

SmartRF06EB Board Support Package

Texas Instruments CC2538 Family of Products

User's Guide




Document License

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License (CC BY-ND 3.0). To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/legalcode> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Copyright

Copyright © 2013 Texas Instruments Incorporated. All rights reserved. CC2538 and SmartRF are registered trademarks of Texas Instruments. ARM and Thumb are registered trademarks and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

Texas Instruments
Post Office Box 655303
Dallas, TX 75265
<http://www.ti.com>



Revision Information

User guide literature numbers from Texas Instruments RF Products start with *SWRU*. The document revision is indicated by a letter suffix after the literature number. The initial version of a document does not have a letter suffix (for example *SWRU321*). The first revision is suffixed *A*, the second *B*, and so on (for example *SWRU321B*). The literature number for this document is in the document footer.

This document was updated on April 11, 2013 (build 9717).

Table of Contents

Document License	2
Copyright	2
Revision Information	2
1 Introduction	5
2 Using the SmartRF06EB BSP	7
2.1 Introduction	7
2.2 Using the BSP as a precompiled library	8
2.3 Using the BSP as Source Files	9
2.4 Configuring and Recompiling the BSP Library	9
3 BSP Base Functions	11
3.1 Introduction	11
3.2 API Functions	11
3.3 Programming Example	15
4 I/O Pin Interrupt Handler	17
4.1 Introduction	17
4.2 API Functions	17
4.3 Programming Example	18
5 LEDs	21
5.1 Introduction	21
5.2 API Functions	21
5.3 Programming Example	22
6 Keys	25
6.1 Introduction	25
6.2 API Functions	25
6.3 Programming Example	28
7 Accelerometer	31
7.1 Introduction	31
7.2 API Functions	31
7.3 Programming Example	35
8 Ambient Light Sensor	37
8.1 Introduction	37
8.2 API Functions	37
8.3 Programming Example	38
9 LCD	39
9.1 Introduction	39
9.2 API Functions	39
9.3 Programming Example	54
10 Micro SD Card Reader	55
10.1 Introduction	55
10.2 API Functions	55
10.3 Programming Example	58
11 UART Backchannel	59
11.1 Introduction	59
11.2 API Functions	59
11.3 Programming Example	63

12 **References** 65

13 **Document History** 67

1 Introduction

The SmartRF06EB Board Support Package (BSP) for CC2538 from Texas Instruments is a set of drivers for accessing the peripherals found on the SmartRF06EB with the CC2538 family of ARM® Cortex™-M based devices.

The SmartRF06EB BSP uses the CC2538 peripheral driver library ([driverlib](#)).

While the SmartRF06EB BSP drivers are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the SmartRF06EB and its peripherals.

The capabilities and organization of the drivers are governed by the following design goals:

- They are written entirely in C language except where absolutely not possible.
- They demonstrate how to use the peripheral in its common mode of operation.
- They are easy to understand.
- They are reasonably efficient in terms of memory and processor usage.
- They are as self-contained as possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.
- They can be built with more than one tool chain.

Some consequences of these design goals are:

- The drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly language and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.
- The drivers do not support the full capabilities of the hardware. Some of the peripherals provide complex capabilities which cannot be used by the drivers in this library, though the existing code can be used as a reference upon which to add support for the additional capabilities.

For many applications, the drivers can be used as is. But in some cases, the drivers must to be enhanced or rewritten to meet the functionality, memory, or processing requirements of the application. If so, the existing driver can be used as a reference on how to operate the peripheral.

The SmartRF06EB BSP is available for all devices in the CC2538 family.

The following tool chains are supported:

- IAR Embedded Workbench® (IAR)
- TI Code Composer Studio™ (CCS)

Source Code Overview

A brief overview of the organization of the SmartRF06EB Board Support Package library source code follows. All paths in this section are given relative to the `bsp/srf06eb_cc2538` folder.

<code>examples/</code>	This directory holds SmartRF06EB BSP examples.
<code>drivers/bin/</code>	This directory holds the precompiled library files for different IDEs.
<code>drivers/source/</code>	This directory holds the source code for the drivers, including header files.
<code>drivers/projects/</code>	This directory holds the IDE project files for compiling the library files.

Trademark Attribution

- ARM® – ARM Physical IP, Inc.
- Code Composer Studio™ – Texas Instruments
- Cortex™-M3 – ARM Limited
- I²C™ – Philips Semiconductor Corp
- IAR Embedded Workbench® – IAR Systems
- SPI™ – Motorola

2 Using the SmartRF06EB BSP

Introduction	7
Using the BSP as a precompiled library	8
Using the BSP as source files	9
Configuring and recompiling the BSP library	9

2.1 Introduction

The SmartRF06EB BSP for CC2538 family can be used as a library, `bsp.lib`, or by including the `.c` and `.h` source files directly into your project. The following sections will go through how to use the SmartRF06EB Board Support Package as a library, and directly from source files, respectively.

The SmartRF06EB BSP uses the CC2538 peripheral driver library to access the CC2538 internal peripheral modules. Therefore, the CC2538 peripheral driver library must also be included in projects using the SmartRF06EB BSP. See Chapter 12 for more information.

The SmartRF06EB BSP for the CC2538 family is released under a standard 3-clause BSD license.

Copyright (C) 2013 Texas Instruments Incorporated - <http://www.ti.com/>

- Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of Texas Instruments Incorporated nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2.2 Using the BSP as a precompiled library

The SmartRF06EB BSP comes as a precompiled library file, `bsp.lib`. The `bsp.lib` library file is in the IDE subfolder of `bsp/srf06eb_cc2538/drivers/bin`.

2.2.1 IAR Embedded Workbench

All paths in this section are given relative to the `bsp/srf06eb_cc2538` folder. The following steps have been tested using IAR EWARM version 6.40.

The predefined IAR variable `$PROJ_DIR$`, which gives the absolute path of the `.ewp` file of the project is very handy when defining include paths and library paths.

To use the precompiled `bsp.lib` in IAR Embedded Workbench for ARM, the project must be set up with the correct include paths. In IAR, the include paths are set under *Project > Options > C/C++ Compiler > Preprocessor*.

- `drivers/source` Path to BSP API definitions
- `../../driverlib/cc2538/source` Path to driverlib API definitions
- `../../driverlib/cc2538/inc` Path to CC2538 register name definitions

The project must be configured to use the correct libraries. This can be set under *Project > Options > Linker > Library*.

- `drivers/bin/iar/bsp.lib`
- `../../driverlib/cc2538/bin/iar/driverlib.lib`

In the application source file, include the header files containing the API functions necessary for the application; for example:

```
#include <bsp.h>           // Base API and board defines
#include <bsp_led.h>       // LED API
```

2.2.2 Code Composer Studio

All paths in this section are given relative to the `bsp/srf06eb_cc2538` folder. The following steps have been tested using CCS release 5.2.0.

The predefined CCS variable `${ProjDirPath}`, which gives the absolute path of the project, is very handy when defining include paths and library paths.

To use the precompiled `bsp.lib` in Code Composer Studio, the project must be set up with the correct include paths. In CCS, the include paths are set under *Project > Properties > CCS Build > ARM Compiler > Include Options*.

- `drivers/source` Path to BSP API definitions
- `../../driverlib/cc2538/source` Path to driverlib API definitions
- `../../driverlib/cc2538/inc` Path to CC2538 register name definitions

The project must be configured to use the correct libraries. This can be set under *Project > Properties > CCS Build > ARM Linker > File Search Path*. Add `bsp.lib` and `driverlib.lib` under *Include library file or command file as input*.

Add the following directories under *Add <dir> to library search path*

- `drivers/bin/ccs`
- `../../driverlib/cc2538/bin/ccs`

In the application source file, include the header files containing the API functions necessary for the application; for example:

```
#include <bsp.h>           // Base API and board defines
#include <bsp_led.h>        // LED API
```

2.3 Using the BSP as Source Files

All paths in this section are given relative to the `bsp/srf06eb_cc2538` folder. To use the BSP in a project, the project must be set up with the correct include paths. The necessary include paths are as follows:

- `drivers/source` Path to BSP API definitions
- `../../driverlib/cc2538/source` Path to driverlib API definitions
- `../../driverlib/cc2538/inc` Path to CC2538 register name definitions

Source files are included to the IAR or CCS project by selecting *Project > Add files*

The SmartRF06EB BSP uses the CC2538 peripheral driver library. To include the precompiled CC2538 peripheral driver library file, `driverlib.lib`, to the project, follow the steps in [Section 2.2](#).

To include the CC2538 peripheral driver library source files to the project, add the `.c` files in `../../driverlib/cc2538/source` listed above.

2.4 Configuring and Recompiling the BSP Library

The IDE projects for building the SmartRF06EB BSP library file, `bsp.lib`, are found in the IDE subfolder under `srf06eb_cc2538/drivers/projects`. In the same IDE folder, there are configuration files, `bsp_*.cfg`, for configuring the BSP library.

To configure which drivers are included in `bsp.lib`, first alter the `bsp_*.cfg` configuration file to suit your needs and then recompile the BSP library project.

3 BSP Base Functions

Introduction	11
API Functions	11
Programming Example	15

3.1 Introduction

The SmartRF06EB BSP Base functions provide a set of functions for initializing the CC2538 and SmartRF06EB for operation, configuring the SPI interface to the SmartRF06EB peripherals, and controlling the SmartRF06EB 3.3-V domain.

The SmartRF06EB Board Support Package base module source files are contained in `bsp/srf06eb_cc2538/drivers`.

- `source/bsp.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/bsp.h` contains the API definitions for use by applications.

3.2 API Functions

Functions

- void [bsp3V3DomainDisable](#) (void)
- void [bsp3V3DomainDisableForced](#) (void)
- void [bsp3V3DomainEnable](#) (void)
- uint8_t [bsp3V3DomainEnabled](#) (void)
- void [bspAssert](#) (void)
- void [bspInit](#) (uint32_t ui32SysClockSpeed)
- uint32_t [bspSpiClockSpeedGet](#) (void)
- void [bspSpiClockSpeedSet](#) (uint32_t ui32ClockSpeed)
- void [bspSpiInit](#) (uint32_t ui32SpiClockSpeed)

3.2.1 Detailed Description

The SmartRF06EB BSP base API is broken into three main groups of functions:

- Those that initialize the CC2538 I/O for use
- Those that deal with the SPI interface to the SmartRF06EB peripherals
- Those that deal with the SmartRF06EB 3.3-V domain

Function [bspInit\(\)](#) configures the CC2538 main clock and its I/O for operation on the SmartRF06EB. [bspInit\(\)](#) should be the first function called when using the SmartRF06EB BSP.

