached.

Illustration below shows what happens during multiple clicks

- Max number of consecutive clicks is 3
- User makes 4 consecutive clicks

Fig. 8: Multi-click events with too many clicks - consecutive send immediately is enabled - it is sent after 3rd onrelease

Image below illustrates when send immediately is enabled. It is visible how first **onclick** is sent just after **onrelease** event (when max consecutive is set to 3).

When **multi-click** feature is disabled, **onclick** event is sent after every valid sequence of **onpress** and **onrelease** events.

Tip: If you do not want multi-click feature, set max number of consecutive clicks to 1. This will eliminate timeout feature since every click event will trigger **maximum clicks detected** and therefore send the event immediately after **onrelease**

Demo log text, with fast pressing of button, and events reported after every **onrelease**

Fig. 9: Multi-click events with too many clicks - consecutive send immediately is disabled

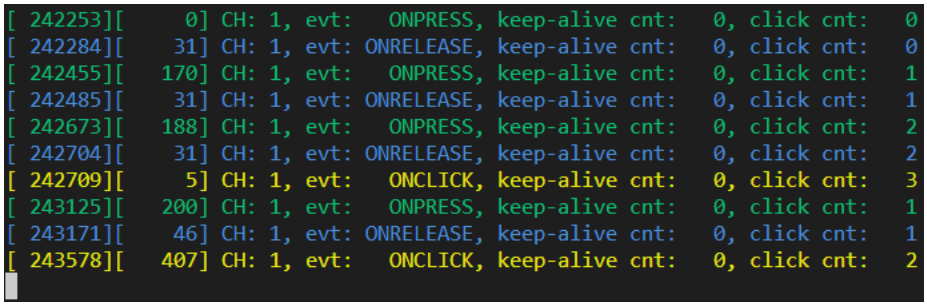

Fig. 10: 5 presses detected with 3 set as maximum. First on-click is sent immediately, while second is sent after timeout

5.2.7 Multi-click special case

There is currently a special case in the library when dealing with multiclicks. Configuration option *LWBTN_CFG_TIME_CLICK_MULTI_MAX* defines the maximum time between 2 consecutive clicks (consecutive **onrelease** events). Timing starts with **previous** valid click. If next click event starts (that starts with **onpress** event) earlier than maximum time but ends later than maximum, then new click is not counted as *consecutive* click to previous one.

As such, library will throw 2 **click** events to the user. First one immediately on second **onrelease** event (to take care of first **onpress** and **onrelease** event group) and second one after defined user timeout.

Note: Colors on picture below indicate events that relate to each other, indicated as **green** or **blue** rectangles

---

5.2.8 Keep alive event

Keep-alive event is sent periodically between **onpress** and **onrelease** events. It can be used to detect application is still alive and provides counter how many keep-alive events have been sent up to the point of event.

Feature can be used to make a trigger at specific time if button is in active state (a hold event).

5.2.9 Debounce

Debouncing is a software mechanics to remove unwanted bouncing events introduced by the physical buttons.

Tip: This chapter will not go into details about generic debouncing problem. Have a look at Wikipedia post about Switches.

Library supports 2 separate debounce options:

- **Debounce on press event**: This is almost always a must-have in the application, and helps to detect valid "press" event only once after the input is in stable active state for minimum time in a row.

Fig. 11: Multi-click events disabled with cp == 1

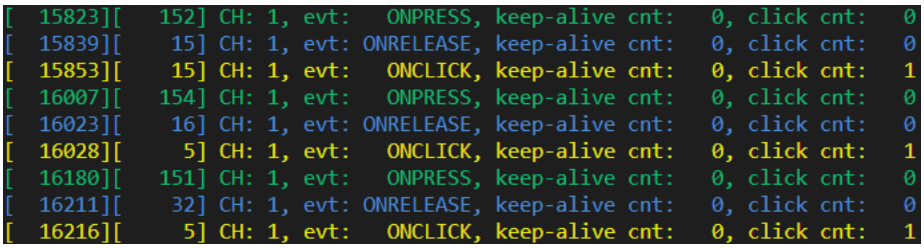

Fig. 12: Multi-click events disabled with cp == 1

Fig. 13: Special case for multi click when timing overlaps. Orange vertical lines indicate period for valid consecutive clicks.

