SBC Microcontroller Library

Lite SBC or MidRange+ SBC Family

About this document

Scope and purpose

This document describes the configuration, setup and usage of the Infineon SBC Microcontroller Library.

Intended audience

This document is intended for software and hardware engineers integrating an Infineon Lite SBC or MidRange+ SBC Family device into their application.

Table of contents

1 Introduction

This document describes the configuration, setup and usage of the Infineon SBC Microcontroller Library and is intended for software and hardware engineers integrating an Infineon Lite SBC or MidRange+ SBC Family device into their application.

Some code examples are presented as Arduino code as the low-level API is easy to understand and known by most embedded software engineers.

The library code is distributed under the Boost Software License, Version 1.0 from August 17th, 2013.

For more details see the full license text inside library files.

The code is mostly in compliance with MISRA-C: 2016.

2 Infineon Toolbox and Config Wizard Setup

The source code of the library is not provided as direct download as the library files must be generated first by the user with a graphical user interface (following called “Config Wizard”).

Go to the **download page** of the Infineon Toolbox and install the Toolbox. After successful installation klick on “Manage tools”, search for “Config Wizard for SBC” and press install.

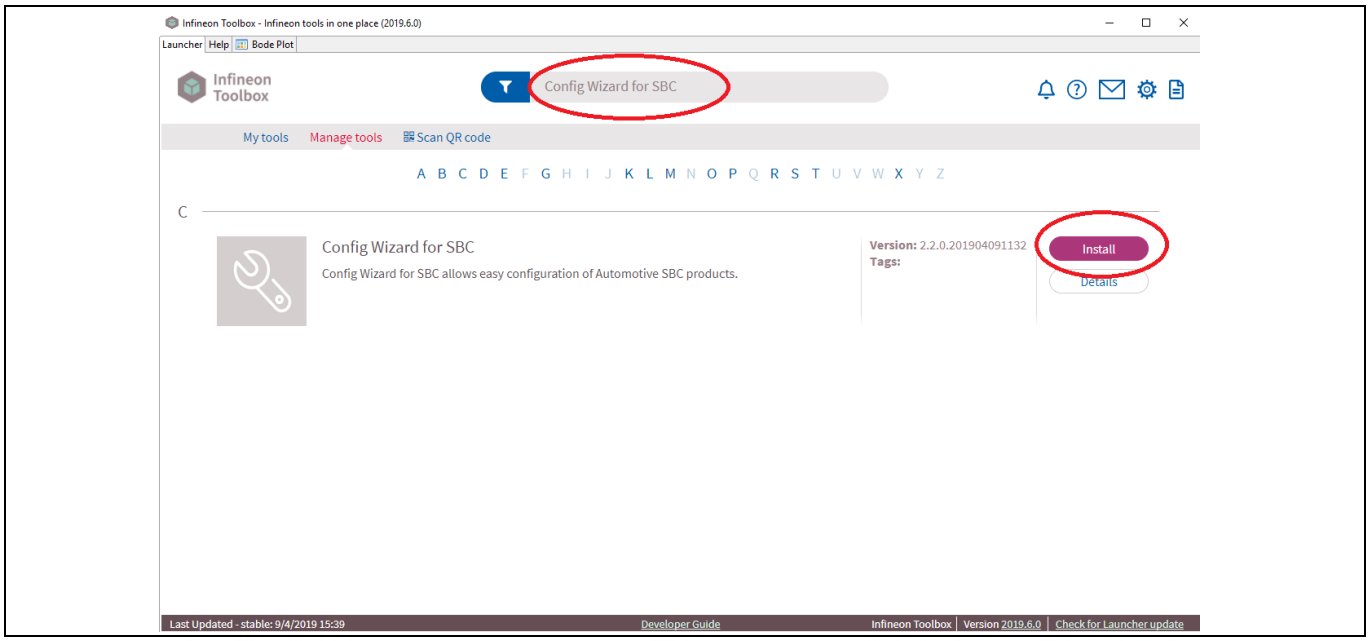


Figure 1 Infineon toolbox

Open the respective “C Lib” plugin inside the “Config Wizard for SBC” for your device.