The following functions are used for configuring the SPI interface between the SmartRF06EB peripherals and the CC2538:

- [bspSpilnit\(\)](#)
- [bspSpiClockSpeedSet\(\)](#)
- [bspSpiClockSpeedGet\(\)](#)

The following functions control the 3.3-V domain on the SmartRF06EB ([LCD](#) and [SD Card Reader](#)):

- [bsp3V3DomainEnable\(\)](#)
- [bsp3V3DomainDisable\(\)](#)
- [bsp3V3DomainDisableForced\(\)](#)
- [bsp3V3DomainEnabled\(\)](#)

Function [bspAssert\(\)](#) is provided as a utility function.

3.2.2 Function Documentation

3.2.2.1 bsp3V3DomainDisable

This function disables the 3.3-V domain on SmartRF06EB. This function is "soft" and only disables the 3.3-V domain if counter variable *i8Bsp3V3DomainEnableCount* is 1 or 0. The function assumes the 3.3-V domain enable pin is configured as output by, for example, [bsplnit\(\)](#).

Prototype:

```
void  
bsp3V3DomainDisable(void)
```

Description:

This function decrements *i8Bsp3V3DomainEnableCount* and disables the 3.3-V domain if *i8Bsp3V3DomainEnableCount* is less than or equal to 0. If *i8Bsp3V3DomainEnableCount* is greater than 0 after decrement, function [bsp3V3DomainEnable\(\)](#) has been called more times than this function, and the 3.3-V domain will not be disabled. To disable the 3.3-V domain irrespective of the value of *i8Bsp3V3DomainEnableCount*, use [bsp3V3DomainDisableForced\(\)](#).

See also:

[bsp3V3DomainEnable\(\)](#), [bsp3V3DomainDisableForced\(\)](#)

Returns:

None

3.2.2.2 bsp3V3DomainDisableForced

This function disables the 3.3-V domain on SmartRF06EB. The function assumes the 3.3-V domain enable pin is configured as output by, for example, [bsplnit\(\)](#). The 3.3-V domain needs approximately 400 us to fall below 0.5 V.

Prototype:

```
void  
bsp3V3DomainDisableForced(void)
```

See also:

[bsp3V3DomainEnable\(\)](#), [bsp3V3DomainDisable\(\)](#)

Returns:

None

3.2.2.3 void bsp3V3DomainEnable (void)

This function enables the 3.3-V domain on SmartRF06EB. The LCD and SD card reader are powered by the 3.3-V domain. This function increments a counter variable each time it is called. The function assumes the 3.3-V domain enable pin is configured as output by, for example, [bspInit\(\)](#). The 3.3-V domain needs up to approximately 400 us to settle when enabled.

See also:[bsp3V3DomainDisable\(\)](#), [bsp3V3DomainDisableForced\(\)](#)**Returns:**

None

3.2.2.4 uint8_t bsp3V3DomainEnabled (void)

This function returns the current state of the 3.3-V domain.

Returns:

Returns 1 if 3.3-V domain is enabled

Returns 0 if 3.3-V domain is disabled

3.2.2.5 void bspAssert (void)

Assert function. Eternal loop that blinks all LEDs quickly. Function assumes LEDs to be initialized by, for example, [bspInit\(\)](#).

Returns:

None

3.2.2.6 void bsplnit (uint32_t ui32SysClockSpeed)

This function initializes the CC2538 clocks and I/O for use on SmartRF06EB.

The function assumes that an external crystal oscillator is available to the CC2538. The CC2538 system clock is set to the frequency given by input argument *ui32SysClockSpeed*. The I/O system clock is set configured to the same value as the system clock.

If the value of *ui32SysClockSpeed* is invalid, the system clock is set to the highest allowed value.

Parameters:

ui32SysClockSpeed is the system clock speed in Hz; it must be one of the following:

- **SYS_CTRL_32MHZ**
- **SYS_CTRL_16MHZ**
- **SYS_CTRL_8MHZ**

- **SYS_CTRL_4MHZ**
- **SYS_CTRL_2MHZ**
- **SYS_CTRL_1MHZ**
- **SYS_CTRL_500KHZ**
- **SYS_CTRL_250KHZ**

Returns:

None

3.2.2.7 uint32_t bspSpiClockSpeedGet (void)

This function returns the clock speed of the BSP SPI interface. It is assumed that the BSP SPI SSI module runs off the I/O clock.

Returns:

Returns the SPI clock speed in Hz

3.2.2.8 void bspSpiClockSpeedSet (uint32_t *ui32ClockSpeed*)

This function configures the SPI interface to the given clock speed, Motorola mode with clock idle high and data valid on the second (rising) edge. For proper SPI function, the SPI interface must first be initialized using [bspSpiInit\(\)](#).

Warning:

Limitations apply to the allowed values of *ui32ClockSpeed*. Please refer to device's driverlib documentation.

Parameters:

ui32ClockSpeed is the SPI clock speed in Hz

Returns:

None

3.2.2.9 void bspSpiInit (uint32_t *ui32SpiClockSpeed*)

This function initializes SPI interface. The SPI is configured to Motorola mode with clock idle high and data valid on the second (rising) edge. The SSI module uses the I/O clock as clock source (I/O clock frequency set in [bspInit\(\)](#)).

Input argument *ui32SpiClockSpeed* must obey the following criteria:

- *ui32SpiClockSpeed* = $\text{srcClk} / 2^n$ where n is integer, $n \geq 2$, and srcClk is the clock frequency set by [bspInit\(\)](#).

Parameters:

ui32SpiClockSpeed is the SPI clock speed in Hz

Returns:

None

3.3 Programming Example

Software examples for the SmartRF06EB BSP are in `bsp/srf06eb_cc2538/examples`.

The following example initializes the CC2538 to its default clock speed and configures the necessary CC2538 I/O. The CC2538 SPI interface to the SmartRF06EB SPI peripherals is initialized.

```
#include "bsp.h"

//
// Initialize the cc2538 clock and srf06eb I/O
//
bspInit(BSP_SYS_CLK_SPD);

//
// Initialize the SPI interface to its default speed
//
bspSpiInit(BSP_SPI_CLK_SPD);
```


4 I/O Pin Interrupt Handler

Introduction	17
API Functions	17
Programming Example	18

4.1 Introduction

The SmartRF06EB BSP includes an I/O pin interrupt handler. The I/O pin interrupt handler is an extension to functionality in the CC2538 peripheral driver library, allowing GPIO pins on the same GPIO port to have different interrupt handlers.

The I/O pin interrupt handler registers a generic interrupt service routine (ISR) to the interrupt vector of the GPIO port. The generic ISR calls the appropriate interrupt handler for each GPIO pin.

The driver files are in `bsp/srf06eb_cc2538/drivers`.

- `source/io_pin_int.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/io_pin_int.h` contains the API definitions for use by applications.

4.2 API Functions

Functions

- void [ioPinIntRegister](#) (uint32_t ui32Base, uint8_t ui8Pins, void (*pfnIntHandler)(void))
- void [ioPinIntUnregister](#) (uint32_t ui32Base, uint8_t ui8Pins)

4.2.1 Detailed Description

The I/O pin interrupt handler has two functions, [ioPinIntRegister\(\)](#) and [ioPinIntUnregister\(\)](#).

The I/O pin interrupt handler module may be excluded from the SmartRF06EB BSP by defining **IO_PIN_INT_EXCLUDE**.

Warning:

Define **IO_PIN_INT_EXCLUDED** should be used with care as other SmartRF06EB BSP modules use the I/O pin interrupt handler. For more information on how to configure the SmartRF06EB BSP for CC2538 precompiled library, see Section [2.4](#).

4.2.2 Function Documentation

4.2.2.1 ioPinIntRegister

Register an interrupt handler to the GPIO pin (or pins) specified by bitmask *ui8Pins* on GPIO port given by *ui32Base*. This function registers a general ISR to the GPIO port and then assigns the ISR specified by *pfnIntHandler* to the given pins.

Prototype:

```
void  
ioPinIntRegister(uint32_t ui32Base,  
                 uint8_t ui8Pins,  
                 void (*pfnIntHandler)(void))
```

Parameters:

ui32Base is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin (or pins).

pfnIntHandler is a pointer to the interrupt handler function.

Returns:

None

4.2.2.2 void ioPinIntUnregister (uint32_t ui32Base, uint8_t ui8Pins)

Unregister the interrupt handler to GPIO pin (or pins) specified by bitmask *ui8Pins* on GPIO port *ui32Base*.

Parameters:

ui32Base is the base address of the GPIO port.

ui8Pins is the bit-packed representation of the pin (or pins).

Returns:

None

4.3 Programming Example

The following code example shows how to register function `myIsr()` as the interrupt handler for rising edge interrupts on GPIO port A pin 3. For examples using the I/O pin interrupt handler, see bsp/srf06eb_cc2538/examples/keys.

```
//  
// Assuming interrupts are disabled  
//  
  
//  
// Register interrupt handler myIsr() to GPIO port A pin 3  
//  
ioPinIntRegister(GPIO_A_BASE, GPIO_PIN_3, &myIsr);
```

```
//  
// Set interrupt type to rising edge (driverlib function)  
//  
GPIOIntTypeSet(GPIO_A_BASE, GPIO_PIN_3, GPIO_RISING_EDGE);  
  
//  
// Enable pin interrupt (driverlib function)  
//  
GPIOPinIntEnable(GPIO_A_BASE, GPIO_PIN_3);  
  
//  
// Enable master interrupt (driverlib function)  
//  
IntMasterEnable();
```


5 LEDs

Introduction	21
API Functions	21
Programming Example	22

5.1 Introduction

The SmartRF06EB has 4 LEDs that can be controlled from the CC2538. The SmartRF06EB BSP LED driver provides functionality for setting, clearing, and toggling these LEDs. The LEDs can be accessed using defines **BSP_LED_1** through **BSP_LED_4**. Define **BSP_LED_ALL** is an ORed bitmask of all LEDs on the SmartRF06EB accessible from the CC2538.