Fig. 14: Keep alive events with 2 successful click events

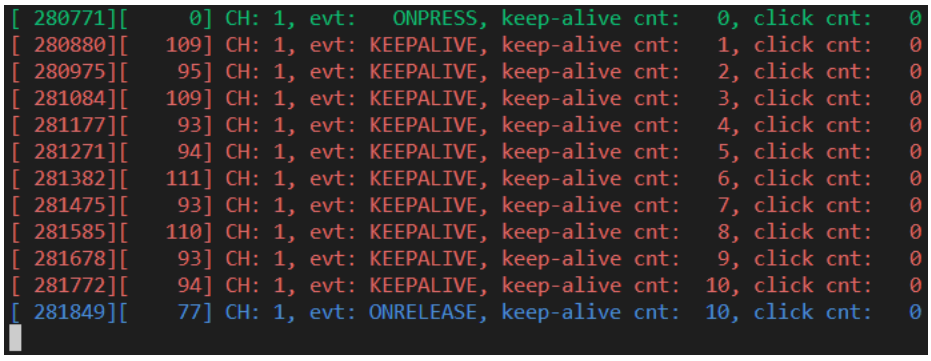

Fig. 15: Keep alive events when button is kept pressed

Press event debounce can only be disabled, if application can ensure stable transition from inactive to active state. This is usually done using capacitor and resistor next to the push button (this may not be the most optimized solution for contact longevity)

- **Debounce on release event**: This is usually not necessary by most of the applications,

  but can be used in harsh environments, where unwanted external noise could affect line and put it to inactive state for short period of time (while user holds button *down* in active state).

Note: Configuration settings *LWBTN_CFG_TIME_DEBOUNCE_PRESS* and *LWBTN_CFG_TIME_DEBOUNCE_RELEASE* are used to set the debounce time in milliseconds. When one of the values is set to `0`, debounce feature for respective transition is not actived.

Tip: Debounce time of around `20ms` is usually a good tradeoff between application reactivity to user events and debounce time required to stabilize the input.

Debounce examples

Examples are demonstrated using NUCLEO-L011K4 board. `2` GPIO pins are used, one in input config, second as output.

- **Input pin (Blue)**: A raw input that acts as an user button. There is no hardware filtering. Pin is active when *low* and inactive when *high*.
- **Output pin (Red)**: Output pin is software controlled. It goes *high* on *press* event and it goes *low* on *release* event. *Press* and *release* events are reported by the library.

Note: Logic analyzer has been connected directly to the microcontroller pins.

Examples #1

- *LWBTN_CFG_TIME_DEBOUNCE_PRESS* = `20`
- *LWBTN_CFG_TIME_DEBOUNCE_RELEASE* = `20`

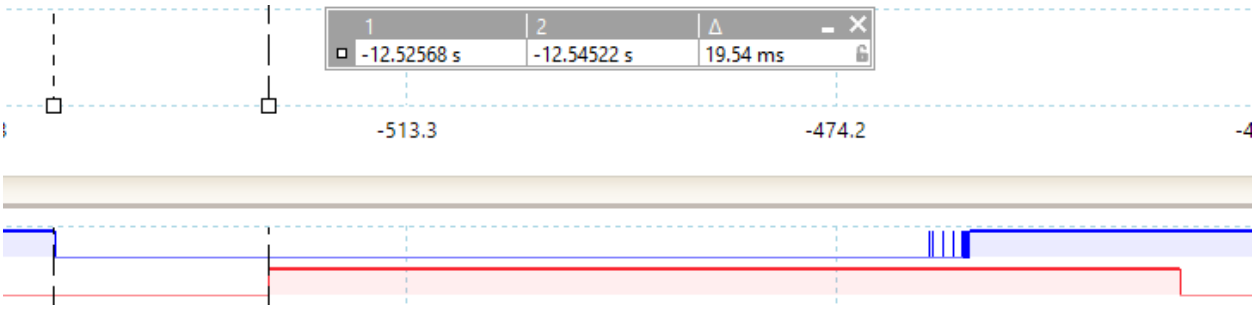

Fig. 16: 20ms debounce for press event. Press event is triggered only after input is stable in active state for minimum time.

Examples #2

- *LWBTN_CFG_TIME_DEBOUNCE_PRESS* = `20`
- *LWBTN_CFG_TIME_DEBOUNCE_RELEASE* = `0` - release debounce is disabled

