User manual

DA14580 Software Patch over the Air (SPotA)

UM-B-007

Abstract

This document describes how to set up a DA14850 development board and a DA14850 USB dongle and run the Software Patching over the Air (SPotA) application.

DA14580   DA14580 USB dongle   SPOTA

Contents

Figures

1   Terms and definitions

SPotA	Software Patching Over the Air
SPOTAR	SPotA Receiver
SPOTAI	SPotA Initiator
SPI	Serial Peripheral Interface
SYSRAM	System RAM
RETRAM	Retention RAM
I2C	Inter-Integrated Circuit
BLE SIG	Bluetooth Low Energy Special Interest Group
MSB	Most Significant Byte
LSB	List Significant Byte
UUID	Universal Unique Identifier

2   References

1. AN-B-003 DA14580 Software Patching over the Air (SPotA), Dialog Semiconductor.
2. UM-B-015, DA14580 Software Architecture, Dialog Semiconductor.
3. Connection Manager Help document, Dialog Semiconductor.
4. UM-B-014, DA14580 Development Kit, Dialog Semiconductor.
5. UM-B-005, DA14580 Peripheral Examples, Dialog Semiconductor.
6. AN-B-002, Application and ROM code patching, Dialog Semiconductor.
7. DA14580 Datasheet, Dialog Semiconductor.

3   Introduction

Software Patching over the Air (SPotA) service exposes a control point to allow a peer device to initiate software patching over the air. SPotA defines 2 roles:

- The "SPotA Initiator" is the endpoint which transmits the patch payload.
- The "SPotA Receiver" is the endpoint which receives and applies the patch payload.

Patching code is sent by the Initiator using the Bluetooth Smart link. Receiver stores the patch into the internal RAM or an external non-volatile memory and then applies the patch. Software Patching over the Air (SPotA) is instantiated as a Primary Service. There is only one instance of this service on a device.

SPotA service is a Dialog Semiconductor proprietary service and is described in detail in [1].

4   Software Description

The DA14580 Software Development Kit (SDK) [2] includes one Keil project which implements the Receiver role (SPOTAR):

dk_apps\keil_projects\spota\spotar_fh\fh_spotar.uvproj

The SPOTAR profile is implemented in the following files:

dk_apps\src\ip\ble\hl\src\profiles\spota\spotar\spotar.c

dk_apps\src\ip\ble\hl\src\profiles\spota\spotar\spotar_task.c

The SPOTAR application source code is located in the following directories:

dk_apps\src\modules\app\src\app_project\spotar_fh

dk_apps\src\modules\app\src\app_profiles\spotar

5   Getting Started

This section describes how to set up the application software and the DA14580 Development board to run the SPOTAR application. It is assumed that the user is familiar with using the DA14580 Development Kit (SDK) [4].

STEP 1: Starting SPOTAR device

On a Windows machine, start the Keil environment and open the SPOTAR project:

dk_apps\keil_projects\spota\spotar_fh\fh_spotar.uvproj

Build this project (F7) and follow the steps in the user guide [4] to load the executable to the SDK and run the executable.

STEP 2: Running Connection Manager (for DA14850 USB dongle)

DA14580 USB Dongle will be used as the SPOTAI device. When the dongle is inserted in a Windows machine (e.g. laptop) a J-Link device should be discovered in Windows devices and printers. In J-Link's properties, a JLink CDC UART Port is displayed. The user must keep the virtual COM port number to configure the Connection Manager application.
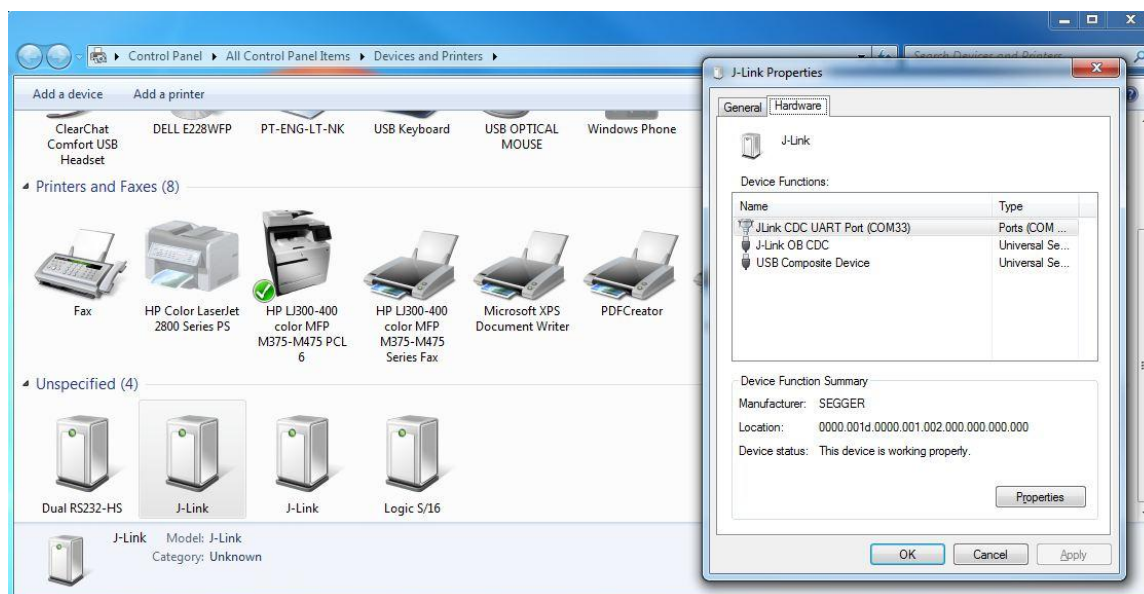
Figure 1: J-Link device discovered