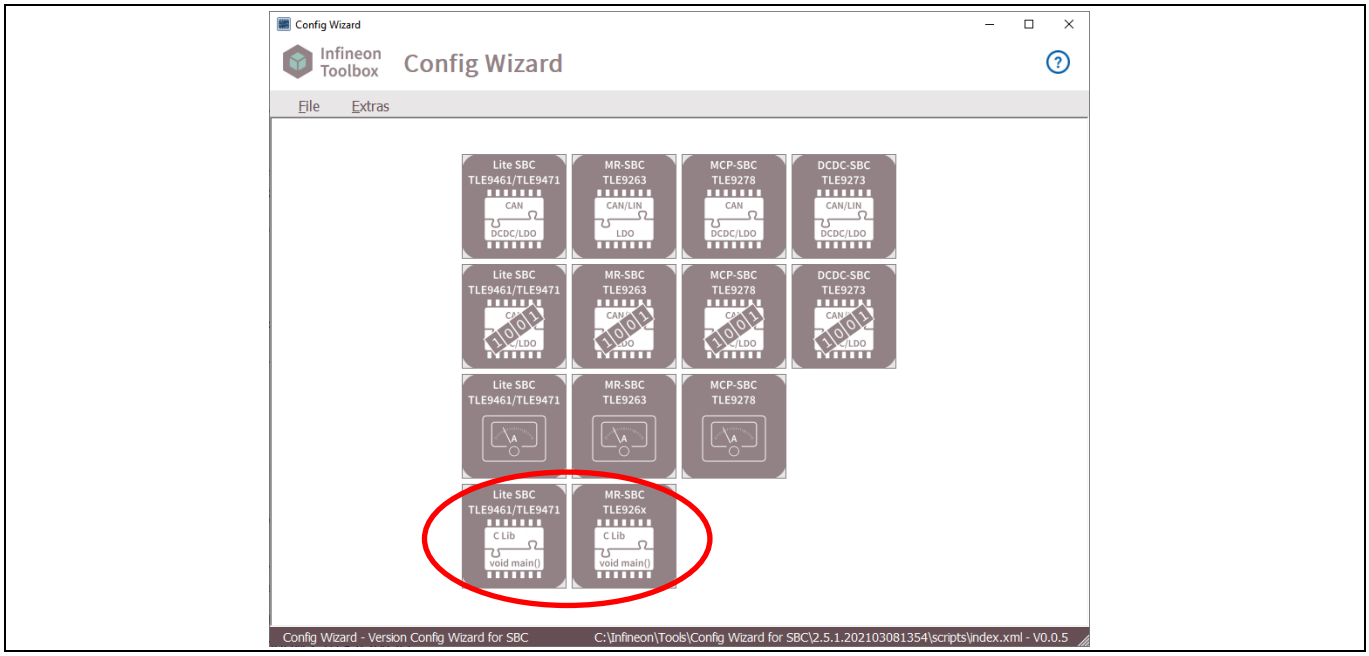


Figure 2 Start screen of Config Wizard for SBC

All settings regarding the SBC can now be setup via the user interface (UI).

