Integration manual - Power optimization

TOSL

Release 4.6.4 (doc Pa/L/Ls/T/Ts)

Jul 18, 2018

Authored by *Trusted Objects*

Contents

1. Introduction

When sending commands to Secure Element, you may want to switch on MCU standby mode while waiting the response availability, in order to optimize power consumption. This guide will help you to optimize your MCU power consumption when working with Trusted Objects Secure Element, and explains how to integrate this in your client application relying on libTO.

2. Wake up methods

Two methods are detailed below, choose the more suitable for your project.

2.1 First wake up method: use Secure Element status PIO

Note: This is the recommended method.

Note: To use this method, Secure Element status PIO feature must be enabled in your Secure Element release.

The Secure Element PIO pin can be used to be notified about the following states:

- IDLE: the Secure Element is ready to receive a new command
- BUSY: a command is currently processed by the Secure Element
- READY: a command response is ready to be read

IDLE can be signalled either the same way as ready, else by status PIO high impedance.

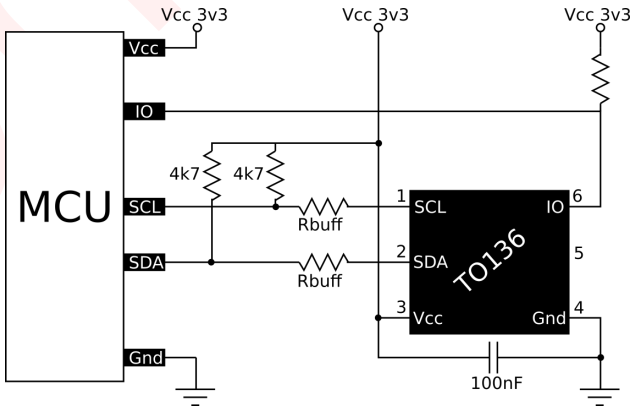
We assume below the Secure Element status PIO is connected to a PIO of your MCU on which interrupts can be configured to wake up from standby on state change.

2.1.1 Signalling method: open drain or push pull

The PIO level can be signalled by open drain or by push pull.

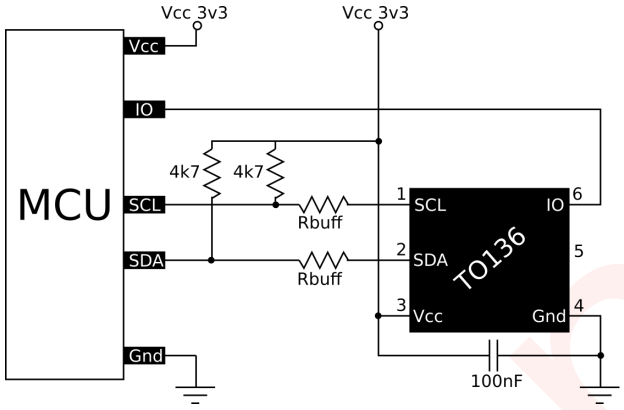
2.1.1.1 TO136 wiring for the open drain method

This method is suitable if you have several devices able to wake up your MCU using the same PIO.



2.1.1.2 TO136 wiring for the push pull method

This method is suitable if TO136 is the only device able to wake up your MCU using this PIO, because it avoids the TO136 open drain internal pull up power consumption.



2.1.2 Status PIO settings

Include the libTO header in your code:

```
#include <TO.h>
```

and initialize the Secure Element I2C bus with `TO_init()`.

The status PIO behavior can be customized to fit your needs through the following libTO API:

int **`TO_set_status_PIO_config`**(int *enable*, int *opendrain*, int *ready_level*, int *idle_hz*)

Configure Secure Element status PIO notification behavior.

The configuration is stored permanently by the Secure Element, and then persists across reboots.

Parameters

- `enable`: Set to 1 to enable status PIO notifications (default: 1)
- `opendrain`: Set to 1 for open drain, 0 for push pull (default: 1)
- `ready_level`: Set to 1 to signal readyness with high PIO level, 0 to signal it with low PIO level (default: 1).
- `idle_hz`: Set to 1 to have idle state signalled by PIO high impedance signal it with a low level (default: 1)

Note: this function do not have BUSY / READY states, the PIO remains in the IDLE state when called. But if the pushed settings change the PIO levels or signalling method, the PIO state can change when this function is called.

Return

- TORSP_SUCCESS on success
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element

- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_ERROR: generic error

The open drain signalling method is interesting to be used if you plan to have several devices able to wake up your MCU using the same PIO. In the other hand, push pull method is interesting if the Secure Element is the only device able to wake up the MCU, because the internal pull up resistor is disabled and then consomes no power.

The READY signalling level allows you to choose if you want to wake up on a rising or a falling edge.

