Debounced Buttons

january 31, 2013 by rtos.be   2 comments

Rating: 0.0/**10** (0 votes cast)

A button-driven handbrake

Buttons (or switches, or keys) are quite common in the world of the embedded systems. In most cases buttons trigger a software action and – in our context – we will use buttons for configuring the settings of the charge controller. We can also use the buttons as a debugging utility.

The high-level API is easy:

```
/*! Check if the button is pressed. */
BOOLEAN button_is_pressed(const BUTTON* p_button);
```

Since our buttons are GPIO-based the most basic (but naive!) implementation would be to check the state of the GPIO pin (high or low) and then just return the corresponding boolean value (true or false).

Nothing more useful than a practical example to explain what would be wrong with the above implementation. See the photograph of that ~~very~~ nice car? Well… it's my half-year old Opel Zafira! 🙂

My Opel Zafira

This car actually has a handbrake… which is button-driven!

handbrake release

handbrake pull up

Now it becomes interesting. How does it work? What safety features are built-in?

It would be unacceptable if the handbrake is ‘pulled up’ or ‘released’ accidentally:

1. by an (electronic) glitch, or
2. by a involuntary user click.

In addition:

3. rapid ‘on/off’ behaviour is undesirable.

Before we introduce a solution, it is important to understand that electronic switches ‘bounce’ (i.e. show some erratic behaviour) before they land at a stable state (see for example this article of Jack Ganssle for some nice graphics). This actually means that item 1 and 3 are quite common and cannot be ignored. And because switches bounce, the solution is called ‘debouncing’.

So how does Opel’s ‘debouncer’ look like?

1. To pull up the handbrake, a long press is required (I guess a few hundreds of milliseconds). A human notices a delay if an action takes longer than 50-100ms.
2. To release the handbrake, a short press is required in combination with pushing the brakes of the car.

This means the button behaviour (i.e. the underlying debouncer) is different for each case.

Several debounce strategies are possible. So this is something we should take into account in our design.

Debouncing

Here’s a common debouncing algorithm:

1. each 10 ms the state of a gpio (high or low) is read
2. if the state of the gpio doesn’t change for 5 times (i.e. 50 ms in total) then this state is regarded as stable

This brings us to next API.

```
struct debouncer_settings {
    //! when a state is regarded as debounced
    int required_count_before_debounced;
};

struct debouncer_state {
    BOOLEAN current_state;
    //! how many times the current state is counted
    int count_same_state;
};

struct debouncer {
    const struct debouncer_settings* p_settings;
    struct debouncer_state state;
};

/*! Initialize the debouncer. */
void debouncer_init(struct debouncer* pDebouncer);

/*! Update the debouncer with new value. */
void debouncer_update(struct debouncer* pDebouncer, BOOLEAN val);

/*! Get the debounced value. */
BOOLEAN debouncer_get_val(const struct debouncer* pDebouncer);
```

Notice the strict separation between state and settings: this makes the code explicit and allows you to put settings in ROM (although in this particular case there is no gain).

The debouncer can be integrated in a higher level object (such as a button). This object then must update the debouncer (*debouncer_update()*) regularly e.g. each 10 ms. In order to get the current debounced value one can use *debouncer_get_val()*.

GPIO Button abstraction

The aforementioned debouncer can be used in the button data structure (see API). Since the button and the debouncer are loosely coupled, different types of buttons can apply different debouncing strategies, which is exactly what we want (think on the handbrake example). Application developers can work at a high level with the button abstraction and don't really have to care about the low-level handling.

```
typedef struct button {
    //! processor pin (will be configured as gpio input)
    const PIN pin;
    //! optional debouncer (if NULL then no debouncing)
    struct debouncer* const p_debouncer;
} BUTTON;

/*!
* Initialize a button.
* The pin is configured as gpio input,
* debouncer (if not NULL) is initialized.
*/
void button_init(const BUTTON* p_button);

/*!
* Update the button:
* read new pin state and update internal state.
*/
void button_update(const BUTTON* p_button);

/*!
* Check if the button is pressed.
* If the button is configured with a debouncer
* (i.e. p_debouncer!=NULL) then the button is debounced.
*/
BOOLEAN button_is_pressed(const BUTTON* p_button);
```

button_init() and *button_update()* respectively initialize and update the underlying debouncer.