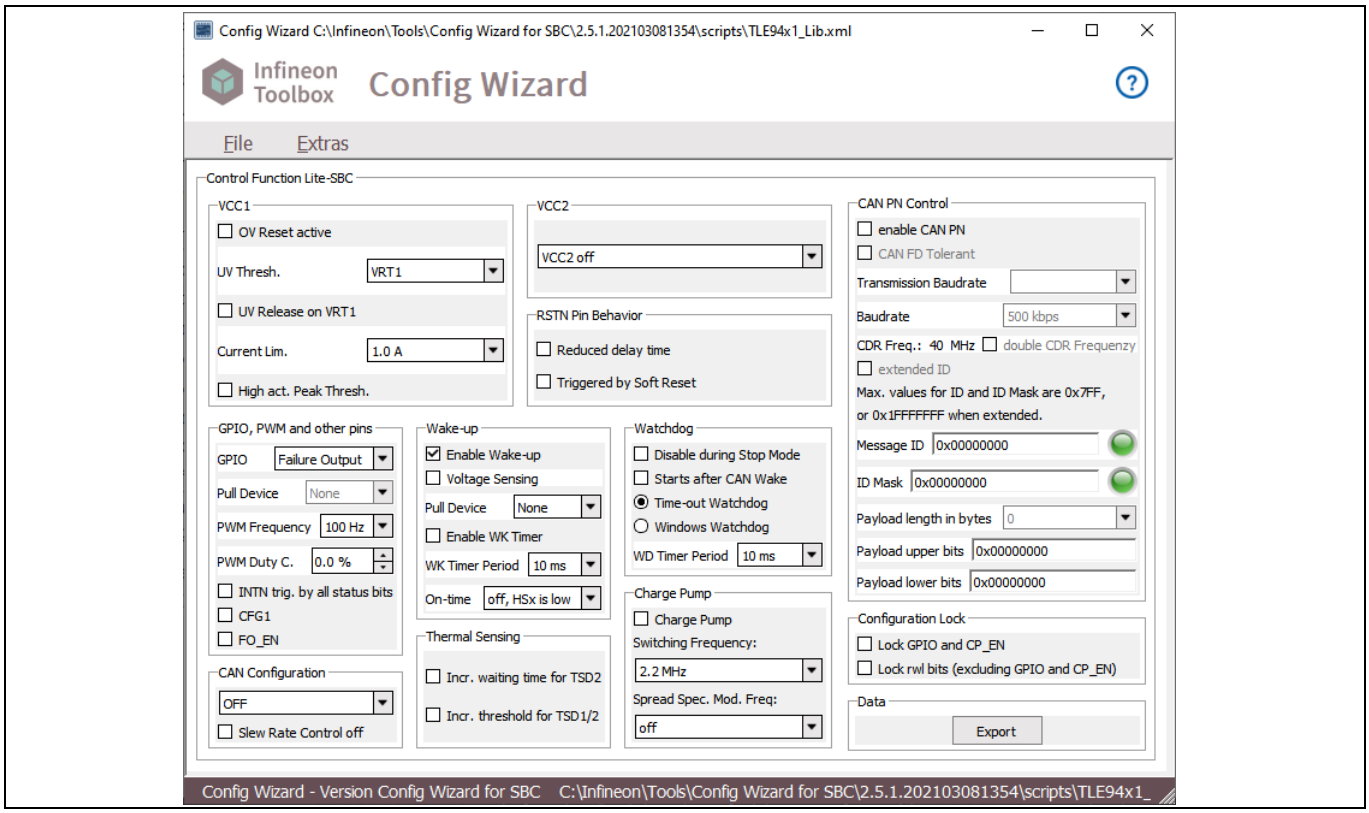


Figure 3 UI for Lite SBC

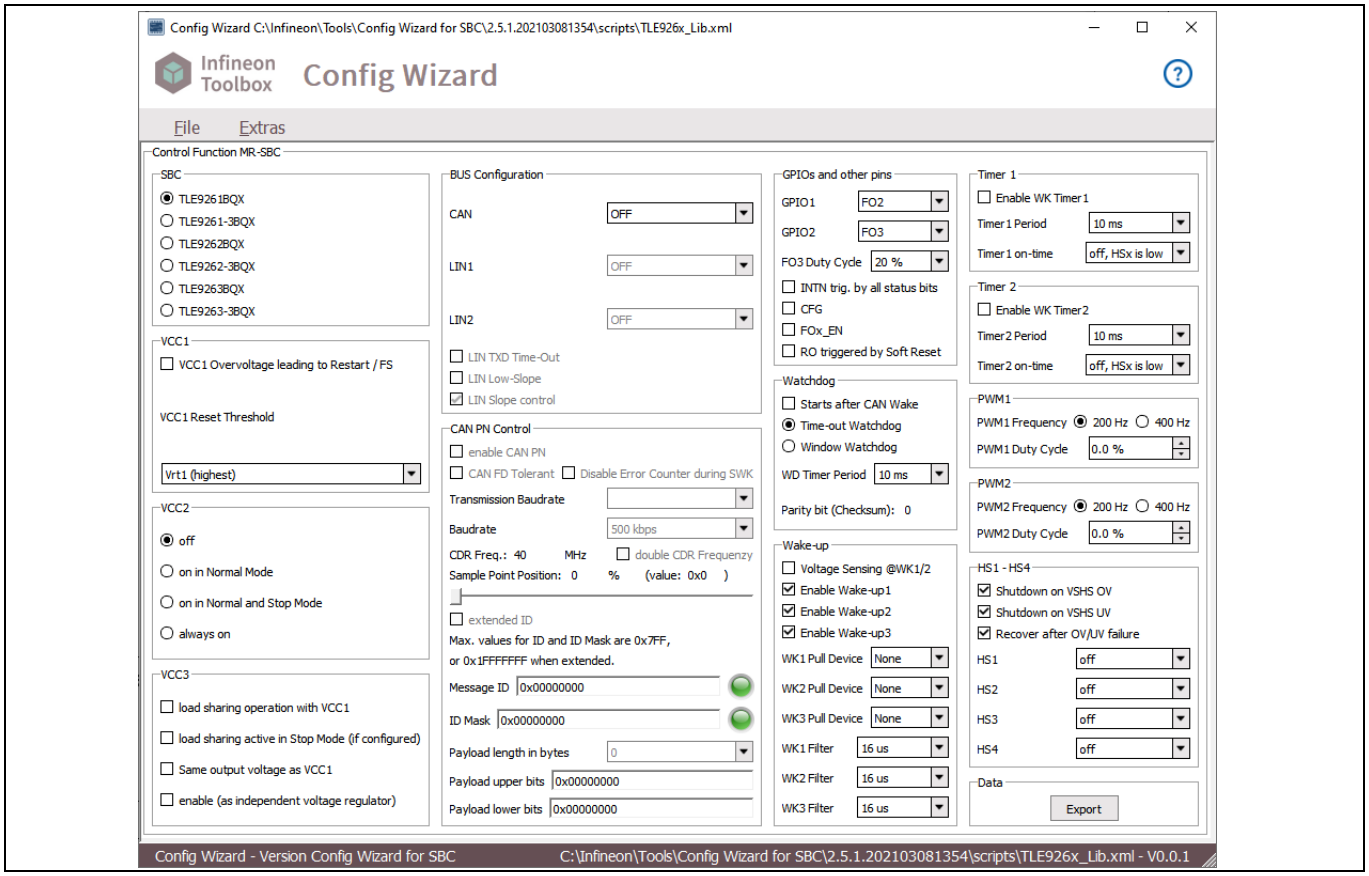


Figure 4 UI for MidRange+ SBC

For more details regarding the dedicated hardware modules of the SBC and their configuration, please have a look into the datasheet or to the user manual which can be found on **http://infineon.com/SBC**.