Start the Connection Manager software, on the right pane set the com port to the com port of the JLink CDC UART Port and click the "Boot as Central" button.

STEP 3: Connecting SPOTAI device to the SPOTAR device

In the left pane of the Connection Manager software, click the "Scan" button to discover the SPOTAR device that should be advertising at this point.
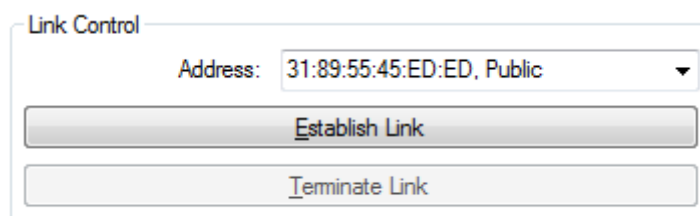

Figure 2: Central Device has discovered advertising device

If the SPOTAR device is discovered, its public address will appear in the "Link Control" section as seen in Figure 2. At this point, a connection to the SPOTAR device can be accomplished by clicking the "Establish Link" button.

STEP 4: Reading the SPOTAR service database

After the connection is established successfully, the entire attribute database is displayed on the bottom pane of the Connection Manager software. To understand the characteristics of the profile the application note [3] needs to be studied carefully.

STEP 5: Configuring the SPOTAR device

At this point it is assumed that the user is familiar with the profile and its characteristics. Note that the SPotA service is a Dialog Semiconductor proprietary service. The Bluetooth SIG has assigned a 16 bit UUID for Dialog SPOTAR service which is: **0xFEF5**. The UUIDs of the characteristics of the profile are proprietary 128 bits UUIDs as defined in [3].

0x0019	2800	-	Primary Service Definition
0x001A	2803	-	Characteristic Declaration
0x001B	8082CAA841A6402191C656F9B954CC34	Read, Write	User-defined UUID
0x001C	2803	-	Characteristic Declaration
0x001D	724249F05EC34B5F880442345AF08651	Read, Write	User-defined UUID
0x001E	2803	-	Characteristic Declaration
0x001F	6C53DB2547A145FEA0227C92FB334FD4	Read	User-defined UUID
0x0020	2803	-	Characteristic Declaration
0x0021	9D84B9A3000C49D89183855B673FDA31	Read, Write	User-defined UUID
0x0022	2803	-	Characteristic Declaration
0x0023	457871E8D5164CA1911657D0B17B9CB2	Read, Write	User-defined UUID
0x0024	2803	-	Characteristic Declaration
0x0025	5F78DF94798C46F5990AB3EB6A065C88	Read, Notify	User-defined UUID
0x0026	2902	-	Client Characteristic Configuration

Figure 3: SPOTAR attribute table

Figure 3 shows the SPOTAR attribute table. The Characteristic UUIDs in this implementation are as follows:

8082caa841a6402191c656f9b954cc34 => SPOTA_MEM_DEV_UUID
724249f05ec34b5f880442345af08651   => SPOTA_GPIO_MAP_UUID
6c53db2547a145fea0227c92fb334fd4 => SPOTA_MEM_INFO_UUID
9d84b9a3000c49d89183855b673fda31 => SPOTA_PATCH_LEN_UUID
457871e8d5164ca1911657d0b17b9cb2 => SPOTA_PATCH_DATA_UUID
5f78df94798c46f5990ab3eb6a065c88   => SPOTA_SERV_STATUS_UUID