You can check the current settings by calling the following function:

int **`TO_get_status_PIO_config`**(int * *enable*, int * *opendrain*, int * *ready_level*, int * *idle_hz*)

Return Secure Element status PIO notification configuration.

Note: this function do not have BUSY / READY states, the PIO remains in the IDLE state when called.

Parameters

- `enable`: Set to 1 if status PIO notification enabled
- `opendrain`: Method to signal level, see *TO_set_status_PIO_config()*
- `ready_level`: PIO level to signal ready state, see *TO_set_status_PIO_config()*
- `idle_hz`: Idle state signalled by PIO high impedance, see *TO_set_status_PIO_config()*

Return

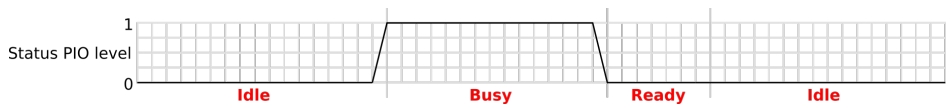
- TORSP_SUCCESS on success
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_ERROR: generic error

2.1.2.1 Example diagrams accoding to settings

Open drain, ready level high, idle high impedance:

Push pull, ready level low, idle high impedance:

Push pull, ready level low:



2.2 Second wake up method: scheduled time wake up

Note: This method is to be considered only if you have not the possibility to use *First wake up method: use Secure Element status PIO*.

This method consists to enable the MCU standby mode, having scheduled a wake up time on an interrupt based timer. The standby duration can be sized according to current Secure Element working command, and to its data size.

See *Appendix A: commands BUSY duration* to have an estimated duration of the BUSY state for some significative commands.

3. Implement the choosen wake up method

The libTO provides hooks to call your client application code automatically at particular library internal steps. The interesting hooks here are PRE COMMAND and POST WRITE, which are called respectively just before sending a command and just after a command has been written to the Secure Element.

You have to define functions having the right hook prototype, and then declare your hooks to libTO. Refer to libTO hook APIs for more details.

You can also see to the *hook.c* example provided with the library to have an overview of a generic implementation.

So the idea is, in your hook functions implementation, to switch on your MCU standby mode according to one of the *Wake up methods*, and then to wake up according to the expected event, then the Secure Element response will be read from the I2C bus by libTO.

Warning: Do NOT call any libTO function from inside PRE COMMAND or POST WRITE hooks

3.1 Wake up with status PIO

Just before sending a command, use the PRE COMMAND hook to prepare PIO wake up interrupt. The POST WRITE hook will be used just after writing the command and before reading the response, to sleep until it is available to be read.