3 Configuration of CAN Partial Networking

If a “-3” variant of the SBC is used, it is possible to make use of the CAN partial networking functionality to wake up from SBC sleep mode with a dedicated CAN message = Wake-up frame (WUF). This can be configured within the “CAN PN Control” section of the UI. All related settings to clock data recovery are done automatically in the background.

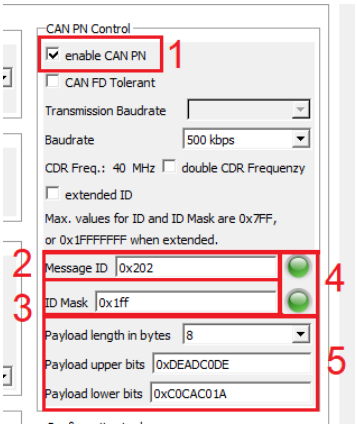
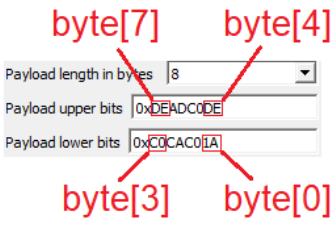
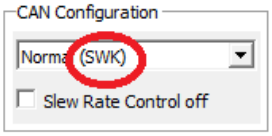


Figure 5 CAN PN Control section of Lite SBC UI

1. Enable CAN PN.
2. Enter the desired message ID to be sensitive on.
3. The integrated CAN protocol handler will do a logic AND operation between ID mask and the message ID. For every ‘1’ bit configured in the mask, the CAN protocol handler will be later sensitive to the corresponding bit of the message ID. This gives the chance to be sensitive on multiple message IDs. If the protocol handler should be only sensitive to one message ID, 0x7FF can be used as mask for non-extended ID message or 0x1FFFFFFF for extended ID messages.
4. This is an indication if the configured values are valid. If the indication turns into red color, the ID or the mask is not possible to be configured.
5. The message payload length and the payload content of the CAN frame can be configured here. Please consider the order of the payload bytes.

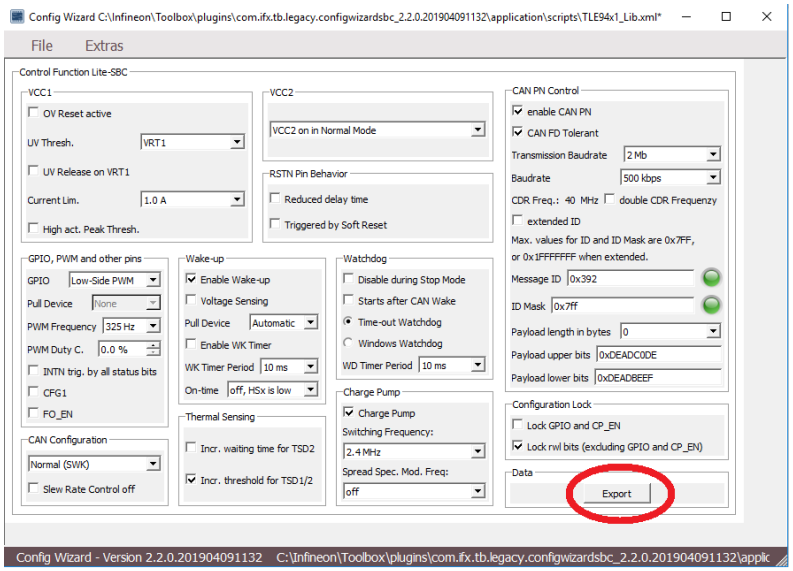


6. Please make sure the CAN configuration was set a CAN mode including SWK.

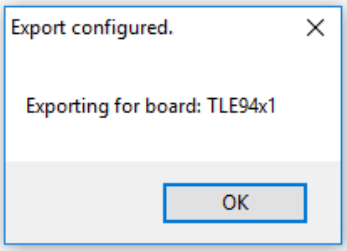


4 Generating library files and integrating into microcontroller project

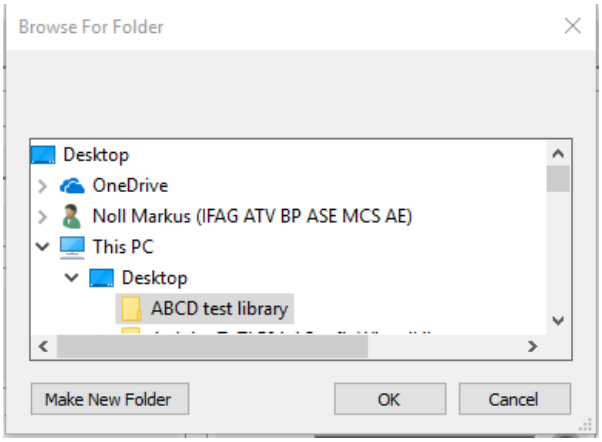
After configuring all settings, press “Export”.