The example that is described below refers to a software patch of 11 words that will be stored in to an external SPI Flash memory [5].
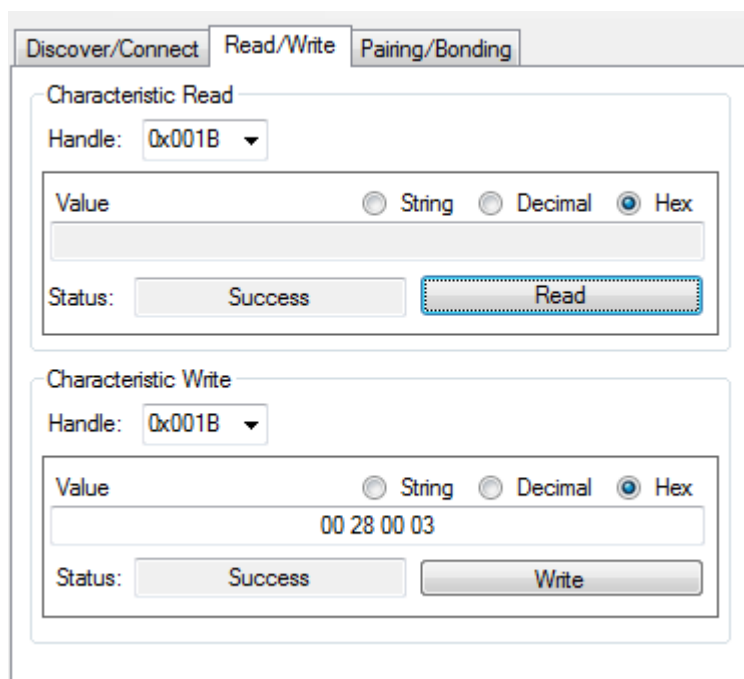
Enable SPOTAR status notifications

This is an optional step. If the Initiator, the client device in this case (Connection Manager & USB dongle), wishes to receive SPOTAR status notifications, it should enable the notifications first. To achieve this, it must write a value of 0x0001 to the client characteristic configuration descriptor for the SPOTA_SERV_STATUS characteristic. The handle for the client characteristic configuration descriptor immediately follows the characteristic value's handle. Therefore, a value of 0x0001 must be written to handle 0x0026. In the lowest pane of the Connection manager software, click on the handle 0x0026 to select the characteristic, select "Hex" as the format of the value and then in the "Characteristic Write" section, enter "01 00" in the "Value" field (note that the LSB is entered first, and the MSB is entered last). Click the "Write" button. The status box will display "Success", indicating that the write was successful.

Set memory base address and memory device

Firstly, in order for the Initiator to indicate to the Receiver that wishes to start the SPotA service, it has to write the SPOTA_MEM_DEV characteristic value.

In this example, the Initiator wishes to write the patch into external SPI Flash memory at address 0x002800. In the Connection Manager software click on the "Read/Write" tab. Then, in the "Characteristic Write" section, select the 0x001B handle (which is the handle of SPOTA_MEM_DEV) and in the value filed enter: "00 28 00 03", as shown in Figure 4, and click the "Write" button. Note that "Hex" has been selected as the format of the value.

To test if the value has been written correctly, the value of the characteristic can be read in the "Characteristic Read" section. Also the status box will display "Success", indicating that the write was successful.

Figure 4: Write Characteristic value example

Set GPIO mapping for SPI FLASH

The next characteristic that should be set by the Initiator is SPOTA_GPIO_MAP. In the "Characteristic Write" section, select the 0x001D handle and in the value filed enter: "00 03 06 05"

This value means: 00:P0_0/CLK, 03:P0_3/CS, 06:P0_6/MOSI, 05:P0_5/MISO.

Read patch information from memory

At this point, the Initiator can read the overall patch length and actual number of patches from the memory if there are any patches written in the past. In the "Characteristic Read" section, select the 0x001F handle to read SPOTA_MEM_INFO characteristic value.

Set overall patch data length

At this point, the Initiator shall set the length of the patch data that will be sent. Note that the length represents byte count. For our example the overall patch length is 44 bytes (or 11 words). In the "Characteristic Write" section, select the 0x0021 handle and in the value filed enter: "2c 00" and click the "Write" button to write the SPOTA_PATCH_LEN characteristic value. Please note that "Hex" shall be selected as the format of the value.