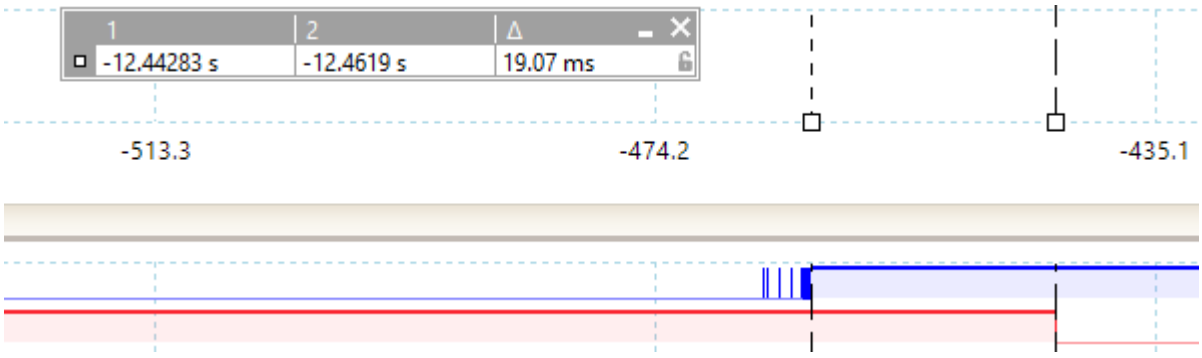

Fig. 17: 20ms debounce for release event. Release event is triggered only after input is stable in inactive state for minimum time.

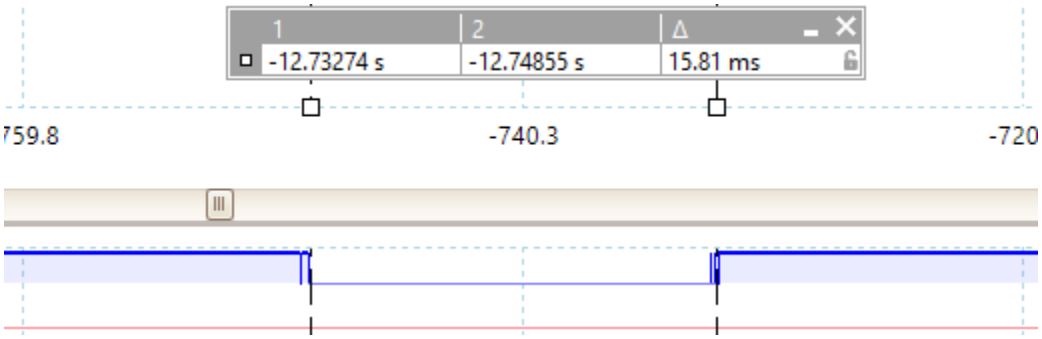

Fig. 18: 20ms debounce for press event - press event was not triggered - input was in stable active state for less than minimum debounce time (red line stays low).

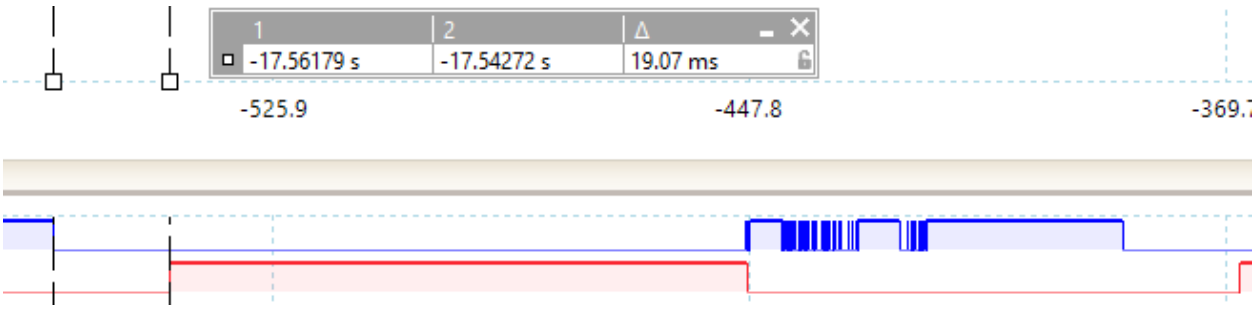

Fig. 19: Press event is detected after initial debounce, while release event is detected immediately on button going to inactive state.