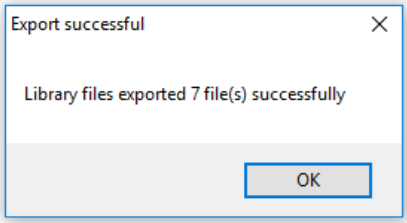
An information message will pop up.



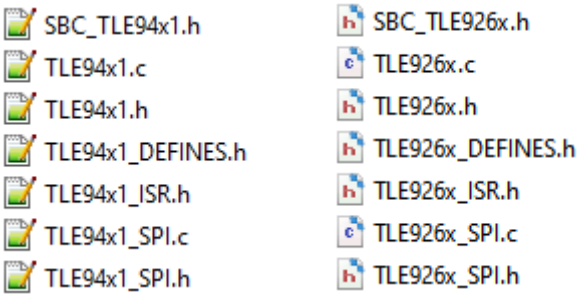
An export location can be chosen now.



After the export, it will show an acknowledgment message.



Files will be exported to your chosen directory:



The code structure at Lite SBC and MidRange+ SBC is similar. The left files named TLE94x1 are generated for LiteSBC. The files on the right side named TLE926x are generated for MidRange+ SBC.
Throughout the document the files are called TLE9xxx to cover both devices.

4.1 SBC_TLE9xxx.h

This is a header file containing all the configuration which were configured inside the Config Wizard.

Every device register available inside the SBC gets its own `#define` with the start-up value.

During initialization phase of the library, this value will be written to the SBC. In case, a new startup configuration is needed in the microcontroller project, it is enough just to replace the SBC_TLE9xxx.h later on.

4.2 TLE9xxx.h / TLE9xxx.c

Those are the main library files containing all important functions like `SBC_Init()` for startup, handling system interrupts of the SBC, `SBC_WD_Trigger()` to trigger the watchdog and a couple of helper functions for reading and manipulating registers and dedicated bit fields inside the registers.

But there are also functions provided which might be useful to use during runtime (e.g. control SBC modes, switch on/off charge pump, control timers/PWM, lock/unlock system configuration and also write and read the system status registers).

4.3 TLE9xxx_SPI.h / TLE9xxx_SPI.c

Those files are necessary for the SPI communication to the SBC.

`SBC_SPI_INIT()` and `SBC_SPI_TRANSFER16()` inside the .c file are not implemented yet as the SPI is dependent on the microcontroller and its configuration.

