SmartRF06EB Board Support Package

Texas Instruments CC26xx Family of Products

User's Guide

Literature Number: SWRU352-prelim-0-2

Document License

Copyright

⚠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 SWRU321**B**). The literature number for this document is in the document footer.

This document was updated on November 06, 2013 (build 11077).

Table of Contents

1 Introduction

The SmartRF06EB Board Support Package (BSP) for CC26xx from Texas Instruments is a set of drivers for accessing the peripehrals found on the SmartRF06EB with the CC26xx family of ARM® Cortex™-M based devices.

The SmartRF06EB BSP uses the CC26xx 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 CC26xx 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_cc26xx` folder.

`examples/` This directory holds SmartRF06EB BSP examples.

`drivers/bin/` This directory holds the precompiled library files for different IDEs.

`drivers/source/` This directory holds the source code for the drivers, including header files.

`drivers/projects/` 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
- I^{2}C™ – Philips Semiconductor Corp
- IAR Embedded Workbench® – IAR Systems
- SPI™ – Motorola

2 Using the SmartRF06EB BSP

2.1 Introduction

The SmartRF06EB BSP for CC26xx 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 CC26xx peripheral driver library to access the CC26xx internal peripheral modules. Therefore, the CC26xx peripheral driver library must also be included in projects using the SmartRF06EB BSP. See Chapter 8 for more information.

The SmartRF06EB BSP for the CC26xx family is released under a standard 3-clause BSD license.

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_cc26xx/drivers/bin`.

2.2.1 IAR Embedded Workbench

All paths in this section are given relative to the `bsp/srf06eb_cc26xx` 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/cc26xx/source` Path to driverlib API definitions
- `../../driverlib/cc26xx/inc` Path to CC26xx 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/cc26xx/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_cc26xx` 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/cc26xx/source` Path to driverlib API definitions
- `../../driverlib/cc26xx/inc` Path to CC26xx 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/cc26xx/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_cc26xx` 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/cc26xx/source` Path to driverlib API definitions
- `../../driverlib/cc26xx/inc` Path to CC26xx register name definitions

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

The SmartRF06EB BSP uses the CC26xx peripheral driver library. To include the precompiled CC26xx peripheral driver library file, `driverlib.lib`, to the project, follow the steps in Section 2.2.

To include the CC26xx peripheral driver library source files to the project, add the .c files in `../../driverlib/cc26xx/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_cc26xx/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

3.1 Introduction

The SmartRF06EB BSP Base functions provide a set of functions for initializing the CC26xx and SmartRF06EB for operation, configuring the SPI interface to the SmartRF06EB peripherals, and controlling the SmartRF06EB 3.3-V domain.

The SmartRF06EB Board Support Pacakge base module source files are contained in `bsp/srf06eb_cc26xx/drivers`.

- `source/bsp.c` contains the function implementations for CC26xx on SmartRF06EB.
- `source/bsp.h` contains the API definitions for use by applications.

3.2 API Functions

Functions

- 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 CC26xx 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 CC26xx 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 CC26xx:

- bspSpiInit()
- 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 bspInit

This function initializes the CC26xx clocks and I/O for use on SmartRF06EB.

Prototype:

```
void
bspInit(uint32_t ui32SysClockSpeed)
```

Description:

The function assumes that an external crystal oscillator is available to the CC26xx. The CC26xx 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:

- **BSP_CLK_SPD_48MHZ**
- **BSP_CLK_SPD_24MHZ**

Returns:

None

3.2.2.2 bspSpiClockSpeedGet

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.

Prototype:

```
uint32_t
bspSpiClockSpeedGet(void)
```

Returns:

Returns the SPI clock speed in Hz

3.2.2.3 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

3.2.2.4 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* = srcClk / n where n is integer, n >= 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_cc26xx/examples` .