Examples #3

- *LWBTN_CFG_TIME_DEBOUNCE_PRESS* = 100
- *LWBTN_CFG_TIME_DEBOUNCE_RELEASE* = 100

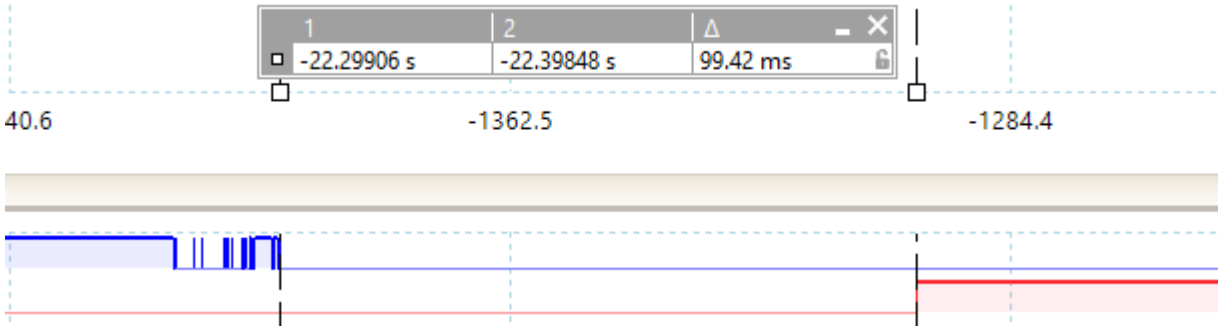

Fig. 20: 100ms debounce for press event. Input bouncing is clearly visible on the diagram. Press event is triggered only after input is stable in active state for minimum time.

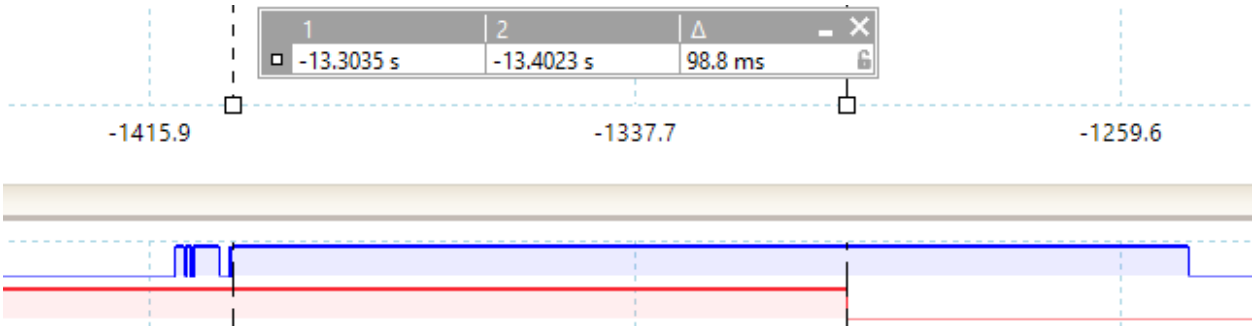

Fig. 21: 100ms debounce for release event. Release event is triggered after input is in stable inactive state for at least release debounce time.

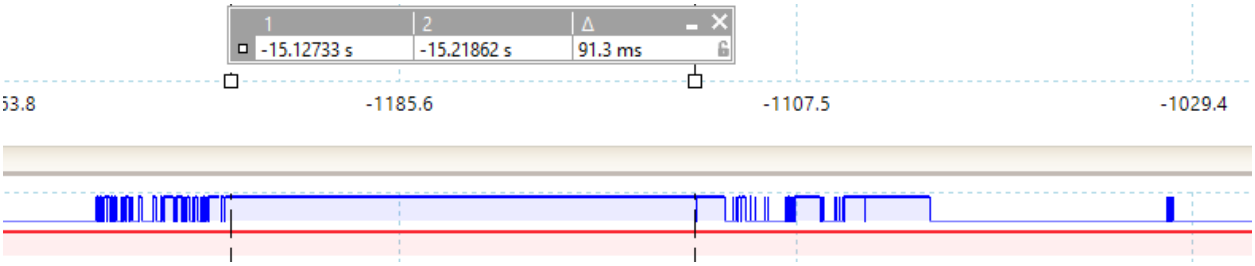

Fig. 22: Input is in *pressed* state (red is high). Blue is in released state for less that minimum stable debounce time, therefore no *release* event has been triggered. This is clearly visible with the *red* line that is staying high for the whole time of the transient period.

5.3 API reference

List of all the modules:

5.3.1 LwBTN

group **LWBTN**

Lightweight button manager.

Defines

lwbtn_init(btns, btns_cnt, get_state_fn, evt_fn) *lwbtn_init_ex*(NULL, btns, btns_cnt, get_state_fn, evt_fn)

Initialize LwBTN library with buttons on default button group.

See also:

lwbtn_init_ex

Parameters

- **btns** – **[in]** Array of buttons to process
- **btns_cnt** – **[in]** Number of buttons to process
- **get_state_fn** – **[in]** Pointer to function providing button state on demand. Can be set to `NULL` if *LWBTN_CFG_GET_STATE_MODE* is NOT set to *LWBTN_GET_STATE_MODE_CALLBACK*
- **evt_fn** – **[in]** Button event function callback

lwbtn_process(mstime) *lwbtn_process_ex*(NULL, mstime)

Periodically read button states and take appropriate actions. It processes the default buttons instance group.