The driver files are located in `bsp/srf06eb_cc2538/drivers`.

- `source/bsp_led.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/bsp_led.h` contains the API definitions for use by applications.

5.2 API Functions

Functions

- void `bspLedClear` (uint8_t ui8Leds)
- void `bspLedInit` (void)
- void `bspLedSet` (uint8_t ui8Leds)
- void `bspLedToggle` (uint8_t ui8Leds)

5.2.1 Detailed Description

The functionality found in `bspLedInit()` is also performed in the BSP initialization function, `bspInit()`. It is therefore not necessary to call `bspLedInit()` if `bspInit()` has already been called.

5.2.2 Function Documentation

5.2.2.1 bspLedClear

This function clears LED(s) specified by *ui8Leds*. This function assumes that LED pins have been initialized by, for example, `bspLedInit()`.

Prototype:

```
void  
bspLedClear(uint8_t ui8Leds)
```

Parameters:

ui8Leds is an ORed bitmask of LEDs (for example **BSP_LED_1**).

Returns:

None

5.2.2.2 void bspLedInit (void)

This function initializes GPIO pins connected to LEDs. LEDs are initialized to be off. The function [bspInit\(\)](#) does the same LED initialization as this function.

Returns:

None

5.2.2.3 void bspLedSet (uint8_t ui8Leds)

This function sets LED(s) specified by **ui8Leds**. The function assumes that LED pins have been initialized by, for example, [bspLedInit\(\)](#).

Parameters:

ui8Leds is an ORed bitmask of LEDs (for example **BSP_LED_1**).

Returns:

None

5.2.2.4 void bspLedToggle (uint8_t ui8Leds)

This function toggles LED(s) specified by **ui8Leds**. The function assumes that LED pins have been initialized by, for example, [bspLedInit\(\)](#).

Parameters:

ui8Leds ORed bitmask of LEDs (for example **BSP_LED_1**).

Returns:

None

5.3 Programming Example

The following example shows how to use the BSP LED API to initialize the LEDs and to turn on an LED. For more LED code examples, see [bsp/srf06eb_cc2538/examples/leds](#).

```
//  
// Initialize the SmartRF06EB LEDs as off.  
//  
bspLedInit();
```

```
//  
// Turn on LED 1 and 2.  
//  
bspLedSet (BSP_LED_1 | BSP_LED_2);
```


6 Keys

Introduction	25
API Functions	25
Programming Example	28

6.1 Introduction

The SmartRF06EB has 5 keys for interfacing the CC2538. The keys can be accessed using defines **BSP_KEY_1** through **BSP_KEY_5**. The keys can also be accessed using more user-friendly defines such as **BSP_KEY_LEFT** and **BSP_KEY_SELECT**. Define **BSP_KEY_ALL** is an ORed bitmask of all keys on the SmartRF06EB accessible from the CC2538.

The driver files are in `bsp/srf06eb_cc2538/drivers`.

- `source/bsp_key.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/bsp_key.h` contains the API definitions for use by applications.

6.2 API Functions

Functions

- `uint8_t bspKeyGetDir` (void)
- `void bspKeyInit` (uint8_t ui8Mode)
- `void bspKeyIntClear` (uint8_t ui8Keys)
- `void bspKeyIntDisable` (uint8_t ui8Keys)
- `void bspKeyIntEnable` (uint8_t ui8Keys)
- `void bspKeyIntRegister` (uint8_t ui8Keys, void (*pfnHandler)(void))
- `void bspKeyIntUnregister` (uint8_t ui8Keys)
- `uint8_t bspKeyPushed` (uint8_t ui8ReadMask)

6.2.1 Detailed Description

The SmartRF06EB BSP key driver is by default interrupt driven and uses the CC2538 watchdog timer for key debounce. Alternatively, the key driver may use polling and active state software debounce.

To configure the key driver as interrupt driven, pass **BSP_KEY_MODE_ISR** as argument to `bspKeyInit()`. To configure the BSP key driver to use polling, pass **BSP_KEY_MODE_POLL** as argument.

If the key driver is initialized using **BSP_KEY_MODE_POLL**, functions `bspKeyPushed()` and `bspKeyGetDir()` will poll the CC2538 I/O pins connected to the keys. In this case, functions with prefix **bspKeyInt** do nothing.

The key driver may be excluded from the SmartRF06EB BSP by defining **BSP_KEY_EXCLUDE**. For more information on how to configure the SmartRF06EB BSP for CC2538 precompiled library, see Section 2.4.

6.2.2 Function Documentation

6.2.2.1 bspKeyGetDir

This function reads the directional event. If multiple keys are registered as "pressed", this function will only return the directional event of the first key. Remaining key events will be ignored.

Prototype:

```
uint8_t  
bspKeyGetDir(void)
```

See also:

[bspKeyPushed\(\)](#)

Returns:

Returns **BSP_KEY_EVT_LEFT** if LEFT key has been pressed.

Returns **BSP_KEY_EVT_RIGHT** if RIGHT key has been pressed.

Returns **BSP_KEY_EVT_UP** if UP key has been pressed.

Returns **BSP_KEY_EVT_DOWN** if DOWN key has been pressed.

Returns **BSP_KEY_EVT_NONE** if no key has been pressed.

6.2.2.2 void bspKeyInit (uint8_t ui8Mode)

This function initializes key GPIO as input pullup and disables interrupts. If *ui8Mode* is **BSP_KEY_MODE_POLL**, key presses are handled using polling and active state debounce. Functions starting with **bspKeyInt** then do nothing.

If *ui8Mode* is **BSP_KEY_MODE_ISR**, key presses are handled by interrupts, and debounce is implemented using a timer.

Parameters:

ui8Mode is the operation mode; must be one of the following:

- **BSP_KEY_MODE_POLL** for polling-based handling
- **BSP_KEY_MODE_ISR** for interrupt-based handling

Returns:

None

6.2.2.3 void bspKeyIntClear (uint8_t ui8Keys)

This function clears interrupt flags on selected key GPIOs.

Note:

If `bspKeyInit()` was initialized with argument **BSP_KEY_MODE_POLL**, this function does nothing.

Parameters:

ui8Keys is an ORed bitmask of keys (for example BSP_KEY_1).

Returns:

None

6.2.2.4 void bspKeyIntDisable (uint8_t *ui8Keys*)

This function disables interrupts on specified key GPIOs.

Note:

If `bspKeyInit()` was initialized with argument **BSP_KEY_MODE_POLL**, this function does nothing.

Parameters:

ui8Keys is an ORed bitmask of keys (for example BSP_KEY_1).

Returns:

None

6.2.2.5 void bspKeyIntEnable (uint8_t *ui8Keys*)

This function enables interrupts on specified key GPIO pins.

Note:

If `bspKeyInit()` was initialized with argument **BSP_KEY_MODE_POLL**, this function does nothing.

Parameters:

ui8Keys is an ORed bitmask of keys (for example BSP_KEY_1).

Returns:

None

6.2.2.6 void bspKeyIntRegister (uint8_t *ui8Keys*, void(*)(void) *pfnHandler*)

This function registers a custom ISR to keys specified by *ui8Keys*.

Note:

If `bspKeyInit()` was initialized with argument **BSP_KEY_MODE_POLL**, this function does nothing.

Parameters:

ui8Keys is an ORed bitmask of keys (for example BSP_KEY_1).

pfnHandler is a void function pointer to ISR.

Returns:

None

6.2.2.7 void bspKeyIntUnregister (uint8_t *ui8Keys*)

This function clears the custom ISR from keys specified by *ui8Keys*.

Note:

If [bspKeyInit\(\)](#) was initialized with argument **BSP_KEY_MODE_POLL**, this function does nothing.

Parameters:

ui8Keys is an ORed bitmask of keys (for example **BSP_KEY_1**).

Returns:

None

6.2.2.8 uint8_t bspKeyPushed (uint8_t *ui8ReadMask*)

This function returns a bitmask of keys pushed.

Note:

If keys are handled using polling (**BSP_KEY_MODE_POLL**), the returned bitmask will never contain a combination of multiple key bitmasks, for example, (**BSP_KEY_LEFT** | **BSP_KEY_UP**). Furthermore, in this case argument *ui8ReadMask* is ignored.

Parameters:

ui8ReadMask is a bitmask of keys to read. Read keys are cleared and new key presses can be registered. Use **BSP_KEY_ALL** to read status of all keys.

Returns:

Returns bitmask of pushed keys

6.3 Programming Example

The following code example initializes the SmartRF06EB keys and toggles an LED if either the UP or DOWN key on SmartRF06EB is pressed. For more key code examples, see [bsp/srf06eb_cc2538/examples/keys](#).

```
#include <bsp.h>
#include <bsp_key.h>
#include <interrupt.h> // Access to driverlib IntMasterEnable()

//
// Initialize keys (interrupt driven with watchdog timer debounce)
//
bspKeyInit(BSP_KEY_ISR);
```

```
//  
// Enable interrupts on UP/DOWN key and global  
// interrupts (driverlib function)  
//  
bspKeyIntEnable (BSP_KEY_UP | BSP_KEY_DOWN);  
IntMasterEnable();  
  
while (1)  
{  
    if (bspKeyPushed (BSP_KEY_UP | BSP_KEY_DOWN) )  
    {  
        bspLedToggle (BSP_LED_1);  
    }  
}
```


7 Accelerometer

Introduction	31
API Functions	31
Programming Example	35

7.1 Introduction

The accelerometer on the SmartRF06EB is a BMA250 3-axis digital accelerometer. The SmartRF06EB BSP drivers provides a set of functions for configuring and accessing the accelerometer.

The driver files are in `bsp/srf06eb_cc2538/drivers`.

- `source/acc_srf06eb.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/acc_bma250.h` contains the API definitions for use by applications and accelerometer register definitions.

Accelerometer register definitions are prefixed using **ACC_** (for example **ACC_CHIPID**).