The following example initializes the CC26xx to its default clock speed and configures the necessary CC26xx I/O. The CC26xx SPI interface to the SmartRF06EB SPI peripherals is initialized.

```
#include "bsp.h"

//
// Initialize the cc26xx 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

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 CC26xx 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, passing the event mask to it as a void pointer.

The driver files are in `bsp/srf06eb_cc26xx/drivers`.

- `source/io_pin_int.c` contains the function implementations for CC26xx on SmartRF06EB.
- `source/io_pin_int.h` contains the API definitions for use by applications.

4.2 API Functions

Functions

- void ioPinIntRegister (uint32_t ui32Pins, void (∗pfnIntHandler)(void ∗pEvent))
- void ioPinIntUnregister (uint32_t ui32Pins)

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 CC26xx 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 *ui32Pins*. 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 ui32Pins,
                 void (**)(void pEvent) pfnIntHandler)
```

Parameters:

ui32Pins 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 *ui32Pins*)

Unregister the interrupt handler to GPIO pin (or pins) specified by bitmask *ui32Pins*.

Parameters:

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_cc26xx/examples/keys`.

```
//
// Assuming interrupts are disabled
//

//
// Register interrupt handler myIsr() to GPIO port A pin 3
//
ioPinIntRegister(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

5.1 Introduction

The SmartRF06EB has 4 LEDs that can be controlled from the CC26xx. 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 CC26xx.

The driver files are located in `bsp/srf06eb_cc26xx/drivers`.

- `source/bsp_led.c` contains the function implementations for CC26xx on SmartRF06EB.
- `source/bsp_led.h` contains the API definitions for use by applications.

5.2 API Functions

Functions

- void bspLedClear (uint32_t ui32Leds)
- void bspLedInit (uint32_t ui32Leds)
- void bspLedSet (uint32_t ui32Leds)
- void bspLedToggle (uint32_t ui32Leds)

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 *ui32Leds*. This function assumes that LED pins have been initialized by, for example, bspLedInit().

Prototype:

```
void
bspLedClear(uint32_t ui32Leds)
```

Parameters:
ui32Leds is an ORed bitmask of LEDs (for example **BSP_LED_1**).

Returns:
None

5.2.2.2 void bspLedInit (uint32_t *ui32Leds*)

This function initializes GPIO pins connected to LEDs. LEDs are initialized to be off. This function should be called after bspInit().

Parameters:
ui32Leds is an ORed bitmask of LEDs (for example **BSP_LED_ALL**).

Returns:
None

5.2.2.3 void bspLedSet (uint32_t *ui32Leds*)

This function sets LED(s) specified by *ui32Leds*. The function assumes that LED pins have been initialized by, for example, bspLedInit().

Parameters:
ui32Leds is an ORed bitmask of LEDs (for example **BSP_LED_1**).

Returns:
None

5.2.2.4 void bspLedToggle (uint32_t *ui32Leds*)

This function toggles LED(s) specified by *ui32Leds*. The function assumes that LED pins have been initialized by, for example, bspLedInit().

Parameters:
ui32Leds 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_cc26xx/examples/leds`.

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

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

6 Keys

6.1 Introduction

The SmartRF06EB has 5 keys for interfacing the CC26xx. 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 CC26xx.

The driver files are in `bsp/srf06eb_cc26xx/drivers`.

- `source/bsp_key.c` contains the function implementations for CC26xx on SmartRF06EB.
- `source/bsp_key.h` contains the API definitions for use by applications.

6.2 API Functions

Functions

- uint32_t bspKeyGetDir (void)
- void bspKeyInit (uint32_t ui32Mode)
- void bspKeyIntClear (uint32_t ui32Keys)
- void bspKeyIntDisable (uint32_t ui32Keys)
- void bspKeyIntEnable (uint32_t ui32Keys)
- void bspKeyIntRegister (uint32_t ui32Keys, void (∗pfnHandler)(void))
- void bspKeyIntUnregister (uint32_t ui32Keys)
- uint32_t bspKeyPushed (uint32_t ui32ReadMask)

6.2.1 Detailed Description

The SmartRF06EB BSP key driver is by default interrupt driven and uses the CC26xx 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 CC26xx 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 CC26xx 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:

```
uint32_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 (uint32_t *ui32Mode*)

This function initializes key GPIO as input pullup and disables interrupts. If *ui32Mode* is **BSP_KEY_MODE_POLL**, key presses are handled using polling and active state debounce. Functions starting with **bspKeyInt** then do nothing.