Send the patch data

The next characteristic that should be set by the Initiator is SPOTA_PATCH_DATA. In this example, the patch that the Initiator wishes to send is three patch entries of **11 words total length**. Refer to [6] and [7] to understand how to construct patch headers. Below, the data of the patch are listed and a short explanation is given:

    a. Block copy of 7 words at start address 0x7000 (offset from 0x20000000 system RAM base).

        0x00077000   // 1st Patch header: Block copy of 7 words at start address 0x7000.

        0x12345678   // 1st of the 7 words.

        0xAAAAAAAA   // 2nd of the 7 words.

        0xBBBBBBBB   // and so on…

        0xCCCCCCCC

        0xDDDDDDDD

        0XEEEEEEEE

0XFFFFFFFF   // last word of the 1st patch (7th word).

b. Dummy patch entry

0x00007000   // **2nd Patch header**: Dummy patch header.

c. Block copy of 1 word, update memory 0x701c

0x0001701C   // **3rd Patch header**: Block copy of 1 word at 0x701c.

0xCAFEDEAD // the word to be copied.

Now that the patch has been constructed, the Initiator needs to write all these words in chunks of 20 bytes (max data MTU that is supported for a single characteristic write) in the *"Characteristic Write"* section. So the following byte sequence should be written to the handle 0x0023:

     i.   00 70 07 00 78 56 34 12 AA AA AA AA BB BB BB BB CC CC CC CC

    ii.   DD DD DD DD EE EE EE EE FF FF FF FF 00 70 00 00 1C 70 01 00

   iii.   AD DE FE CA

As illustrated in this example, the patch data shall be sent in the following order: the first patch header first and least significant byte first.

NOTE: At this point the Initiator can read the overall patch length and actual number of patches from the memory. In the *"Characteristic Read"* section, select the 0x001F handle to read SPOTA_MEM_INFO characteristic value. The following value should be displayed: *"0B 00 03 00"*, meaning that 3 patches are already written in the memory and the overall length is 11, 32-bit words.

Execute the software patch

Assuming that the patch has been received correctly by the SPOTAR device, then when the SPOTAR device wakes up from sleep or after reset, the patch will be applied. In this example, after the patch has been applied, the system RAM memory area at address 0x20007000 will have the values shown in Figure 5.

```
Memory 1

Address: 0x20007000

0x20007000: FFFFFFFF EEEEEEEE DDDDDDDD CCCCCCCC BBBBBBBB AAAAAAAA 12345678 CAFEDEAD 93C17F78
0x20007024: 6155143E D9C3E935 CEDBD571 3A3708DD 52291192 0517EF5C 3C7F3D54 8B302A96 216E4CF8
0x20007048: 7BE19E69 2BC47992 47FF68DE 32CC79A0 EF07A63F A5228FB1 154871D0 292766D3 13FDD6C8
0x2000706C: 89F5714F B54524FD 1FB174E9 419514AB BEB194EC FBD062B2 E11B8EE2 9428A80D 8FB99EAE
0x20007090: F3BB4DCE A09E466A 2ADCBF2D 9132A1F8 A444315F D36E71FD 4C8BDAC1 759F75FE E8C6F771
```

Figure 5: Memory area after applying patch

6   SPOTAI host application

The DA14580 Software Development Kit (SDK) [2] also includes a Windows Visual C++ project which implements the Initiator application (SPOTAI). A command line driven console application has been developed and can be also used to test the SPOTA implementation. The source code can be found under the following directory:

    host_apps\windows\spota\initiator

This application, in conjunction with the DA14580 USB Dongle, implements an external processor solution of SPOTAI. Therefore, to run this application from a Windows console it is assumed that the DA14580 USB Dongle is already installed and can be used by the Windows machine.

SPOTAI application command description:

```
host_spotai <com port number> <bdaddr> <.bin file> <mem_dev_opts>
```

<com port number> = the com port number for the USB Dongle DA14580 for the SPOTA initiator. E.g. 16 for COM16.

<bdaddr> = the BD address of the target SPOTA receiver device. E.g. 25:07:19:77:23:01.