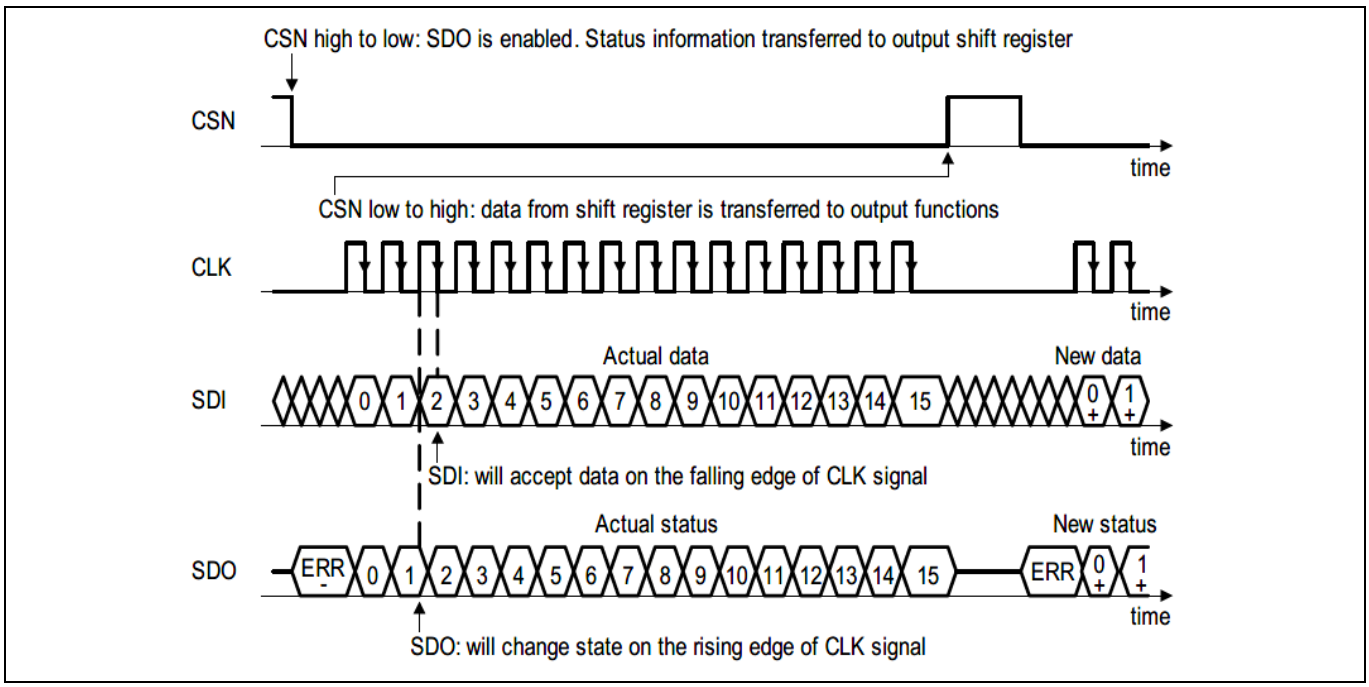
Note: The SPI functions need to be implemented by the user.

```
/* ================================================================================ */
/* ============================   SPI Functions     ============================= */
/* ================================================================================ */

/**
 * @brief   IMPORTANT! THIS METHOD HAS TO BE DEFINED BY THE USER
 *
 *          The function has to initialze the SPI of the uC and will be called once during SBC_Init().
 *          In case, the SPI hardware is already initialized by some other code before, it can be left blank.
 *
 * @retval  Method has to return 0 if initialization was successful.
 */
uint8_t SBC_SPI_INIT(void);

/**
 * @brief   IMPORTANT! THIS METHOD HAS TO BE DEFINED BY THE USER
 *
 *          The function will be called by the library everytime when a SPI communication is needed.
 *          The function proceeds a bidirectional 16-bit transfer to/from the SBC  .
 *          As some UCs only supports 8-Bit transfers, the input arguments are split in two 8-bit arguments.
 *          For further implementation details have a look at datasheet chapter 13.1 or at the Arduino-examples.
 *
 * @param   Upper   The first 8 bit to transmit to the SBC.
 * @param   Lower   The second 8 bit to transmit to the SBC.
 * @retval  The function will return all 16 bits received from the SBC.
 *          Bit[15:8] are the first 8 bits received (Status-Information-Field).
 *          Bit[7:0] is the data-field transmitted of the SBC.
 */
uint16_t SBC_SPI_TRANSFER16(uint8_t Upper, uint8_t Lower);
```

`SBC_SPI_INIT()` is called from the library itself one time at startup and should include the code for initialization of the SPI hardware.

The `SBC_SPI_TRANSFER16()` function implements a single 16-bit SPI transfer between the SBC and the microcontroller which is called by the library every time a communication is needed. The function is intended to be a blocking function and should return when the whole SPI transfer is finished. The two arguments provided by the library are delivering the first and the second 8 bit to transfer to the SBC.

Please consider the polarity and phase of the clock as shown in the picture below. The SPI interface on the SBC expects LSB first bit-order.



A possible implementation for Arduino as orientation could look like following example (considering pin 8 as CSN).

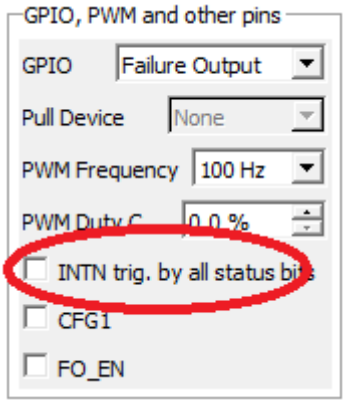
```
#include "TLE94x1_SPI.h"
#include <Arduino.h>
#include <SPI.h>

int8_t SBC_SPI_INIT(void) {
    pinMode(8, OUTPUT);
    digitalWrite(8, HIGH);
    SPI.begin();
    return 0;
}

uint16_t SBC_SPI_TRANSFER16(uint8_t Upper, uint8_t Lower) {
    uint16_t outdata = 0;
    SPI.beginTransaction(SPISettings(1000000, LSBFIRST, SPI_MODE1));
    digitalWrite(8, LOW);
    outdata = (SPI.transfer(Upper) << 8);
    outdata |= SPI.transfer(Lower);
    SPI.endTransaction();
    digitalWrite(8, HIGH);
    return outdata;
}
```

4.4 TLE9xxx_DEFINES.h

This file includes definitions for all register addresses, masks and bit positions for all bit fields and all possible enumerations of bit field values.

4.5 TLE9xxx_ISR.h

This file includes definitions for all possible interrupts, which can be handled by the library. If INT_GLOBAL is configured to '1', all status registers will generate interrupts. Otherwise, be aware that only WK_STAT_0 and WK_STAT_1 (respectively for MidRange+ SBC: WK_STAT_1 and WK_STAT_2) will generate interrupts. Make sure you configured "INTN trig. by all status bits" according to the needs of the application inside Config Wizard.