7.2 API Functions

Functions

- void [accDisable](#) (void)
- void [accInit](#) (void)
- void [accIntClear](#) (uint8_t ui8Pins)
- void [accIntDisable](#) (uint8_t ui8Pins)
- void [accIntEnable](#) (uint8_t ui8Pins)
- void [accIntRegister](#) (uint8_t ui8Pins, void (*pfnHandler)(void))
- void [accIntTypeSet](#) (uint8_t ui8Pins, uint32_t ui32IntType)
- void [accIntUnregister](#) (uint8_t ui8Pins)
- void [accReadData](#) (int16_t *pi16XVal, int16_t *pi16YVal, int16_t *pi16ZVal)
- void [accReadReg](#) (uint8_t ui8Addr, uint8_t *pui8Buf, uint8_t ui8Len)
- void [accWriteReg](#) (uint8_t ui8Addr, const uint8_t *pui8Buf, uint8_t ui8Len)

7.2.1 Detailed Description

The accelerometer API is broken into two groups of functions: those that access the accelerometer, and those that deal with interrupts from the accelerometer on the CC2538.

The following functions configure and access the accelerometer:

- [acclnit\(\)](#)
- [accDisable\(\)](#)
- [accReadReg\(\)](#)
- [accWriteReg\(\)](#)
- [accReadData\(\)](#)

The following functions configure accelerometer interrupts:

- [acclntRegister\(\)](#)
- [acclntUnregister\(\)](#)
- [acclntEnable\(\)](#)
- [acclntDisable\(\)](#)
- [acclntClear\(\)](#)
- [acclntTypeSet\(\)](#)

The accelerometer driver may be excluded from the SmartRF06EB BSP by defining **ACC_EXCLUDE**. For more information on how to configure the SmartRF06EB BSP for CC2538 precompiled library, see Section [2.4](#).

7.2.2 Function Documentation

7.2.2.1 accDisable

This function disables the accelerometer by turning off its power. This function assumes the accelerometer PWR pin is already configured as output by, for example, [acclnit\(\)](#).

Prototype:

```
void  
accDisable(void)
```

Returns:

None

7.2.2.2 void acclnit (void)

This function initializes the accelerometer. This must be run before you can use the accelerometer. The function assumes that the SPI interface has already been initialized using, for example, [bspSpiInit\(\)](#).

Returns:

None

7.2.2.3 void acclntClear (uint8_t *ui8Pins*)

This function clears interrupt flag on GPIO pins connected to accelerometer.

Parameters:

ui8Pins is a bitpacked bitmask of accelerometer interrupt pins; it can be an ORed combination of the following values:

- **BSP_ACC_INT1**
- **BSP_ACC_INT2**

Returns:

None

7.2.2.4 void acclntDisable (uint8_t *ui8Pins*)

This function disables interrupts on GPIO pins connected to accelerometer.

Parameters:

ui8Pins is a bitpacked bitmask of accelerometer interrupt pins; it can be an ORed combination of the following values:

- **BSP_ACC_INT1**
- **BSP_ACC_INT2**

Returns:

None

7.2.2.5 void acclntEnable (uint8_t *ui8Pins*)

This function enables interrupts on GPIO pins connected to accelerometer.

Parameters:

ui8Pins is a bitpacked bitmask of accelerometer interrupt pins; it can be an ORed combination of the following values:

- **BSP_ACC_INT1**
- **BSP_ACC_INT2**

Returns:

None

7.2.2.6 void acclntRegister (uint8_t *ui8Pins*, void(*)(void) *pfnHandler*)

This function registers a custom interrupt handler to the GPIO pins specified by *ui8Pins*.

Parameters:

ui8Pins is a bitpacked bitmask of accelerometer interrupt pins; it can be an ORed combination of the following values:

- **BSP_ACC_INT1**

■ **BSP_ACC_INT2**

pfnHandler is a pointer to the interrupt handler function.

Returns:

None

7.2.2.7 void accIntTypeSet (uint8_t *ui8Pins*, uint32_t *ui32IntType*)

This function sets the interrupt type for the GPIO pins connected to the accelerometer interrupt pin.

Parameters:

ui8Pins is a bitpacked bitmask of accelerometer interrupt pins; it can be an ORed combination of the following values:

- **BSP_ACC_INT1**
- **BSP_ACC_INT2**

ui32IntType is an enumerated data type that must be one of the following values:

- **GPIO_FALLING_EDGE**
- **GPIO_RISING_EDGE**
- **GPIO_BOTH_EDGES**
- **GPIO_LOW_LEVEL**
- **GPIO_HIGH_LEVEL**

7.2.2.8 void accIntUnregister (uint8_t *ui8Pins*)

This function unregisters the custom interrupt handler from the GPIO pins specified by *ui8Pins*.

Parameters:

ui8Pins is a bitpacked bitmask of accelerometer interrupt pins; it can be an ORed combination of the following values:

- **BSP_ACC_INT1**
- **BSP_ACC_INT2**

Returns:

None

7.2.2.9 void accReadData (int16_t * *pi16XVal*, int16_t * *pi16YVal*, int16_t * *pi16ZVal*)

This function reads present acceleration data. the function assumes the SPI in FIFO of the device to be empty.

Parameters:

- **pi16XVal* is a pointer to where the x-axis value is stored.
- **pi16YVal* is a pointer to where the y-axis value is stored.
- **pi16ZVal* is a pointer to where the z-axis value is stored.

Returns:

None

7.2.2.10 void accReadReg (uint8_t *ui8Addr*, uint8_t * *pui8Buf*, uint8_t *ui8Len*)

This function reads one or more accelerometer registers.

Parameters:

ui8Addr is the register start address.
pui8Buf is a pointer to the destination buffer.
ui8Len is the number of registers to read.

Returns:

None

7.2.2.11 void accWriteReg (uint8_t *ui8Addr*, const uint8_t * *pui8Buf*, uint8_t *ui8Len*)

This function writes one or more accelerometer registers. The function implements burst-like functionality. The BMA250 accelerometer does not support burst write (multiple writes with CSn low) thus CSn is pulled high between each address-data pair.

Parameters:

ui8Addr is the register start address.
pui8Buf is the pointer to source buffer.
ui8Len is the number of registers to write.

Returns:

None

7.3 Programming Example

The following code example initializes the accelerometer and periodically reads the X, Y and Z axis data. For more accelerometer code examples, see bsp/srf06eb_cc2538/examples/accelerometer.

```
#include <bsp.h>
#include <acc_bma250.h>

unsigned short x, y, z, i;

//
// Initialize the SPI interface
//
bspSpiInit (BSP_SPI_CLK_SPD);

//
// Initialize the accelerometer (interrupts disabled, 2g mode)
//
accInit();

while(1)
```

```
{  
    accReadData(&x, &y, &z);  
  
    // Simple wait  
    for(i=0; i<65000; i++);  
}
```

8 Ambient Light Sensor

Introduction	37
API Functions	37
Programming Example	38

8.1 Introduction

The ambient light sensor (ALS) on the SmartRF06EB is an automotive qualified Bosch SFH5711 with logarithmic current output. The SmartRF06EB BSP drivers provides a set of functions for configuring and reading the ALS output.

The driver files are in `bsp/srf06eb_cc2538/drivers`.

- `source/als_srf06eb.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/als_sfh5711.h` contains the API definitions for use by applications and accelerometer register definitions.

8.2 API Functions

Functions

- void `alsInit` (void)
- uint16_t `alsRead` (void)
- void `alsUninit` (void)

8.2.1 Detailed Description

The ALS API has three functions:

- Functions `alsInit()` and `alsUninit()` are used to configure the ALS.
- Function `alsRead()` uses the CC2538 ADC to read the voltage over an external resistor on the SmartRF06EB.

The ALS driver may be excluded from the SmartRF06EB BSP by defining **ALS_EXCLUDE**. For more information on how to configure the SmartRF06EB BSP for CC2538 precompiled library, see Section 2.4

8.2.2 Function Documentation

8.2.2.1 alsInit

This function initializes the ALS. The sensor is powered up and the onboard ADC is configured.

Prototype:

```
void  
alsInit(void)
```

Returns:

None

8.2.2.2 uint16_t alsRead (void)

This function triggers and returns ADC conversion from the ALS output. A 12-bit ADC conversion results in a value of [0, 4095].

Returns:

Returns the value read from the light sensor

8.2.2.3 void alsUninit (void)

This function uninitializes the ALS. This function assumes that the ALS power pin has already been configured as output using, for example, [alsInit\(\)](#).

Returns:

None

8.3 Programming Example

The following code example shows how to initialize and read data from the SFH5711 ALS. For more ALS code examples, see [bsp/srf06eb_cc2538/examples/lightsensor](#).

```
#include <als_sfh5711.h>  
  
//  
// Initialize the Ambient Light Sensor  
//  
alsInit();  
  
//  
// Sample value  
//  
unsigned short alsValue = alsRead();
```

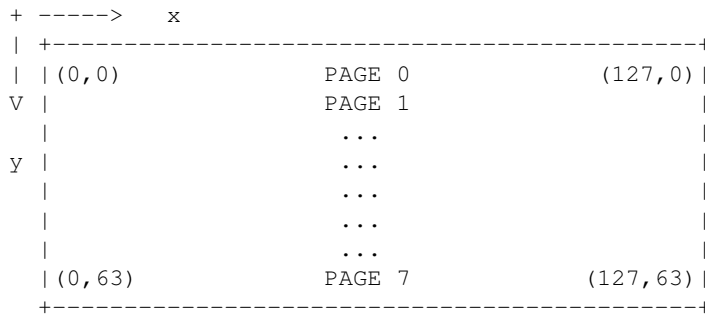
9 LCD

Introduction	39
API Functions	39
Programming Example	54

9.1 Introduction

The SmartRF06EB is fitted with a DOGM128-6 128 by 64 pixel dot matrix LCD display that is divided into 8 pages (**LCD_PAGE_0** through **LCD_PAGE_7**), each 8 pixels high.

An illustration of the (x,y) coordinate system used in this device driver follows:



Some of the features of the SmartRF06EB BSP LCD driver are:

- Print string, integers, and floating point numbers.
- Provide left, center and right alignment of strings, integers, and floating point numbers.
- Update entire LCD display, or parts of it.
- Draw vertical, horizontal, and tilted lines.
- Draw vertical and horizontal arrows.

The driver files are in `bsp/srf06eb_cc2538/drivers`.

- `source/lcd_srf06eb.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/lcd_dogm128_6.c` contains generic function implementations.
- `source/lcd_dogm128_6.h` contains the API definitions for use by applications.
- `source/lcd_dogm128_6_alphabet.c` contains the font array for the DOGM128-6 LCD display.