See also:

lwbtn_process_ex

Parameters

- **mstime** – **[in]** Current system time in milliseconds

lwbtn_process_btn(btn, mstime) *lwbtn_process_btn_ex*(NULL, (btn), (mstime))

Process specific button in a default LwBTN instance.

Parameters

- **btn** – **[in]** Button instance to process
- **mstime** – **[in]** Current system time in milliseconds

`lwbtn_keepalive_get_period`(btn) ((btn)->time_keepalive_period)

Get keep alive period for specific button.

Parameters

- **btn** – **[in]** Button instance to get keep alive period for

Returns

Keep alive period in ms

`lwbtn_keepalive_get_count`(btn) ((btn)->keepalive.cnt)

Get actual number of keep alive counts since the last on-press event. It is set to 0 if btn isn't pressed.

See also:

lwbtn_keepalive_get_count_for_time

Parameters

- **btn** – **[in]** Button instance to get keep alive period for

Returns

Number of keep alive events since on-press event

`lwbtn_keepalive_get_count_for_time`(btn, ms_time) ((ms_time) / *lwbtn_keepalive_get_period*(btn))

Get number of keep alive counts for specific required time in milliseconds. It will calculate number of keepalive ticks specific button shall make, before requested time is reached.

Result of the function can be used with *lwbtn_keepalive_get_count* which returns actual number of keep alive counts since last on-press event of the button.

See also:

lwbtn_keepalive_get_count

Note: Value is always integer aligned, with granularity of one keepalive time period

Note: Implemented as macro, as it may be optimized by compiler when static keep alive is used

Parameters

- **btn** – **[in]** Button to use for check
- **ms_time** – **[in]** Time in ms to calculate number of keep alive counts

Returns

Number of keep alive counts

`lwbtn_click_get_count`(btn) ((btn)->click.cnt)

Get number of consecutive click events on a button.

Parameters

- **btn** – **[in]** Button instance to get number of clicks

Returns
Number of consecutive clicks on a button

Typedefs

typedef void (*`lwbtn_evt_fn`)(struct lwbtn *lwobj, struct lwbtn_btn *btn, *lwbtn_evt_t* evt)

Button event function callback prototype.

Param lwobj
[in] LwBTN instance

Param btn
[in] Button instance from array for which event occured

Param evt
[in] Event type

typedef uint8_t (*`lwbtn_get_state_fn`)(struct lwbtn *lwobj, struct lwbtn_btn *btn)

Get button/input state callback function.

Param lwobj
[in] LwBTN instance

Param btn
[in] Button instance from array to read state

Return
`1` when button is considered `active`, `0` otherwise

Enums

enum `lwbtn_evt_t`

List of button events.

Values:

enumerator `LWBTN_EVT_ONPRESS` = 0x00

On press event - sent when valid press is detected (after debounce if enabled)

enumerator `LWBTN_EVT_ONRELEASE`

On release event - sent when valid release event is detected (from active to inactive)

enumerator `LWBTN_EVT_ONCLICK`

On Click event - sent when valid sequence of on-press and on-release events occurs

enumerator `LWBTN_EVT_KEEPALIVE`

Keep alive event - sent periodically when button is active

Functions

uint8_t **lwbtn_init_ex**(*lwbtn_t* *lwobj, *lwbtn_btn_t* *btns, uint16_t btns_cnt, *lwbtn_get_state_fn* get_state_fn, *lwbtn_evt_fn* evt_fn)

Initialize button manager.

Parameters

- **lwobj** – **[in]** LwBTN instance. Set to `NULL` to use default one
- **btns** – **[in]** Array of buttons to process
- **btns_cnt** – **[in]** Number of buttons to process
- **get_state_fn** – **[in]** Pointer to function providing button state on demand. May be set to `NULL` when *LWBTN_CFG_GET_STATE_MODE* is set to manual.
- **evt_fn** – **[in]** Button event function callback

Returns
1 on success, `0` otherwise

uint8_t **lwbtn_process_ex**(*lwbtn_t* *lwobj, uint32_t mstime)

Button processing function, that reads the inputs and makes actions accordingly.

It checks state of all the buttons, linked to the specific LwBTN instance (group).