Additional Information for using the interrupt service

The library provides an interrupt service handler routine. The user can link self-defined functions with a dedicated ISR vector provided by the TLE9xxx_ISR.h.

Every time the registered event occurs, the library will call back the user defined function.

The user has to link the external interrupt pin of the SBC to the `SBC_ISR()` function of the library. Every time a rising edge of the INTN pin is detected, the `SBC_ISR()` function of the library should be called.

With the `SBC_Register_Callback()` function, the user can link a self-defined function to a specific event. Every time the event occurs, the library will call back the user function. The user function must implement one `uint8_t` parameter. The content of the related status register which generated the interrupt will be passed to the user function as the `uint8_t` parameter.

It is recommended to set a flag every time the external interrupt event occurs and call the `SBC_ISR()` inside the main software-loop every time the flag was set.

An external interrupt calling directly the `SBC_ISR()` method could lead to disturbing possible other SPI communication in case the SPI is in use when the interrupt is triggered. In this case, a safety mechanism has to be implemented by the user to guarantee the `SBC_ISR()` is only called when the SPI communication module is free to be used.

A possible implementation for Arduino could look like following example below (showing how to handle interrupt generated by toggle event at WK input):

```
#include “TLE94x1.h”

bool irqSBC = false;

void irqSBC_Handler() {
        /* remember IRQ flag signalized by SBC */
        irqSBC = true;
}

void irqSBC_Wake_Handler(uint8_t value) {

        /* will be called if event on WK input occured */
        /* ‘value’ variable will contain content of the WK_STAT_0 register when called */
}

void setup() {

        SBC_Init();

        /* handle interrupts from SBC by rising edge on INTN */
        attachInterrupt(1, irqSBC_Handler, RISING);

        /* register handler routine for SBC_ISR_WK_WU */
        SBC_Register_Callback(SBC_ISR_WK_WU, irqSBC_Wake_Handler);
}

void loop() {

    /* trigger watchdog */
    SBC_WD_Trigger();

    /* Handle SBC_ISR() if INTN was toggeled */
    if (irqSBC) {

      SBC_ISR();
      irqSBC = false;

    }
}
```

5 Startup of the library and handling errors

After successful modification of the TLE9xxx_SPI.c and setting up the handling of the `SBC_ISR()` routine by external interrupt pin, the library is ready to use.

In general only one `SBC_Init()` call at startup of the microcontroller is needed to initialize the SBC.

This method will start with a watchdog trigger and will initialize all register configuration as defined in the Config Wizard UI. All write commands will be verified by a readout of the corresponding register.

In case a write event during initialization failed, an error code will be returned. Also the `SBC_SPI_INIT()` will be called by the `SBC_Init()` function. This has not to be done manually by the user.

Afterwards, depending on the configuration of the SBC (test mode active or not), the watchdog has to be triggered regularly depending on the timing configuration by calling `SBC_WD_Trigger()`.

All functions in the library which are sending write commands to the SBC in some way will verify the written value by a readback and will return an error code based on a C-struct which is defined in the TLE9xxx.h

```
typedef struct __SBC_ErrorCode {
    uint8_t SBC_Register;       //!< The register where an error occurred.
    uint8_t flippedBitsMask;    //!< Masks the bits that differ from the expected value. Is 0 if readout is as
                                     expected.
    uint8_t expectedValue;      //!< Expected readout of the register.
} SBC_ErrorCode;
```

- **SBC_Register** is the register address where the error occurred.
- **flippedBitsMask** is a logical XOR between the original value to be written and the read back value out of the register. If no error occurred, this field is 0.
- **expectedValue** is the original value intended to be written.

The user can (but doesn't have to) check against errors. However it is recommended.
See example below how to use:

```
SBC_ErrorCode err = SBC_Init();

    If (err.flippedBitsMask != 0) {

       /* Initialization not successfull. See failed register with err.SBC_Register and wrong written bits with
        err.flippedBitsMask */

    } else {

        /* Initialization was successful */

    }
```

6 Additional functionalities provided by the library

6.1 SBC mode control

The library offers functions to control the SBC modes, meaning SBC normal-mode, SBC stop-mode or also SBC sleep-mode.

This can be managed by just calling SBC_Mode_Normal() or SBC_Mode_Stop().

In case of calling SBC_Mode_Sleep(), all possible wake sources which could lead directly to a wake up of the SBC once again are cleared before entering sleep mode.

If the selective wake module was configured for partial networking, all necessary steps to initiate sleep-mode with partial networking will be handled automatically.