9.2 API Functions

Functions

- void `lcdBufferClear` (char *pcBuffer)

- void [lcdBufferClearHLine](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8XTo, uint8_t ui8Y)
- void [lcdBufferClearLine](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8YFrom, uint8_t ui8XTo, uint8_t ui8YTo)
- void [lcdBufferClearPage](#) (char *pcBuffer, tLcdPage iPage)
- void [lcdBufferClearPart](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8XTo, tLcdPage iPageFrom, tLcdPage iPageTo)
- void [lcdBufferClearPx](#) (char *pcBuffer, uint8_t ui8X, uint8_t ui8Y)
- void [lcdBufferClearVLine](#) (char *pcBuffer, uint8_t ui8X, uint8_t ui8YFrom, uint8_t ui8YTo)
- void [lcdBufferCopy](#) (const char *pcFromBuffer, char *pcToBuffer)
- void [lcdBufferHArrow](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8XTo, uint8_t ui8Y)
- void [lcdBufferInvert](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8YFrom, uint8_t ui8XTo, uint8_t ui8YTo)
- void [lcdBufferInvertPage](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8XTo, tLcdPage iPage)
- void [lcdBufferPrintFloat](#) (char *pcBuffer, float fNumber, uint8_t ui8Decimals, uint8_t ui8X, tLcdPage iPage)
- void [lcdBufferPrintFloatAligned](#) (char *pcBuffer, float fNumber, uint8_t ui8Decimals, tLcdAlign iAlignment, tLcdPage iPage)
- void [lcdBufferPrintInt](#) (char *pcBuffer, int32_t i32Number, uint8_t ui8X, tLcdPage iPage)
- void [lcdBufferPrintIntAligned](#) (char *pcBuffer, int32_t i32Number, tLcdAlign iAlignment, tLcdPage iPage)
- void [lcdBufferPrintString](#) (char *pcBuffer, const char *pcStr, uint8_t ui8X, tLcdPage iPage)
- void [lcdBufferPrintStringAligned](#) (char *pcBuffer, const char *pcStr, tLcdAlign iAlignment, tLcdPage iPage)
- void [lcdBufferSetHLine](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8XTo, uint8_t ui8Y)
- void [lcdBufferSetLine](#) (char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8YFrom, uint8_t ui8XTo, uint8_t ui8YTo)
- void [lcdBufferSetPx](#) (char *pcBuffer, uint8_t ui8X, uint8_t ui8Y)
- void [lcdBufferSetVLine](#) (char *pcBuffer, uint8_t ui8X, uint8_t ui8YFrom, uint8_t ui8YTo)
- void [lcdBufferVArrow](#) (char *pcBuffer, uint8_t ui8X, uint8_t ui8YFrom, uint8_t ui8YTo)
- void [lcdClear](#) (void)
- uint8_t [lcdGetFloatLength](#) (float fNumber, uint8_t ui8Decimals)
- uint8_t [lcdGetIntLength](#) (int32_t i32Number)
- uint8_t [lcdGetStringLength](#) (const char *pcStr)
- void [lcdGotoXY](#) (uint8_t ui8X, uint8_t ui8Y)
- void [lcdInit](#) (void)
- void [lcdSendBuffer](#) (const char *pcBuffer)
- void [lcdSendBufferAnimated](#) (const char *pcToBuffer, const char *pcFromBuffer, tLcdMotion iMotion)
- void [lcdSendBufferPart](#) (const char *pcBuffer, uint8_t ui8XFrom, uint8_t ui8XTo, tLcdPage iPageFrom, tLcdPage iPageTo)
- void [lcdSendCommand](#) (const char *pcCmd, uint8_t ui8Len)
- void [lcdSendData](#) (const char *pcData, uint16_t ui1Len)
- void [lcdSetContrast](#) (uint8_t ui8Contrast)
- void [lcdSpilnit](#) (void)

9.2.1 Detailed Description

The SmartRF06EB BSP LCD API is broken into two main groups:

- Functions that manipulate a local buffer on the CC2538.
- Functions that accesses the LCD display.

Functions that manipulate a local LCD buffer are prefixed with **lcdBuffer**, for example [lcdBufferPrintString\(\)](#). Functions that manipulate the LCD display are prefixed with **lcdSend**, for example [lcdSendBuffer\(\)](#).

Function [lcdInit\(\)](#) configures the LCD display and must be executed before calling any other functions accessing the LCD display. The CC2538 SPI interface must be initialized before calling [lcdInit\(\)](#), using, for example, [lcdSpiInit\(\)](#). Function [lcdClear\(\)](#) clears the content of the LCD display while [lcdSetContrast\(\)](#) sets the display contrast.

Functions for sending raw data and commands to the LCD display are [lcdSendData\(\)](#) and [lcdSendCommand\(\)](#). To update parts, or the entire LCD display, functions [lcdSendBuffer\(\)](#), [lcdSendBufferPart\(\)](#), and [lcdSendBufferAnimated\(\)](#) are provided.

Functions for handling text strings are [lcdBufferPrintString\(\)](#), [lcdBufferPrintStringAligned\(\)](#), and utility function [lcdGetStringLength\(\)](#).

Functions for handling integers are [lcdBufferPrintInt\(\)](#), [lcdBufferPrintIntAligned\(\)](#), and utility function [lcdGetIntLength\(\)](#).

Functions for handling float numbers are [lcdBufferPrintFloat\(\)](#), [lcdBufferPrintFloatAligned\(\)](#), and utility function [lcdGetFloatLength\(\)](#).

Functions for drawing lines, arrows, and single pixels are [lcdBufferSetLine\(\)](#), [lcdBufferClearLine\(\)](#), [lcdBufferSetVLine\(\)](#), [lcdBufferClearVLine\(\)](#), [lcdBufferSetHLine\(\)](#), [lcdBufferClearHLine\(\)](#), [lcdBufferVArrow\(\)](#), [lcdBufferHArrow\(\)](#), [lcdBufferSetPx\(\)](#), and [lcdBufferClearPx\(\)](#).

Other functions for manipulating the LCD buffer are [lcdBufferInvert\(\)](#) and [lcdBufferInvertPage\(\)](#).

By default, the LCD driver allocates 1024 bytes for a local LCD buffer. Passing 0 as the buffer argument manipulates or sends this buffer. To reduce RAM use, build flag **LCD_NO_DEFAULT_BUFFER** may override the allocation of the buffer.

Warning:

If **LCD_NO_DEFAULT_BUFFER** is defined, passing 0 as the buffer argument results in undefined behavior.

The LCD driver may be excluded from the SmartRF06EB BSP by defining **LCD_EXCLUDE**. For more information on how to configure the SmartRF06EB BSP for CC2538 precompiled library, see Section 2.4.

9.2.2 Function Documentation

9.2.2.1 lcdBufferClear

This function empties the LCD buffer specified by argument *pcBuffer* by filling it with zeros.

Prototype:

```
void
lcdBufferClear(char *pcBuffer)
```

Parameters:

pcBuffer is a pointer to the target buffer.

Returns:

None

9.2.2.2 void lcdBufferClearHLine (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8XTo*, uint8_t *ui8Y*)

this function Clears a horizontal line from (*ui8XFrom*, *ui8Y*) to (*ui8XTo*, *ui8Y*) from buffer *pcBuffer*.

Parameters:

pcBuffer is a pointer to the target buffer.

ui8XFrom is the start column [0–127].

ui8XTo is the end column [0–127].

ui8Y is the row [0–63].

Returns:

None

9.2.2.3 void lcdBufferClearLine (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8YFrom*, uint8_t *ui8XTo*, uint8_t *ui8YTo*)

This function clears a line in buffer *pcBuffer* from (*ui8XFrom*, *ui8YFrom*) to (*ui8XTo*, *ui8YTo*). The function uses Bresenham's line algorithm.

Parameters:

pcBuffer is a pointer to the target buffer.

ui8XFrom is the start column [0–127].

ui8XTo is the end column [0–127].

ui8YFrom is the start row [0–63].

ui8YTo is the end row [0–63].

Returns:

None

9.2.2.4 void lcdBufferClearPage (char * *pcBuffer*, tLcdPage *iPage*)

This function clears the page specified by *iPage* in LCD buffer specified by *pcBuffer*.

Parameters:

pcBuffer is a pointer to the target buffer.

iPage is the page to clear. Must be one of the following enumerated values:

- eLcdPage0
- eLcdPage1
- eLcdPage2

- eLcdPage3
- eLcdPage4
- eLcdPage5
- eLcdPage6
- eLcdPage7

Returns:

None

9.2.2.5 void lcdBufferClearPart (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8XTo*, tLcdPage *iPageFrom*, tLcdPage *iPageTo*)

This function clears the pixels in a given piece of a page. Resolution is given in coulmnns [0–127] and pages [0–7]. The function assumes *ui8XFrom* <= *ui8XTo* and *iPageFrom* <= *iPageTo*.

Parameters:

pcBuffer is a pointer to the target buffer.

ui8XFrom is the lowest x-position (column) to be cleared [0–127].

ui8XTo is the highest x-position to be cleared [*ui8XFrom*–127].

iPageFrom is the first page cleared. Must be one of the following enumerated values:

- eLcdPage0
- eLcdPage1
- eLcdPage2
- eLcdPage3
- eLcdPage4
- eLcdPage5
- eLcdPage6
- eLcdPage7

iPageTo is the last page cleared [*iPageFrom*–eLcdPage7].

Returns:

None

9.2.2.6 void lcdBufferClearPx (char * *pcBuffer*, uint8_t *ui8X*, uint8_t *ui8Y*)

This function clears the pixel at (*ui8X*, *ui8Y*).

Parameters:

pcBuffer is a pointer to the target buffer.

ui8X is the pixel x-position (column) [0–127].

ui8Y is the pixel y-position (row) [0–63].

Returns:

None

9.2.2.7 void lcdBufferClearVLine (char * *pcBuffer*, uint8_t *ui8X*, uint8_t *ui8YFrom*, uint8_t *ui8YTo*)

This function clears a vertical line from (*ui8X*, *ui8YFrom*) to (*ui8X*, *ui8YTo*) from buffer specified by argument *pcBuffer*.