Parameters

- **lwobj** – **[in]** LwBTN instance. Set to `NULL` to use default one
- **mstime** – **[in]** Current system time in milliseconds

Returns
1 on success, `0` otherwise

uint8_t **lwbtn_process_btn_ex**(*lwbtn_t* *lwobj, *lwbtn_btn_t* *btn, uint32_t mstime)

Process single button instance from the specific LwOBJ instance (group).

This feature can be used if application wants to process the button events only when interrupt hits (as a trigger). It gives user higher autonomy to decide which and when it will call specific button processing.

Parameters

- **lwobj** – **[in]** LwBTN instance. Set to `NULL` to use default one
- **btn** – **[in]** Button object. Must not be set to `NULL`
- **mstime** – **[in]** Current system time in milliseconds

Returns
1 on success, `0` otherwise

uint8_t **lwbtn_set_btn_state**(*lwbtn_btn_t* *btn, uint8_t state)

Set button state to either “active” or “inactive”.

Parameters

- **btn** – **[in]** Button instance
- **state** – **[in]** New button state. `1` is for active (pressed), `0` is for inactive (released).

Returns
1 on success, `0` otherwise

uint8_t **lwbtn_is_btn_active**(const *lwbtn_btn_t* *btn)

Check if button is active. Active is considered when initial debounce period has been a pass. This is the period between on-press and on-release events.

Parameters
btn – **[in]** Button handle to check

Returns
1 if active, 0 otherwise

struct **lwbtn_argdata_port_pin_state_t**

#include <lwbtn.h> Custom user argument data structure.

This is a simple pre-defined structure, that can be used by user to define most commonly required feature in embedded systems, that being GPIO port, GPIO pin and state when button is considered active.

User can later attach this structure as argument to button structure

Public Members

void ***port**

User defined GPIO port information

void ***pin**

User defined GPIO pin information

uint8_t **state**

User defined GPIO state level when considered active

struct **lwbtn_btn_t**

#include <lwbtn.h> Button/input structure.

Public Members

uint16_t **flags**

Private button flags management

uint8_t **curr_state**

Current button state to be processed. It is used to keep track when application manually sets the button state

uint8_t **old_state**

Old button state - 1 means active, 0 means inactive

uint32_t **time_change**

Time in ms when button state got changed last time after valid debounce

uint32_t **time_state_change**

Time in ms when button state got changed last time

uint32_t **last_time**

Time in ms of last send keep alive event

Time in ms of last successfully detected (not sent!) click event

uint16_t **cnt**

Number of keep alive events sent after successful on-press detection. Value is reset after on-release

struct *lwbtn_btn_t*::[anonymous] **keepalive**

Keep alive structure

uint8_t **cnt**

Number of consecutive clicks detected, respecting maximum timeout between clicks

struct *lwbtn_btn_t*::[anonymous] **click**

Click event structure

void ***arg**

User defined custom argument for callback function purpose

uint16_t **time_debounce**

Debounce time in milliseconds

uint16_t **time_debounce_release**

Debounce time in milliseconds for release event

uint16_t **time_click_pressed_min**

Minimum pressed time for valid click event

uint16_t **time_click_pressed_max**

Maximum pressed time for valid click event

uint16_t **time_click_multi_max**

Maximum time between 2 clicks to be considered consecutive click

uint16_t **time_keepalive_period**

Time in ms for periodic keep alive event

uint16_t **max_consecutive**

Max number of consecutive clicks

struct **lwbtn_t**

#include <lwbtn.h> LwBTN group structure.

Public Members

lwbtn_btn_t ***btns**

Pointer to buttons array

uint16_t **btns_cnt**

Number of buttons in array

lwbtn_evt_fn **evt_fn**

Pointer to event function

lwbtn_get_state_fn **get_state_fn**

Pointer to get state function

5.3.2 Configuration

This is the default configuration of the middleware. When any of the settings shall be modified, it shall be done in dedicated application config `lwbtn_opts.h` file.

Note: Check *Getting started* for guidelines on how to create and use configuration file.

group **LWBTN_OPT**

Default configuration setup.

Defines

LWBTN_MEMSET(dst, val, len) memset((dst), (val), (len))

Memory set function.

Note: Function footprint is the same as memset

LWBTN_MEMCPY(dst, src, len) memcpy((dst), (src), (len))

Memory copy function.

Note: Function footprint is the same as memcpy

LWBTN_CFG_USE_KEEPALIVE 1

Enables `1` or disables `0` periodic keep alive events.

Keep alive events are periodically sent to the application, when input is kept in active state.s

Default keep alive period is set with *LWBTN_CFG_TIME_KEEPALIVE_PERIOD* macro

LWBTN_CFG_TIME_DEBOUNCE_PRESS 20

Minimum debounce time for press event in units of milliseconds.