Attention: ***For entering successfully SBC sleep-mode with SWK enabled, the CAN protocol handler must be in sync to the CAN bus before entering sleep-mode. Otherwise, the SBC_Mode_Sleep() function will return with an error and sleep-mode is not entered. Please make sure that at least one CAN frame (which is not the magic SWK frame) was received before and check if the SWK module is in sync before entering sleep-mode. See code example below.***

```
/* wait until SWK protocol handler is in sync */
while ( SBC_Read_RegField(SBC_SWK_STAT, SBC_SWK_STAT_SYNC_Msk, SBC_SWK_STAT_SYNC_Pos) != SBC_SYNC_VALID_FRAME_RECEIVED) {}

/* enter sleep mode */
SBC_ErrorCode err = SBC_Mode_Sleep();

if (err.flippedBitsMask != 0) {

        /* entering sleep mode failed – can occure e.g. if the magic SWK frame was sent before entering sleep mode */

} else {

        /* SBC is in sleep-mode now... this piece of code is ideally never reached as the microcontroller is not powered
           anymore */

}
```

6.2 Handling custom write / read commands

The library offers several functions to manipulate and to read register contents with following functions:

- **SBC_Read_Command()** reads the content of a dedicated register address.
- **SBC_Read_RegField()** reads the content of a dedicated bit field inside a register. The user must provide register address, bit field-mask and bit field-position (definitions can be found inside TLE94x1_DEFINES.h).
- **SBC_Write_Reg()** writes data to specific register address. It will also verify the content and provide an error code to the user. Additionally the old content of the register can be read.
- **SBC_Write_RegField()** can be used for manipulating a single bit field inside a register. The values of the other bit fields inside the register will remain. An error code and the old register value can be read.

See code example below how to use the write and read functions.

```
/* read full register content of GPIO_CTRL register */

uint8_t data_gpio_ctrl = SBC_Read_Command(SBC_GPIO_CTRL);

/* ------------------------------------------------------ */

/* read spread spectrum modulation content */

uint8_t config_spread_spectrum = SBC_Read_RegField(SBC_HW_CTRL_2, SBC_HW_CTRL_2_SS_MOD_FR_Msk, SBC_HW_CTRL_2_SS_MOD_FR_Pos);


/* ------------------------------------------------------ */

/* set SMPS frequency to 2.4 MHz and remain value of other bitfields */

uint16_t oldval = 0;

SBC_ErrorCode err = SBC_Write_RegField(SBC_HW_CTRL_2, SBC_HW_CTRL_2_2MHZ_FREQ_Msk, SBC_HW_CTRL_2_2MHZ_FREQ_Pos, SBC_2MHZ_FREQ_2_4_MHZ, &oldval);

if (err.flippedBitsMask != 0) {

          /* error while writing */
} else {

          /* write successful, old register value available in 'oldval' [7:0] */
}


/* ------------------------------------------------------ */

/* write full register content - write 127 to PWM_Ctrl = 50% duty cycle */

uint16_t old_pwm_dc = 0;
SBC_ErrorCode err = SBC_Write_Reg(SBC_PWM_CTRL, 127, &old_pwm_dc);

if (err.flippedBitsMask != 0) {

          /* error while writing */
} else {

          /* write successful, old PWM duty cycle value available in old_pwm_dc [7:0] */
}
```

6.3 Read and write system status

The SBC has an integrated 16 bit register which can store user data while the SBC is e.g. in sleep-mode (and the microcontroller is not supplied). After a SBC restart (and a fresh startup of the microcontroller), the data written before can be read back. This can be done with the `SBC_SYS_STAT_Write()` and `SBC_SYS_STAT_Read()` functions.

```
/* write data (0xABCD) to the system status register */

uint16_t data_to_write = 0xABCD;
SBC_ErrorCode err = SBC_SYS_STAT_Write(data_to_write)

if (err.flippedBitsMask != 0) {
        /* error while writing */
} else {

        /* write successful */
}

/* ------------------------------------------------------ */

/* read system status data */

uint16_t sys_status_data = SBC_SYS_STAT_Read();

/* 'sys_status_data* will return 0xABCD */
```

6.4 Further library features

The library also features further methods to control hardware related blocks like the fail-output, integrated charge-pump, PWM generator and voltage sensing via WK input.

Those methods can be called during runtime and will overwrite the initial setting of the corresponding hardware modules configured in SBC_TLE9xxx.h by Config Wizard.

```
/* Switch charge-pump on */

SBC_ErrorCode err = SBC_CP_On();

if (err.flippedBitsMask != 0) {
        /* error while writing */
} else {

        /* CP is enabled */
}
```

An overview of all further library functions can be found in the library files.

A Doxygen documentation is available for LiteSBC library. See link below. It may be usefull also for MidRange+ library as both are very similar.

7 Additional information

For further information you may contact **http://www.infineon.com/SBC** or your regional FAE.

Further code examples based on the LiteDCDC SBC Shield for Arduino and a Doxygen documentation for Lite SBC library can be found on **https://github.com/Infineon/SBC-for-Arduino/**.

Revision history

Document version	Date of release	Description of changes
1.1	2022-02-23	Updated document to cover LiteSBC and MidRange+ library
1.0	2019-04-12	Initial release.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2022-02-23

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

Z8F65358716

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie") .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.