Finally, similar to the LED example, a set of buttons can be grouped in a table (see example below).

All interfaces can be found (and downloaded) here:

Button and button group

Debouncer

Full example… feel the abstraction!

Step 1: initialization

```
/* setup debouncer */
static const struct debouncer_settings debouncer_setts = {
.required_count_before_debounced = 5
};

static struct debouncer debouncer_button_user = {
.p_settings = &debouncer_setts
};

/* setup table with buttons */
#define BUTTON_HANDBRAKE_PULLUP 0
#define BUTTON_HANDBRAKE_RELEASE 1
static const BUTTON button_table[] = {
{ .pin = PA0, .p_debouncer = &debouncer_button_user },
{ .pin = PB6, .p_debouncer = &debouncer_button_user },
BUTTON_TABLE_END
};

/* finally initialize everything */
button_group_init(button_table);
```

Step 2: running system

```
/* call this each 10 ms e.g. in a thread */
button_group_update(button_table);

/* finally, to know the debounced value... */
button_is_pressed(&button_table[BUTTON_HANDBRAKE_RELEASE]);
```

Conclusion

*For an application developer the focus is on configuration (**what you want**) and not on implementation (**how to do it**)*

Share

Rating: 0.0/**10** (0 votes cast)

Related Posts

- Coming Soon
- The embedded hierarchy — Part 1
- Reverse engineering the bug of my Opel Zafira's volume knob (with movie!)
- Driving LEDs by GPIO: finally resolved!
- MCU pin configuration, GPIOs and a word on software architecture

Share / Save …

filed under: energy harvester project, interfaces, software design and architecture tagged with: debouncing, gpio, switches

About rtos.be

rtos.be is a joined initiative from Gert Boddaert and Pieter Beyens.

More info: about us, our mission, contact us.

Comments

Fernando says
February 1, 2013 at 00:31

Hello, I have a cuestion regarding the method described.
You are decoupling the debounce algorithm from the hardware, but you are using a polling method to test the switch state from the application, that generates some coupling between modules. Why not to use a publish-subscribe system?

Regards,
Fernando

Reply

rtos.be says
February 3, 2013 at 15:23

To be concise, the described method is chosen because it adheres to the KISS principle.

To elaborate on it, two issues are raised: 'Polling' and 'coupling'.
To retrieve a (debounced) button state on a pin, multiple strategies are possible.
A few we can mention are…

1) polling: it is simple, and when used in a prioritized RTOS task it is very deterministic, however it could ‘waste’ processor time and power.
2) gpio interrupt: the device is e.g. in a (deep) sleep mode and awakes on a pin triggered interrupt, debouncing is done by a task or by a timer (task or interrupt), which makes this strategy better suited for low (or battery) power(ed) devices. It is possible to use this strategy in real-time applications but it is less common to do so. Safety critical system also usually have some aversion to do these kind of things in interrupt, ‘polling’ is the more common method (or debouncing is done in hardware).

Debounced button state is maintained by polling, and the state is available on request.

We don’t see the choice for polling as a cause for less or more ‘coupling’ or ‘cohesion’.
(low-end) embedded systems are typically very static. We know ‘at compile time’ how our board looks like (how many buttons, leds etc.) and how to configure it. In this particular case, software framework designs with static behaviour are preferred instead of dynamic ones like publish-subscribe, since ‘compile-time-static’ algorithms typically need less code and use less RAM (we will publish an article about this topic).
Furthermore, the goal of the code is to decouple the *debouncing strategy* from the button module (one might look at this as a simplified form of the strategy design pattern). But indeed, they are (loosely) coupled and that is because buttons and debouncing go hand-in-hand on our (and many other) embedded system application(s).

Reply