If *ui32Mode* is **BSP_KEY_MODE_ISR**, key presses are handled by interrupts, and debounce is implemented using a timer.

Parameters:
ui32Mode 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 (uint32_t *ui32Keys*)

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:

ui32Keys is an ORed bitmask of keys (for example BSP_KEY_1).

Returns:

None

6.2.2.4 void bspKeyIntDisable (uint32_t *ui32Keys*)

This function disables interrupts on specified key GPIOs.

Note:

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

Parameters:

ui32Keys is an ORed bitmask of keys (for example BSP_KEY_1).

Returns:

None

6.2.2.5 void bspKeyIntEnable (uint32_t *ui32Keys*)

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:

ui32Keys is an ORed bitmask of keys (for example BSP_KEY_1).

Returns:

None

6.2.2.6 void bspKeyIntRegister (uint32_t *ui32Keys*, void(∗)(void) *pfnHandler*)

This function registers a custom ISR to keys specified by *ui32Keys*.

Note:

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

Parameters:

ui32Keys 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 (uint32_t *ui32Keys*)

This function clears the custom ISR from keys specified by *ui32Keys*.

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

Parameters:
ui32Keys is an ORed bitmask of keys (for example BSP_KEY_1).

Returns:
None

6.2.2.8 uint32_t bspKeyPushed (uint32_t *ui32ReadMask*)

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:
ui32ReadMask 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_cc26xx/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 LCD

7.1 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:

```
+ ----->   x
| +----------------------------------------------+
| |(0,0)              PAGE 0              (127,0)|
V |                   PAGE 1                     |
  |                    ...                       |
y |                    ...                       |
  |                    ...                       |
  |                    ...                       |
  |                    ...                       |
  |(0,63)             PAGE 7             (127,63)|
  +----------------------------------------------+
```

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_cc26xx/drivers`.

- `source/lcd_srf06eb.c` contains the function implementations for CC26xx 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.

7.2 API Functions

Functions

- void lcdBufferClear (char ∗pBuffer)

- 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)
- 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 lcdSendBuffer (const char ∗pcBuffer)
- void lcdSendBufferPart (const char ∗pcBuffer, uint8_t ui8XFrom, uint8_t ui8XTo, tLcdPage iPageFrom, tLcdPage iPageTo)
- void lcdSetContrast (uint8_t ui8Contrast)

7.2.1 Detailed Description

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

- Functions that manipulate a local buffer on the CC26xx.
- 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 CC26xx 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 CC26xx precompiled library, see Section 2.4.

7.2.2 Function Documentation

7.2.2.1 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

7.2.2.2 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

7.2.2.3 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

7.2.2.4 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

7.2.2.5 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 coulmns [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

7.2.2.6 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

7.2.2.7 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

7.2.2.8 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

7.2.2.9 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

7.2.2.10 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

7.2.2.11 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

7.2.2.12 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

7.2.2.13 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

7.2.2.14 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

7.2.2.15 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

7.2.2.16 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

7.2.2.17 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

7.2.2.18 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

7.2.2.19 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

7.2.2.20 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

7.2.2.21 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

7.2.2.22 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

7.2.2.23 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*.

7.2.2.24 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*.

7.2.2.25 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*

7.2.2.26 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

7.2.2.27 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

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

This function sends the specfied part of *pcBuffer* to the corresponding part on the LCD. This function assumes *ui8XFrom* <= *ui8XTo* and *iPageFrom* <= *iPageTo*. The resolution is given in coulmns [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

7.2.2.29 void lcdSetContrast (uint8_t *ui8Contrast*)

This function sets the LCD contrast.

Parameters:
ui8Contrast is the contrast value [0–63].

Returns:
None

7.3 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_cc26xx/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);
```

8 References

References and other useful material:

- CC26xx Technical Reference Manual (SWRU319)
- CC26xx Peripheral Driver Library User's Guide (SWRU325)
- SmartRF06 Evaluation Board User's Guide (SWRU321)

9 Document History

Version	Date	Description
SWRU352-prelim-0-2	2013-09-11	Rev. 0.2 - Function briefs updated from source.
SWRU352-prelim-0-1	2013-09-11	Rev. 0.1 - Preliminary 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		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265