This is the time when the input shall have stable active level to detect valid *onpress* event.

When value is set to > `0`, input must be in active state for at least minimum milliseconds time, before valid *onpress* event is detected.

To be safe not using this feature, external logic must ensure stable transition at input level.

See also:

LWBTN_CFG_TIME_DEBOUNCE_PRESS_DYNAMIC

Note: If value is set to `0`, debounce is not used and *press* event will be triggered immediately when input states goes to *inactive* state.

LWBTN_CFG_TIME_DEBOUNCE_PRESS_DYNAMIC 0

Enables `1` or disables `0` dynamic settable time debounce.

When enabled, additional field is added to button structure to allow each button setting its very own debounce time for press event.

If not used, *LWBTN_CFG_TIME_DEBOUNCE_PRESS* is used as default debouncing configuration

LWBTN_CFG_TIME_DEBOUNCE_RELEASE 0

Minimum debounce time for release event in units of milliseconds.

This is the time when the input shall have minimum stable released level to detect valid *onrelease* event.

This setting can be useful if application wants to protect against unwanted glitches on the line when input is considered "active".

When value is set to > `0`, input must be in inactive low for at least minimum milliseconds time, before valid *onrelease* event is detected

See also:

LWBTN_CFG_TIME_DEBOUNCE_RELEASE_DYNAMIC

Note: If value is set to `0`, debounce is not used and *release* event will be triggered immediately when input states goes to *inactive* state

---

LWBTN_CFG_TIME_DEBOUNCE_RELEASE_DYNAMIC 0

Enables `1` or disables `0` dynamic settable time debounce for release event.

When enabled, additional field is added to button structure to allow each button setting its very own debounce time for release event.

If not used, *LWBTN_CFG_TIME_DEBOUNCE_RELEASE* is used as default debouncing configuration

LWBTN_CFG_TIME_CLICK_MIN 20

Minimum active input time for valid click event, in milliseconds.

Input shall be in active state (after debounce) at least this amount of time to even consider the potential valid click event. Set the value to `0` to disable this feature

See also:

LWBTN_CFG_TIME_CLICK_MIN_DYNAMIC

LWBTN_CFG_TIME_CLICK_MIN_DYNAMIC 0

Enables `1` or disables `0` dynamic settable min time for click.

When enabled, additional field is added to button structure, allowing application to manually set min pressed time for click for each button instance. Default value is set to *LWBTN_CFG_TIME_CLICK_MIN*

LWBTN_CFG_TIME_CLICK_MAX 300

Maximum active input time for valid click event, in milliseconds.

Input shall be pressed at most this amount of time to still trigger valid click. Set to `-1` to allow any time triggering click event.

When input is active for more than the configured time, click even is not detected and is ignored.

See also:

LWBTN_CFG_TIME_CLICK_MAX_DYNAMIC

LWBTN_CFG_TIME_CLICK_MAX_DYNAMIC 0

Enables `1` or disables `0` dynamic settable max time for click.

When enabled, additional field is added to button structure, allowing application to manually set max pressed time for click for each button instance. Default value is set to *LWBTN_CFG_TIME_CLICK_MAX*

LWBTN_CFG_TIME_CLICK_MULTI_MAX 400

Maximum allowed time between last on-release and next valid on-press, to still allow multi-click events, in milliseconds.

This value is also used as a timeout length to send the *onclick* event to application from previously detected valid click events.

If application relies on multi consecutive clicks, this is the max time to allow user to trigger potential new click, or structure will get reset (before sent to user if any clicks have been detected so far)

See also:

LWBTN_CFG_TIME_CLICK_MULTI_MAX_DYNAMIC

LWBTN_CFG_TIME_CLICK_MULTI_MAX_DYNAMIC 0

Enables `1` or disables `0` dynamic settable max time for multi click.

When enabled, additional field is added to button structure, allowing application to manually set max time for multi click for each button instance. Default value is set to *LWBTN_CFG_TIME_CLICK_MULTI_MAX*

LWBTN_CFG_CLICK_MAX_CONSECUTIVE 3

Maximum number of allowed consecutive click events, before structure gets reset to default value.

See also:

LWBTN_CFG_CLICK_MAX_CONSECUTIVE_DYNAMIC

Note: When consecutive value is reached, application will get notification of clicks. This can be executed immediately after last click has been detected, or after standard timeout (unless next on-press has already been detected, then it is send to application just before valid next press event). Configuration can be changed *LWBTN_CFG_CLICK_MAX_CONSECUTIVE_SEND_IMMEDIATELY*

LWBTN_CFG_CLICK_MAX_CONSECUTIVE_DYNAMIC 0

Enables `1` or disables `0` dynamic settable max consecutive clicks.