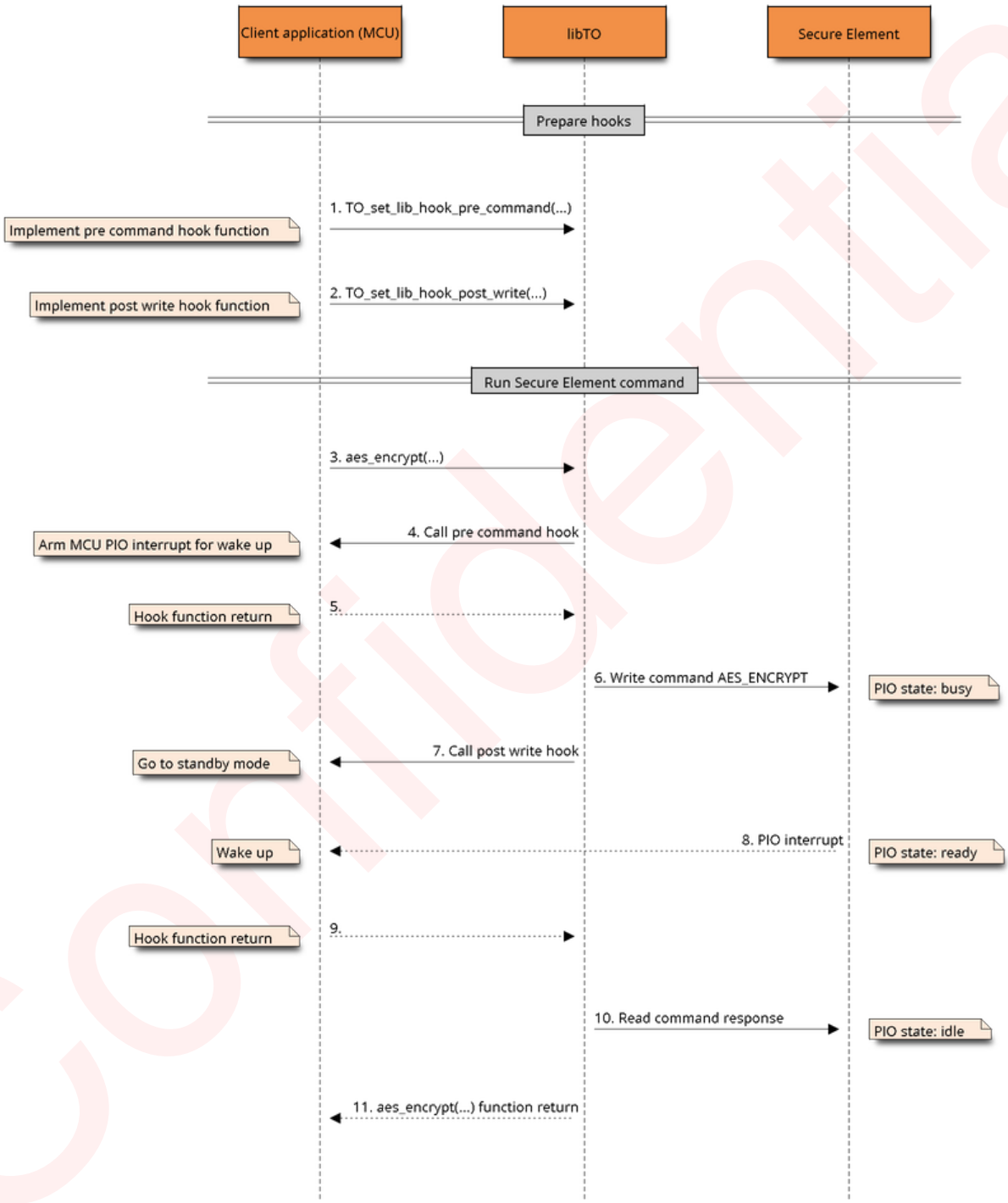
```
#include <TO.h>
#include <TO_cmd.h>

void my_pre_command_hook(uint16_t cmd, uint16_t cmd_data_len)
{
        // TODO: arm standby wake up mechanism, here you prepare the
        // wake up interrupt to be sure to do not miss the event after
        // going to standby.
}

void my_post_write_hook(uint16_t cmd, uint16_t cmd_data_len)
{
        // TODO: go to standby from this function, and return on wakeup
        // to allow libTO to read the Secure Element response.
}

/* Declare these hooks to libTO */
TO_set_lib_hook_pre_command(my_pre_command_hook);
TO_set_lib_hook_post_write(my_post_write_hook);
```

3.1.1 Sequence diagram for such use case



3.2 Wake up with timer

Just after writing the command to the Secure Element, go to standby, and wait a timer interrupt to wake up and let libTO read the response.

```
#include <TO.h>
#include <TO_cmd.h>

/* Your hook which will be called just after writing command */
void my_post_write_hook(uint16_t cmd, uint16_t cmd_data_len)
{
        // TODO: arm the timer interrupt with a delay sized according
        // to 'cmd' and 'cmd_data_len'.
        // TODO: go to standby mode.
        // TODO: return function on wake up.
}

/* Declare this hook to libTO */
TO_set_lib_hook_post_write(my_post_write_hook);
```

See *Appendix A: commands BUSY duration* to have an estimated duration of the BUSY state for some significative commands.

4. Appendix A: commands BUSY duration

Below you can find BUSY time estimations (milliseconds) for some Secure Element commands, with different data sizes.

Note: These value are informative and given with no guarantee, they are subject to change with Secure Element versions.

4.1 Timings with TO136 Secure Element

4.1.1 aes_encrypt() / aes_decrypt()

Data size	16	32	64	128	256	512
BUSY duration (ms)	4	5	7	11	18	34

4.1.2 compute_cmac() / verify_cmac()

Data size	16	32	64	128	256	512
BUSY duration (ms)	8	12	19	35	66	127

4.1.3 compute_hmac() / verify_hmac()

Data size	16	32	64	128	256	512
BUSY duration (ms)	14	14	14	15	16	19

4.1.4 secure_message()

Data size	16	32	64	128	256	512
BUSY duration (ms)	18	19	21	25	35	53

4.1.5 unsecure_message()

Data size	16	32	64	128	256	512
BUSY duration (ms)	20	21	24	28	38	57

4.1.6 sha256()

Data size	16	32	64	128	256	512
BUSY duration (ms)	0.7	0.8	1.5	2	3	6

4.1.7 sign()

Data size	16	32	64	128	256	512
BUSY duration (ms)	1080	1080	1080	1080	1080	1080

4.1.8 verify()

Data size	16	32	64	128	256	512
BUSY duration (ms)	2040	2040	2040	2040	2040	2040

Index

T