Parameters:

pcBuffer is a pointer to the target buffer.
ui8X is the x-position (column) of the line [0–127].
ui8YFrom is the start row [0–63].
ui8YTo is the end row [0–63].

Returns:

None

9.2.2.8 void lcdBufferCopy (const char * *pcFromBuffer*, char * *pcToBuffer*)

This function copies the content of *pcFromBuffer* to *pcToBuffer*. If either of the two arguments are 0, the default buffer is used for this argument.

Parameters:

pcToBuffer is a pointer to the destination buffer.
pcFromBuffer is a pointer to the target buffer.

Returns:

None

9.2.2.9 void lcdBufferHArrow (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8XTo*, uint8_t *ui8Y*)

This function draws a horizontal arrow from (*ui8XFrom*, *ui8Y*) to (*ui8XTo*, *ui8Y*) to buffer specified by *pcBuffer*. The function assumes *ui8Y* to be in the range [2–61] in order for arrowhead to fit on the LCD.

Parameters:

pcBuffer is a pointer to target buffer.
ui8XFrom is the start column [0–127].
ui8XTo is the end column [0–127].
ui8Y is the the y-position (row) of the arrow [2–61].

Returns:

None

9.2.2.10 void lcdBufferInvert (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8YFrom*, uint8_t *ui8XTo*, uint8_t *ui8YTo*)

This function inverts the pixels (bits) in a given region of the buffer specified by *pcBuffer*.

Parameters:

pcBuffer is a pointer to the target buffer.
ui8XFrom is the first x-position (column) to invert [0–127].
ui8YFrom is the first y-position (row) to invert [0–63].
ui8XTo is the last x-position (column) to invert [0–127].
ui8YTo is the last y-position (row) to invert [0–63].

Returns:

None

9.2.2.11 void lcdBufferInvertPage (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8XTo*, tLcdPage *iPage*)

This function inverts a range of columns in the display buffer on a specified page (for example, **eLcdPage0**). This function assumes *ui8XFrom* ≤ *ui8XTo*.

Parameters:

pcBuffer is a pointer to the target buffer.
ui8XFrom is the first x-position (column) to invert [0–127].
ui8XTo is the last x-position to invert [*ui8XFrom*–127].
iPage is the page on which to invert. Must be one of the following enumerated values:

- **eLcdPage0**
- **eLcdPage1**
- **eLcdPage2**
- **eLcdPage3**
- **eLcdPage4**
- **eLcdPage5**
- **eLcdPage6**
- **eLcdPage7**

Returns:

None

9.2.2.12 void lcdBufferPrintFloat (char * *pcBuffer*, float *fNumber*, uint8_t *ui8Decimals*, uint8_t *ui8X*, tLcdPage *iPage*)

This function writes a number of data type float on the display at a specified column and page. Use this function instead of performing a float to c-string conversion and then using [lcdBufferPrintString\(\)](#).

Parameters:

pcBuffer is a pointer to the target buffer.
fNumber is the number to print.
ui8Decimals is the number of decimals to print, MAX = 10.
ui8X is the x-position (column) to begin printing [0–127].
iPage is the page on which to print. Must be one of the following enumerated values:

- **eLcdPage0**

- eLcdPage1
- eLcdPage2
- eLcdPage3
- eLcdPage4
- eLcdPage5
- eLcdPage6
- eLcdPage7

Returns:

None

9.2.2.13 void lcdBufferPrintFloatAligned (char * *pcBuffer*, float *fNumber*, uint8_t *ui8Decimals*, tLcdAlign *iAlignment*, tLcdPage *iPage*)

This function writes a float number to buffer *pcBuffer* as specified by the *iAlignment* argument.

Parameters:

pcBuffer is a pointer to the target buffer.

fNumber is the number to be printed.

ui8Decimals is the number of decimals to be printed, MAX = 10.

iAlignment is the text alignment. Can be one of the following enumerated values:

- eLcdAlignLeft
- eLcdAlignCenter
- eLcdAlignRight

iPage is the page on which to print. Must be one of the following enumerated values:

- eLcdPage0
- eLcdPage1
- eLcdPage2
- eLcdPage3
- eLcdPage4
- eLcdPage5
- eLcdPage6
- eLcdPage7

Returns:

None

9.2.2.14 void lcdBufferPrintInt (char * *pcBuffer*, int32_t *i32Number*, uint8_t *ui8X*, tLcdPage *iPage*)

This function writes an integer to the buffer specified by *pcBuffer*.

Parameters:

pcBuffer is a pointer to the target buffer.

i32Number is the number to print.

ui8X is the x-position (column) to begin printing [0–127].

iPage is the page on which to print. Must be one of the following enumerated values:

- **eLcdPage0**
- **eLcdPage1**
- **eLcdPage2**
- **eLcdPage3**
- **eLcdPage4**
- **eLcdPage5**
- **eLcdPage6**
- **eLcdPage7**

Returns:

None

9.2.2.15 void **LcdBufferPrintIntAligned** (char * *pcBuffer*, int32_t *i32Number*, tLcdAlign *iAlignment*, tLcdPage *iPage*)

This function writes an integer to buffer *pcBuffer* as specified by the *ui8Alignment* argument.

Parameters:

pcBuffer is a pointer to the target buffer.

i32Number is the number to be printed.

iAlignment is the text alignment. Must be one of the following enumerated values:

- **eLcdAlignLeft**
- **eLcdAlignCenter**
- **eLcdAlignRight**

iPage is the page on which to print. Must be one of the following enumerated values:

- **eLcdPage0**
- **eLcdPage1**
- **eLcdPage2**
- **eLcdPage3**
- **eLcdPage4**
- **eLcdPage5**
- **eLcdPage6**
- **eLcdPage7**

Returns:

None

9.2.2.16 void **LcdBufferPrintString** (char * *pcBuffer*, const char * *pcStr*, uint8_t *ui8X*, tLcdPage *iPage*)

This function writes a string to the buffer specified by *pcBuffer*.

Parameters:

pcBuffer is a pointer to the output buffer.

pcStr is a pointer to the string to print.

ui8X is the x-position (column) to begin printing [0–127].

iPage is the page on which to print. Must be one of the following enumerated values:

- **eLcdPage0**
- **eLcdPage1**
- **eLcdPage2**
- **eLcdPage3**
- **eLcdPage4**
- **eLcdPage5**
- **eLcdPage6**
- **eLcdPage7**

Returns:

None

9.2.2.17 void lcdBufferPrintStringAligned (char * *pcBuffer*, const char * *pcStr*, tLcdAlign *iAlignment*, tLcdPage *iPage*)

This function writes a string to buffer *pcBuffer* as specified by the *iAlignment* argument.

Parameters:

pcBuffer is a pointer to the target buffer.

pcStr is a pointer to the string to print.

iAlignment is the text alignment. Must be one of the following enumerated values:

- **eLcdAlignLeft**
- **eLcdAlignCenter**
- **LCD_ALIGN_RIGHT**

iPage is the page on which to print. Must be one of the following enumerated values:

- **eLcdPage0**
- **eLcdPage1**
- **eLcdPage2**
- **eLcdPage3**
- **eLcdPage4**
- **eLcdPage5**
- **eLcdPage6**
- **eLcdPage7**

Returns:

None

9.2.2.18 void lcdBufferSetHLine (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8XTo*, uint8_t *ui8Y*)

This function draws a horizontal line from (*ui8XFrom*, *ui8Y*) to (*ui8XTo*, *ui8Y*) into buffer *pcBuffer*.

Parameters:

pcBuffer is a pointer to the target buffer.

ui8XFrom is the start column [0–127].
ui8XTo is the end column [0–127].
ui8Y is the row [0–63].

Returns:

None

9.2.2.19 void lcdBufferSetLine (char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8YFrom*,
uint8_t *ui8XTo*, uint8_t *ui8YTo*)

This function draws a line in buffer *pcBuffer* from (*ui8XFrom*, *ui8YFrom*) to (*ui8XTo*, *ui8YTo*). The function uses Bresenham's line algorithm.

Parameters:

pcBuffer is a pointer to the target buffer.
ui8XFrom is the start column [0–127].
ui8XTo is the end column [0–127].
ui8YFrom is the start row [0–63].
ui8YTo is the end row [0–63].

Returns:

None

9.2.2.20 void lcdBufferSetPx (char * *pcBuffer*, uint8_t *ui8X*, uint8_t *ui8Y*)

This function sets a pixel on (*ui8X*, *ui8Y*).

Parameters:

pcBuffer is a pointer to the target buffer.
ui8X is the pixel x-position (column) [0–127].
ui8Y is the pixel y-position (row) [0–63].

Returns:

None

9.2.2.21 void lcdBufferSetVLine (char * *pcBuffer*, uint8_t *ui8X*, uint8_t *ui8YFrom*, uint8_t
ui8YTo)

This function draws a vertical line from (*ui8X*, *ui8YFrom*) to (*ui8X*, *ui8YTo*) into buffer *pcBuffer*.

Parameters:

pcBuffer is a pointer to the target buffer.
ui8X is the x-position (column) of the line [0–127].
ui8YFrom is the start row [0–63].
ui8YTo is the end row [0–63].

Returns:

None

9.2.2.22 void lcdBufferVArrow (char * *pcBuffer*, uint8_t *ui8X*, uint8_t *ui8YFrom*, uint8_t *ui8YTo*)

This function draws a vertical arrow from (*ui8X*, *ui8YFrom*) to (*ui8X*, *ui8YTo*) to the buffer specified by *pcBuffer*. The function assumes that *ui8X* is in the range [2–125] for the arrowhead to fit on the LCD.

Parameters:

pcBuffer is a pointer to the target buffer.
ui8X is the the x-position (column) of the arrow [2–125].
ui8YFrom is the start row [0–63].
ui8YTo is the end row [0–63].

Returns:

None

9.2.2.23 void lcdClear (void)

This function clears the LCD display. This function acts directly on the display and does not modify internal buffers.

Returns:

None

9.2.2.24 uint8_t lcdGetFloatLength (float *fNumber*, uint8_t *ui8Decimals*)

This function returns the character length a float will need on the LCD display. This function is used by [lcdBufferPrintFloat\(\)](#) and [lcdBufferPrintFloatAligned\(\)](#). *ui8Decimals* must be provided to limit the number of decimals.