When enabled, additional field is added to button structure, allowing application to manually set max consecutive clicks for each button instance. Default value is set to *LWBTN_CFG_CLICK_MAX_CONSECUTIVE*

LWBTN_CFG_TIME_KEEPALIVE_PERIOD 100

Keep-alive event period, in milliseconds.

When input is active, keep alive events will be sent through this period of time. First keep alive will be sent *LWBTN_CFG_TIME_KEEPALIVE_PERIOD* ms after input being considered active.

See also:

LWBTN_CFG_TIME_KEEPALIVE_PERIOD_DYNAMIC

LWBTN_CFG_TIME_KEEPALIVE_PERIOD_DYNAMIC 0

Enables `1` or disables `0` dynamic settable keep alive period.

When enabled, additional field is added to button structure, allowing application to manually set keep alive period for each button instance. Default value is set to *LWBTN_CFG_TIME_KEEPALIVE_PERIOD*

LWBTN_CFG_CLICK_MAX_CONSECUTIVE_SEND_IMMEDIATELY 1

Enables `1` or disables `0` immediate onclick event after on-release event, if number of consecutive clicks reaches max value.

When this mode is disabled, onclick is sent in one of 2 cases:

- An on-click timeout occurred
- Next on-press event occurred before timeout expired

LWBTN_GET_STATE_MODE_CALLBACK 0

Callback-only state mode

LWBTN_GET_STATE_MODE_MANUAL 1

Manual-only state mode

LWBTN_GET_STATE_MODE_CALLBACK_OR_MANUAL 2

Callback or manual state mode

LWBTN_CFG_GET_STATE_MODE *LWBTN_GET_STATE_MODE_CALLBACK*

Sets the mode how new button state is acquired.

Different modes are availale, set with the level number:

- `LWBTN_GET_STATE_MODE_CALLBACK`: State of the button is checked through *get state* callback function
- `LWBTN_GET_STATE_MODE_MANUAL`: Only manual state set is enabled. Application must set the button state with API functions. Callback API is not used.
- `LWBTN_GET_STATE_MODE_CALLBACK_OR_MANUAL`: State of the button is checked through *get state* callback function (by default). It enables API to manually set the state with approapriate function call. Button state is checked with the callback at least until manual state API function is called.

  This allows multiple build configurations for various button types

LWBTN_CFG_CLICK_CONSECUTIVE_KEEP_AFTER_SHORT_PRESS 0

Enables `1` or disables `0` keeping the consecutive click event group if last sequence of *onpress* and *onrelease* was too short.

5.4 Changelog

```
# Changelog

## Develop

## v1.1.0

- Add `lwbtn_keepalive_get_period` function
- Add `lwbtn_keepalive_get_count` function to get keepalive count
- Add `lwbtn_click_get_count` function to get consecutive clicks
- Add `lwbtn_keepalive_get_count_for_time` function to calculate number of required keep␣
↪alive tick
- Improve configuration documentation text
- Add `LWBTN_CFG_CLICK_CONSECUTIVE_KEEP_AFTER_SHORT_PRESS` option to report previous␣
↪clicks, even if last sequence of *onpress* and *onrelease* is too short for valid␣
↪click event.

## v1.0.0

- Send `CLICK` event if there is an overlap between max time between clicks and new␣
↪click arrives
- Do not send `CLICK` event if there was previously detected long hold press (hold time␣
```

(continues on next page)

(continued from previous page)

```
↪exceeded max allowed click time)

## v0.0.2

- Add `LWBTN_CFG_GET_STATE_MODE` to control *get state* mode
- Add option to check if button is currently active (after debounce period has elapsed)
- Add option to set time/click parameters at run time for each button specifically
- Rename `_RUNTIME` configuration with `_DYNAMIC`
- Change `LWBTC_CFG_TIME_DEBOUNCE` to `LWBTC_CFG_TIME_DEBOUNCE_PRESS` and `LWBTC_CFG_
↪TIME_DEBOUNCE_RUNTIME` to `LWBTC_CFG_TIME_DEBOUNCE_PRESS_DYNAMIC` respectively
- Add option release debounce with `LWBTC_CFG_TIME_DEBOUNCE_RELEASE` and `LWBTC_CFG_TIME_
↪DEBOUNCE_RELEASE_DYNAMIC` options

## v0.0.1

- First commit
```

INDEX

L