<.bin file> = the binary file containing the patch data. The file is expected to store the words in LE order.
E.g. if the patch includes 1 patch entry with the following 2 words:
```
0x0001701C
0xCAFEDEAD
```
then the binary file shall contain the following bytes:
```
1C 70 01 00 AD DE FE CA
```

<mem_dev_opts> = sysram | retram | i2c <i2c_dev_opts> | spi <spi_dev_opts>

<i2c_dev_opts> = <patch base addr> <I2C device addr> <SCL gpio> <SDA gpio>

<spi_dev_opts> = <patch base addr> <MISO gpio> <MOSI gpio> <CS gpio> <SCK gpio>

<patch base addr> = the patch base address in the target memory device. It is a HEX value, e.g. A000

<I2C device addr> = the I2C slave address for the I2C memory device. It is an 8-bit HEX value, e.g. 0B.

<XXX gpio> = the name of the gpio that is assigned for the XXX function, e.g. P2_2

Command line examples:

- **SYSRAM**:

  host_spotai 56 25:07:19:77:23:01 patch.bin sysram

- **RETRAM**:

  host_spotai 56 25:07:19:77:23:01 patch.bin retram

- **I2C**:

  host_spotai 56 25:07:19:77:23:01 patch.bin i2c 800 50 P0_2 P0_3

- **SPI**:

  host_spotai 56 25:07:19:77:23:01 patch.bin spi 2800 P0_5 P0_6 P0_3 P0_0

Appendix A   Notes to Developers

A.1   SPOTAR Memory requirements

As explained in [1], patch code is downloaded using the Bluetooth Smart link, and the SPOTAR device can store the patch into the internal RAM or an external non-volatile memory. The software patch is eventually executed as described in [6]. There are of course certain constraints with respect to the physical storage of the patching code. In this appendix, a detailed explanation is given for the implementation of the patch data handling functions so that developers can understand and be able to modify the code according to their needs.

Storing patch into Internal RAM

Two memory areas have been reserved in order to handle the software patch data for the Internal RAM implementation:

```
uint8_t spota_new_pd[SPOTA_NEW_PD_SIZE]
__attribute__((section("spotar_patch_area"),zero_init));

uint8_t spota_all_pd[SPOTA_OVERALL_PD_SIZE]
__attribute__((section("spotar_patch_area"),zero_init));
```

`SPOTA_NEW_PD_SIZE`: Is the size of the maxim individual software patch expected.
`SPOTA_OVERALL_PD_SIZE`: Is the size of the overall patch space that holds all the individual patches during the lifetime of the device.

It is advised that sleep mode is disabled during software patching in the SPOTAR device. However, if sleep is enabled, then the above memory areas need to be reserved in retention RAM. The following define, in *da14580_config.h*, determines if these two buffers should be in retention RAM or non-retainable RAM:

```
#ifdef CFG_PRF_SPOTAR
// Placed in the RetRAM when SPOTAR_PATCH_AREA is 0 and in SYSRAM when 1
#define SPOTAR_PATCH_AREA   0
#endif
```

The header file is included in the *scatterfile_common.sct* where according to the define value the `spotar_patch_area` section is reserved in the relevant RAM section. It is important to note that if the SPOTAR project has been compiled with this define set to "1", then the SPOTA_MEM_DEV shall not be set to select retention RAM memory device. Equally, if compiled with the define value set to "0", then SPOTA_MEM_DEV shall not be set to select RAM memory device. Therefore, the developer should select the right memory area according to his/her needs and configure SPOTAR accordingly. The SPI and the I2C memory devices are not affected by this define.

As seen in paragraph 5, step 5, the patch data are sent in chunks of 20 bytes. SPOTAR device stores these data into spota_new_pd[] area. When the number of the data received matches the patch length specified by the Initiator, then the CRC is calculated, and if correct, the received patch is stored into spota_all_pd[] area. This is implemented in function app_spotar_pd_hdlr(). Note that the new patch is written into spota_all_pd[] area staring at the memory location where the old patch(es) end.

Note 1   If patch data are sent in the reversed order, i.e. least significant word first, then the spota_new_pd[]is not really required. The new patch data could be copied to the spota_all_pd[] directly in the order they are received. However, in the case the new patch data has not received correctly (CRC fails or any other protocol error) then the incomplete patch that has been written to spota_all_pd[] must be deleted.

Note 2   CRC calculation is not implemented in the example code. It is left to the developer to choose a CRC method and polynomial length.

Storing patch into external non-volatile memory

The same two memory areas are used in this case. As in the case of internal RAM storage, the `spota_new_pd[]` is used to store the new patch data as they received. After the entire new patch has been received successfully, the patch data are written to SPI FLASH or I2C EEPROM using the

functions `spi_flash_write_data()` and `i2c_eeprom_write_data()`, respectively. The example memory access functions are written for the following memory devices:

● SPI flash Winbond W25X10CL

● I2C EEPROM STMicroelectronics M24M01

The spota_all_pd[]area is used to store the overall patch when it will be fetched in the system RAM from the external non-volatile memory before the patch is executed.