Parameters:

fNumber is the number whose character length is determined.
ui8Decimals is the desired number of decimals to use (maximum 10).

Returns:

Returns the character length of *fNumber*.

9.2.2.25 uint8_t lcdGetIntLength (int32_t *i32Number*)

This function returns the character length an integer will use on the LCD display. For example, *i32Number* = 215 returns 3 and *i32Number* = –215 returns 4 (add one for the minus character). Multiply result of [lcdGetIntLength\(\)](#) by **LCD_CHAR_WIDTH** to determine the number of pixels needed by *i32Number*.

Parameters:

i32Number is the number whose character length is determined.

Returns:

Returns the character length of *i32Number*.

9.2.2.26 uint8_t lcdGetStringLength (const char * *pcStr*)

Returns the length a c-string in number of characters by looking for the end-of-string character '\0'. Multiply by **LCD_CHAR_WIDTH** to get length in pixels.

Parameters:

pcStr is the null-terminated string whose character length is determined.

Returns:

Returns length of *pcStr*

9.2.2.27 void lcdGotoXY (uint8_t *ui8X*, uint8_t *ui8Y*)

This function sets the internal data cursor of the LCD to the location specified by *ui8X* and *ui8Y*. When data is sent to the display, data will start printing at internal cursor location.

Parameters:

ui8X is the column [0–127].

ui8Y is the page [0–7].

Returns:

None

9.2.2.28 void lcdInit (void)

This function initializes the LCD. This function assumes that the SPI interface has been initialized using, for example, [bspSpiInit\(\)](#). [lcdInit\(\)](#) must be run before you can use the LCD.

Returns:

None

9.2.2.29 void lcdSendBuffer (const char * *pcBuffer*)

This function sends the specified buffer to the display. The buffer size is assumed to be 1024 bytes. Passing *pcBuffer* as 0 will send the default buffer. If **LCD_NO_DEFAULT_BUFFER** is defined, passing *pcBuffer* as 0 will result in undefined behavior.

Parameters:

pcBuffer is a pointer to the source buffer.

Returns:

None

9.2.2.30 void lcdSendBufferAnimated (const char * *pcToBuffer*, const char * *pcFromBuffer*, tLcdMotion *iMotion*)

This function updates the LCD display by creating an animated transition between two display buffers. Two animations, **eLcdSlideLeft** and **eLcdSlideRight**, slide the new screen left or right, respectively.

Function **lcdSendBuffer()** updates the display to show the new buffer instantaneously. **lcdSendBufferAnimated()** on the other side, makes a smooth transition into showing the new buffer.

pcToBuffer should point to the buffer the LCD display transitions in to. *pcFromBuffer* should point to the buffer that what was sent to the LCD display last time **lcdSendBuffer()** or **lcdSendBufferAnimated()** was called. By taking both the present and the upcoming display buffers as parameters, **lcdSendBufferAnimated()** does not take up any memory unless used.

Example:

1. Send a buffer to the display using for example **lcdSendBuffer()**.
2. Manipulate a second buffer using **lcdBuffer** functions.
3. Run **lcdSendBufferAnimated()** to update display with a smooth transition from the initial to the second buffer.

Parameters:

pcToBuffer is a pointer to the buffer with the new display content.

pcFromBuffer is a pointer to the buffer with the existing display content.

iMotion indicates which animation to use for transition. Must be one of the following enumerated values:

- **eLcdSlideLeft**
- **eLcdSlideRight**

Returns:

None

9.2.2.31 void lcdSendBufferPart (const char * *pcBuffer*, uint8_t *ui8XFrom*, uint8_t *ui8XTo*, tLcdPage *iPageFrom*, tLcdPage *iPageTo*)

This function sends the specified part of *pcBuffer* to the corresponding part on the LCD. This function assumes *ui8XFrom* ≤ *ui8XTo* and *iPageFrom* ≤ *iPageTo*. The resolution is given in columns [0–127] and pages [0–7].

Parameters:

pcBuffer is a pointer to the buffer to send. The default buffer is sent if *pcBuffer* is 0.

ui8XFrom is the lowest x-position (column) to write [0–127].

ui8XTo is the highest x-position to write [*ui8XFrom*–127].

iPageFrom is the first page to write. Must be one of the following enumerated values:

- **eLcdPage0**
- **eLcdPage1**
- **eLcdPage2**
- **eLcdPage3**
- **eLcdPage4**

- eLcdPage5
- eLcdPage6
- eLcdPage7

iPageTo is the last page to write [*iPageFrom*–eLcdPage7].

Returns:

None

9.2.2.32 void lcdSendCommand (const char * *pcCmd*, uint8_t *ui8Len*)

This function sends *ui8Len* bytes of commands to the LCD controller.

Parameters:

pcCmd is a pointer to the array of commands.

ui8Len is the number of bytes to send.

Returns:

None

9.2.2.33 void lcdSendData (const char * *pcData*, uint16_t *ui1Len*)

This function sends *ui8Len* bytes of data to be displayed on the LCD.

Parameters:

pcData is a pointer to the array of data.

ui1Len is the number of bytes to send.

Returns:

None

9.2.2.34 void lcdSetContrast (uint8_t *ui8Contrast*)

This function sets the LCD contrast.

Parameters:

ui8Contrast is the contrast value [0–63].

Returns:

None

9.2.2.35 void lcdSpilnit (void)

This function initializes the LCD SPI interface to the maximum allowed speed.

Returns:

None

9.3 Programming Example

The following example shows how to use the LCD API to initialize the LCD, manipulate a local buffer and transmit it to the LCD display. For more LCD code examples, see bsp/srf06eb_cc2538/examples/lcd.

```
#include "bsp.h"
#include "lcd_dogm128_6.h"

//
// Initialize the SPI interface and then LCD display.
//
bspSpiInit();
lcdInit();

//
// Write a string string to page 2 of the default buffer
// (first arg. is 0), starting at x-position (column) 1.
//
lcdBufferPrintString(0, "Hello world!", 1, LCD_PAGE_2);

//
// Send the default buffer to the LCD display.
//
lcdSendBuffer(0);
```

10 Micro SD Card Reader

Introduction	55
API Functions	55
Programming Example	58

10.1 Introduction

The SmartRF06EB is fitted with a micro SD card reader. The SmartRF06EB BSP SD card driver provides a set of functions for initializing and accessing a connected micro SD/MMC card over SPI.

The driver files are in `bsp/srf06eb_cc2538/drivers/`.

- `source/sdcard_srf06eb.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/sdcard.h` contains the API definitions for use by applications and Micro SD Card register definitions.

SD card commands and definitions are prefixed using **SDCARD_** (for example **SDCARD_CMD0**).

10.2 API Functions

Functions

- `uint8_t sdCardBlockRead` (`uint32_t ui32Block`, `uint8_t *pui8Buffer`)
- `uint8_t sdCardBlockWrite` (`uint32_t ui32Block`, `const uint8_t *pui8Buffer`)
- `uint32_t sdCardGetBlockSize` (`void`)
- `uint8_t sdCardGetCid` (`tSdCardCid *psCid`)
- `uint8_t sdCardGetCsd` (`uint8_t *pui8Csd`)
- `uint32_t sdCardGetSize` (`void`)
- `uint8_t sdCardGetStatusReg` (`uint8_t *pui8Buffer`)
- `uint8_t sdCardInit` (`void`)
- `uint8_t sdCardStatus` (`void`)

10.2.1 Detailed Description

The SmartRF06EB micro SD card API provides basic functionality for initializing and accessing a mounted SD card.

Function `sdCardInit()` initializes the connected card and must be called before calling any other SD card API functions. Function `sdCardStatus()` returns the current connection state of the SD card.

Functions `sdCardBlockRead()` and `sdCardBlockWrite()` provide functionality for reading and writing blocks of data to the micro SD card.

Functions `sdCardGetStatusReg()`, `sdCardGetCid()`, `sdCardGetCsd()`, and `sdCardGetSize()` are provided to read various registers from the SD card.

Function `sdCardGetBlockSize()` is a utility function that returns the default SD card block size.

The SD card driver may be excluded from the SmartRF06EB BSP by defining **SD-CARD_EXCLUDE**. For more information on how to configure the SmartRF06EB BSP for CC2538 precompiled library, see Section 2.4.

10.2.2 Function Documentation

10.2.2.1 sdCardBlockRead

This function reads block *ui32Block* from the SD card. The function converts argument *ui32Block* to byte address if needed.

Prototype:

```
uint8_t  
sdCardBlockRead(uint32_t ui32Block,  
                uint8_t *pui8Buffer)
```

Parameters:

ui32Block is the logical block to read (LBA, Logical Block Addressing).

pui8Buffer is a pointer to the destination array for read data.

Returns:

Returns SDCARD_SUCCESS on success

10.2.2.2 uint8_t sdCardBlockWrite (uint32_t *ui32Block*, const uint8_t * *pui8Buffer*)

This function writes **SDCARD_BLOCKLENGTH** bytes to block *ui32Block* on the SD card. The function converts argument *ui32Block* to byte address if needed.

Parameters:

ui32Block is the logical block to write (LBA).

pui8Buffer is a pointer to the source array with data.

Returns:

Returns SDCARD_SUCCESS on success

10.2.2.3 uint32_t sdCardGetBlockSize (void)

This function returns the SD card block size. This function does not access the SD card.

Returns:

Returns block size in bytes

10.2.2.4 uint8_t sdCardGetCid (tSdCardCid * *psCid*)

This function reads out the 16-byte long card identification data (CID) register.

Parameters:

psCid is a pointer to the *tSdCardCid* structure.

Returns:

Returns SDCARD_SUCCESS on success

10.2.2.5 uint8_t sdCardGetCsd (uint8_t * *pui8Csd*)

This function reads out the card specific data (CSD) register . The size of the CSD register is 16 bytes. The data are stored in the buffer specified by *pui8Csd* with MSB first; for example, CSD[127:126] is at *pui8Csd*[7:6].

Parameters:

pui8Csd is a pointer to the destination array.

Returns:

Returns SDCARD_SUCCESS on success

10.2.2.6 uint32_t sdCardGetSize (void)

This function returns the size of card in KiB (1 KiB = 2¹⁰ bytes).

Returns:

Returns size of card in bytes, 0 if failed

10.2.2.7 uint8_t sdCardGetStatusReg (uint8_t * *pui8Buffer*)

This function reads out the 2-byte SD card status register.

Parameters:

pui8Buffer is a pointer to the destination array.

Returns:

Returns SDCARD_SUCCESS on success

10.2.2.8 uint8_t sdCardInit (void)

This function initialies an SD/MMC card. This function must be run before you can use the SD card. This function assumes that the SPI interface has been initialized using, for example, [bspSpiInit\(\)](#).

Returns:

Returns SDCARD_SUCCESS on success

10.2.2.9 uint8_t sdCardStatus (void)

This function checks the card connection status.

Returns:

Returns **SDCARD_STATUS_READY** if card is present and initialized.

Returns **SDCARD_STATUS_NOINIT** if card is present, but not initialized.

Returns **SDCARD_STATUS_NOCARD** if no card is detected.

10.3 Programming Example

The below code example shows how to initialize a micro SD card and read a data block from the SD card. See bsp/srf06eb_cc2538/examples/sdcard for more micro SD card reader code examples.

```
#include <bsp.h>
#include <sdcard.h>

unsigned char readBuffer[SDCARD_BLOCKLENGTH];

//
// Initialize SPI interface, then try to
// initialize micro SD card
//
bspSpiInit(BSP_SPI_CLK_SPD);
if(sdCardInit() != SDCARD_SUCCESS)
{
    bspAssert();    // Initialization failed
}

//
// Read block 0 from SD card into readBuffer.
// An SD card block is 512 bytes.
//
if(sdCardBlockRead(0, readBuffer) != SDCARD_SUCCESS)
{
    bspAssert(); // Failed to read from SD card
}
```

11 UART Backchannel

Introduction	59
API Functions	59
Programming Example	63

11.1 Introduction

The SmartRF06EB BSP for CC2538 implements a simple UART driver that is targeted for use with the UART to USB backchannel that is available on the SmartRF06EB. The BSP UART driver supports the 8-N-1 UART format: 8 data bits, no parity bits, and 1 stop bit. Hardware flow control is not supported.

The BSP UART driver uses internal buffers for received and transmitted data. The size of the internal buffers is run time customizable.

The driver files are in `bsp/srf06eb_cc2538/drivers`.

- `source/bsp_uart.c` contains the function implementations for CC2538 on SmartRF06EB.
- `source/bsp_uart.h` contains the API definitions for use by applications.

11.2 API Functions

Functions

- `uint32_t bspUartBaudRateGet` (void)
- `uint8_t bspUartBufInit` (`uint8_t *pui8TxBuf`, `uint16_t ui16TxByteAlloc`, `uint8_t *pui8RxBuf`, `uint16_t ui16RxByteAlloc`)
- `uint8_t bspUartClose` (void)
- `uint16_t bspUartDataGet` (`uint8_t *pui8Data`, `uint16_t ui16Length`)
- `uint16_t bspUartDataPut` (`uint8_t *pui8Data`, `uint16_t ui16Length`)
- `void bspUartFlushRx` (void)
- `void bspUartFlushTx` (void)
- `void bspUartIsrHandler` (void)
- `uint8_t bspUartOpen` (`uint32_t ui32BaudRate`)
- `uint16_t bspUartRxCharsAvail` (void)
- `uint16_t bspUartTxSpaceAvail` (void)

11.2.1 Detailed Description

The SmartRF06EB BSP UART driver is broken into three groups:

- Functions that configure the UART connection and buffers
- Functions for transmitting and receiving data

■ Utility functions

The `bspUartBufInit()` function configures the internal buffers and must be executed before calling any other BSP UART functions. The `bspUartOpen()` function configures the CC2538 UART module for operation, and the `bspUartClose()` function stops UART operation.

The `bspUartDataGet()` and `bspUartDataPut()` functions reads and transmits data over the UART connection.

The `bspUartBaudRateGet()`, `bspUartTxSpaceAvail()` and `bspUartRxCharsAvail()`, `bspUartFlushTx()`, and `bspUartFlushRx()` functions are utility functions.

The SmartRF06EB BSP UART driver is interrupt driven, but does not register a handler to the CC2538 UART interrupt vector; this must be done by the application. It is possible to configure the BSP UART driver to register an interrupt handler, using define **BSP_UART_ALLOCATE_ISR**.

By default, the `bspUartDataGet()` function will limit the number of bytes read to the smaller of the following two numbers:

- the requested number of bytes
- the number of bytes available in the internal receive buffer

The `bspUartDataPut()` function will similarly limit the number of bytes written to the internal transmit buffer. To configure the BSP UART driver to only read(write) data if the requested number of bytes is available in the internal receive(transmit) buffer, use define **BSP_UART_ALL_OR_NOTHING**.

The BSP UART driver may be excluded from the SmartRF06EB BSP by defining **BSP_UART_EXCLUDE**. For more information on how to configure the SmartRF06EB BSP for CC2538 precompiled library, see Section 2.4.

11.2.2 Function Documentation

11.2.2.1 bspUartBaudRateGet

This function returns the current BSP UART baud rate in baud. The function returns 0 if the BSP UART module is not configured.

Prototype:

```
uint32_t
bspUartBaudRateGet(void)
```

Returns:

Returns the current BSP UART baud rate in baud.

11.2.2.2 `uint8_t bspUartBufInit (uint8_t * pui8TxBuf, uint16_t ui16TxByteAlloc, uint8_t * pui8RxBuf, uint16_t ui16RxByteAlloc)`

This function initializes buffers used by BSP UART module.

Parameters:

pui8TxBuf is a pointer to the TX buffer.

ui16TxByteAlloc is the size of the TX buffer in bytes.

pui8RxBuf is a pointer to the RX buffer.

ui16RxByteAlloc is the size of the RX buffer in bytes.

Returns:

Returns BSP_UART_SUCCESS on success.

Returns BSP_UART_ERROR on configuration error.

11.2.2.3 uint8_t bspUartClose (void)

This function stops UART communication.

Returns:

Returns BSP_UART_SUCCESS on success.

11.2.2.4 uint16_t bspUartDataGet (uint8_t * *pui8Data*, uint16_t *ui16Length*)

This function reads up to *ui16Length* bytes from the BSP UART RX buffer into the buffer specified by *pui8Data*.

If **BSP_UART_ALL_OR_NOTHING** is defined, data is read only if *ui16Length* or more bytes are available in the BSP UART RX buffer.

Parameters:

pui8Data is a pointer to the destination buffer.

ui16Length is the number of bytes to transfer.

Returns:

Returns the number of bytes read from the BSP UART RX buffer.

11.2.2.5 uint16_t bspUartDataPut (uint8_t * *pui8Data*, uint16_t *ui16Length*)

This function puts up to *ui16Length* bytes into the BSP UART TX buffer and starts to transfer data over UART.

If **BSP_UART_ALL_OR_NOTHING** is defined, data is put into the TX buffer only if there is room for all *ui16Length* bytes.

Parameters:

pui8Data is a pointer to the source buffer.

ui16Length is the number of bytes to transfer.

Returns:

Returns the number of bytes actually copied to the TX buffer.

11.2.2.6 void bspUartFlushRx (void)

This function flushes the BSP UART RX buffer by resetting the buffer control structure.

Returns:
None

11.2.2.7 void bspUartFlushTx (void)

This function flushes the BSP UART TX buffer by resetting the buffer control structure.

Returns:
None

11.2.2.8 void bspUartIsrHandler (void)

This function handles BSP UART interrupts. This function clears all handled interrupt flags.

Returns:
None

11.2.2.9 uint8_t bspUartOpen (uint32_t *ui32BaudRate*)

This function initializes UART communication at the baud rate specified by *ui32BaudRate*. This function must be called after initializing the UART buffers using [bspUartBufInit\(\)](#). The UART format between the BSP MCU is 8-N-1 (that is, 8 data bits, no parity bit, and 1 stop bit). The implementation does not support flow control.

The UART module is configured to use IO clock as clock source.

Parameters:

ui32BaudRate is the UART baud rate. The baud rate must be one of the following enumerated values:

- **eBaudRate9600**
- **eBaudRate38400**
- **eBaudRate57600**
- **eBaudRate115200**
- **eBaudRate230400**
- **eBaudRate460800**

Returns:

Returns BSP_UART_SUCCESS on success.

Returns BSP_UART_UNCONFIGURED if buffers are not configured.

Returns BSP_UART_BAUDRATE_ERROR if the baud rate is not supported.

11.2.2.10 uint16_t bspUartRxCharsAvail (void)

This function returns the number of data bytes available in the BSP UART RX buffer.

Returns:

Returns the number data bytes available in the BSP UART RX buffer.

11.2.2.11 uint16_t bspUartTxSpaceAvail (void)

This function returns the number of bytes available in the BSP UART TX buffer.

Returns:

Returns the free space in bytes of the BSP UART TX buffer.

11.3 Programming Example

The following code example opens a UART connection at 115200 baud. For more BSP UART code examples, see bsp/srf06eb_cc2538/examples/uart.

```
#include <bsp.h>
#include <bsp_uart.h>

//
// Let BSP UART module allocates handler to
// CC2538 UART interrupt vector
//
#define BSP_UART_ALLOCATE_ISR

//
// Buffers
//
static uint8_t pui8TxBuf[32];
static uint8_t pui8RxBuf[32];

int main(void)
{
    uint8_t ui8Data = 0x34;

    //
    // Initialize buffers and open connection
    //
    bspUartBufInit(&pui8TxBuf, 32, &pui8RxBuf, 32);
    if (bspUartOpen(eBaudRate115200) != BSP_UART_SUCCESS)
    {
        //
        // Failed to open UART connection
        //
        while(1);
    }
}
```

```
    }

    //
    // Write a byte (0x34)
    //
    bspUartDataPut (&ui8Data, 1);

    //
    // Close UART connection
    //
    bspUartClose();

    //
    // Enter infinite loop
    //
    while(1);
}
```


12 References

References and other useful material:

- CC2538 Technical Reference Manual ([SWRU319](#))
- CC2538 Peripheral Driver Library User's Guide ([SWRU325](#))
- SmartRF06 Evaluation Board User's Guide ([SWRU321](#))

13 Document History

Version	Date	Description
SWRU327	2013-04-11	Initial version.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com