Note 3   Reserving memory space for the `spota_new_pd[]` can be avoided. New patch data can be stored in to `spota_all_pd[]` and when the entire patch is received successfully it can be written into the external non-volatile memory. However, since the example SPOTAR application supports all memory configurations, the `spota_new_pd[]` is used for code simplicity.

A.2   Patch Execution

If the software patch has been stored successfully, it will be executed every time the main_func() in arch_main.c is called (after reset or when system wakes up from deep sleep). In this case, the function app_spotar_exec_patch() is called to execute the patch. In the example application code, it can be seen that the function exec_patching_spota() is finally called:

void exec_patching_spota(WORD mem_dev, WORD gpio_map, WORD* ptr, WORD patch_length)

If the SPOTAR device supports more than one memory patch areas, then the exec_patching_spota() function shall be called sequentially for every memory area where a patch is stored.

Note 4   In the example application code, when the `exec_patching_spota()` is called, the GPIO map parameter is hardcoded. The GPIO map characteristic value SPOTA_GPIO_MAP cannot be read from the profile data base after reset or after system wake up since it will be uninitialized. However, this should not be a problem considering that the GPIO map of the external non-volatile memory is known for a particular device and it should be fixed.

Note 5   When running in deep sleep, the patch needs to be executed every time the system wakes up from deep sleep. In this case the patch execute function call needs to be inside the main while(1) loop in arch_main(). If the patch has been saved in to an external memory it is very important that the time required for reading the patch from the external memory does not take too long and delays the BLE scheduling. Generally speaking, to be safe, it should finish the patch read and execution before the BLE_SLP_IRQ interrupt is triggered.

Note 6   If we apply the patch in SYSTEM RAM, then the system should never enter deep sleep otherwise the patch data will be lost. In this case, the patch execute function needs to run straight after the patch has been successfully downloaded. An example is provided in demo application and can be found in function `app_spotar_pd_hdlr()`:

```
// Apply patch if SYSRAM has been selected. Can not reset system when patch
// is stored in SYSRAM.
if( (mem_info & 0xffff) > 0)
{
    exec_patching_spota((mem_dev & 0x0000ffff), 0x0,(WORD *) spota_all_pd, (mem_info & 0xffff) );
}
```

7 Revision history

Revision	Date	Description
1.0	18-Mar-2014	Initial version

Status definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

Disclaimer

Information in this document is believed to be accurate and reliable. However, Dialog Semiconductor does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. Dialog Semiconductor furthermore takes no responsibility whatsoever for the content in this document if provided by any information source outside of Dialog Semiconductor.

Dialog Semiconductor reserves the right to change without notice the information published in this document, including without limitation the specification and the design of the related semiconductor products, software and applications.

Applications, software, and semiconductor products described in this document are for illustrative purposes only. Dialog Semiconductor makes no representation or warranty that such applications, software and semiconductor products will be suitable for the specified use without further testing or modification. Unless otherwise agreed in writing, such testing or modification is the sole responsibility of the customer and Dialog Semiconductor excludes all liability in this respect.

Customer notes that nothing in this document may be construed as a license for customer to use the Dialog Semiconductor products, software and applications referred to in this document. Such license must be separately sought by customer with Dialog Semiconductor.

All use of Dialog Semiconductor products, software and applications referred to in this document are subject to Dialog Semiconductor's Standard Terms and Conditions of Sale, unless otherwise stated.

RoHS Compliance

Dialog Semiconductor complies to European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2).
Dialog Semiconductor's statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

Contacting Dialog Semiconductor

Germany Headquarters
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

United Kingdom
Dialog Semiconductor (UK) Ltd
Phone: +44 1793 757700

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822

Email:
enquiry@diasemi.com

North America
Dialog Semiconductor Inc.
Phone: +1 408 845 8500

Japan
Dialog Semiconductor K. K.
Phone: +81 3 5425 4567

Taiwan
Dialog Semiconductor Taiwan
Phone: +886 281 786 222

Web site:
www.dialog-semiconductor.com

Singapore
Dialog Semiconductor Singapore
Phone: +65 64 849929

China
Dialog Semiconductor China
Phone: +86 21 5178 2561

Korea
Dialog Semiconductor Korea
Phone: +82 2 3469 8291