User manual

DA14580/581/583 Software architecture

UM-B-015

Abstract

This document describes the software architecture of the DA14580/581/583 Software Development Kit. The ROM/RAM code division is explained, the APIs for application development and the development tools are described.

Contents

Figures

Tables

1 Terms and definitions

AES	Advanced Encryption Standard
BLE	Bluetooth Low Energy
CPU	Central Processing Unit
GAP	Generic Access Profile
GTL	Generic Transport Layer
HCI	Host Controller Interface
HW	Hardware
NVDS	Non-Volatile Data Storage
OTP	One Time Programmable (memory)
SDK	Software Development Kit
SoC	System on Chip
SPotA	Software Patching over the Air
SW	Software

2 References

[1] DA14580 Data sheet, Dialog Semiconductor
[2] RW-BLE Host Interface Specification (RW-BLE-HOST-IS), Riviera Waves
[3] RW-BLE Host Software (RW-BLE-HOST-SW-FS), Riviera Waves
[4] RealView Development Suite Getting Started Guide, ARM Ltd
[5] UM-B-014, DA14580/581 Development Kit, User manual, Dialog Semiconductor
[6] Bluetooth Specification Version 4.0
[7] Riviera Waves Kernel (RW-BT-KERNEL-SW-FS), Riviera Waves
[8] GAP Interface Specification (RW-BLE-GAP-IS), Riviera Waves
[9] Proximity Profile Interface Specification (RW-BLE-PRF-PXP-IS), Riviera Waves
[10] UM-B-003, DA14580_581_583 Software development guide, User manual, Dialog Semiconductor
[11] UM-B-008, DA14580_581_583 Production test tool, User manual, Dialog Semiconductor
[12] UM-B-013, DA14580/581 External processor interface over SPI, User manual, Dialog Semiconductor
[13] UM-B-007, DA14580 Software Patching over the Air (SPotA), User manual, Dialog Semiconductor
[14] UM-B-011, DA14580/581 Memory file and scatter file, User manual, Dialog Semiconductor
[15] UM-B-004, DA14580_581_583 Peripheral drivers, User manual, Dialog Semiconductor
[16] UM-B-012, DA14580/581 Creation of a secondary boot loader, User manual, Dialog Semiconductor
[17] UM-B-004, DA14580_581_583 Peripheral examples, User manual, Dialog Semiconductor
[18] UM-B-008, DA14580/581 Sleep mode configuration, User manual, Dialog Semiconductor
[19] UM-B-010, DA14580_581_583 Proximity application, User manual, Dialog Semiconductor
[20] DA14581 Data sheet, Dialog Semiconductor

3 Introduction

One of the software components of the Dialog DA14580/581/583 development kit is the BLE Software Development Kit (SDK). The SDK implements the Bluetooth® Smart protocol as specified in Version 4.0 of the Bluetooth® standard and is fully compliant with this standard. It is a single-mode BLE implementation, which means that there is no support for the Basic Rate / Enhanced Data Rate protocol (BR/EDR).

4 Overview

The BLE core protocol stack is a third party implementation licensed from Riviera Waves. Therefore, the SDK only “exposes” the source code of the application API layer and the rest of the BLE core stack is delivered as object code (BLE core library). This document provides an overview of the software architecture and describes the application API layer. An overview can be seen below for easy reference.

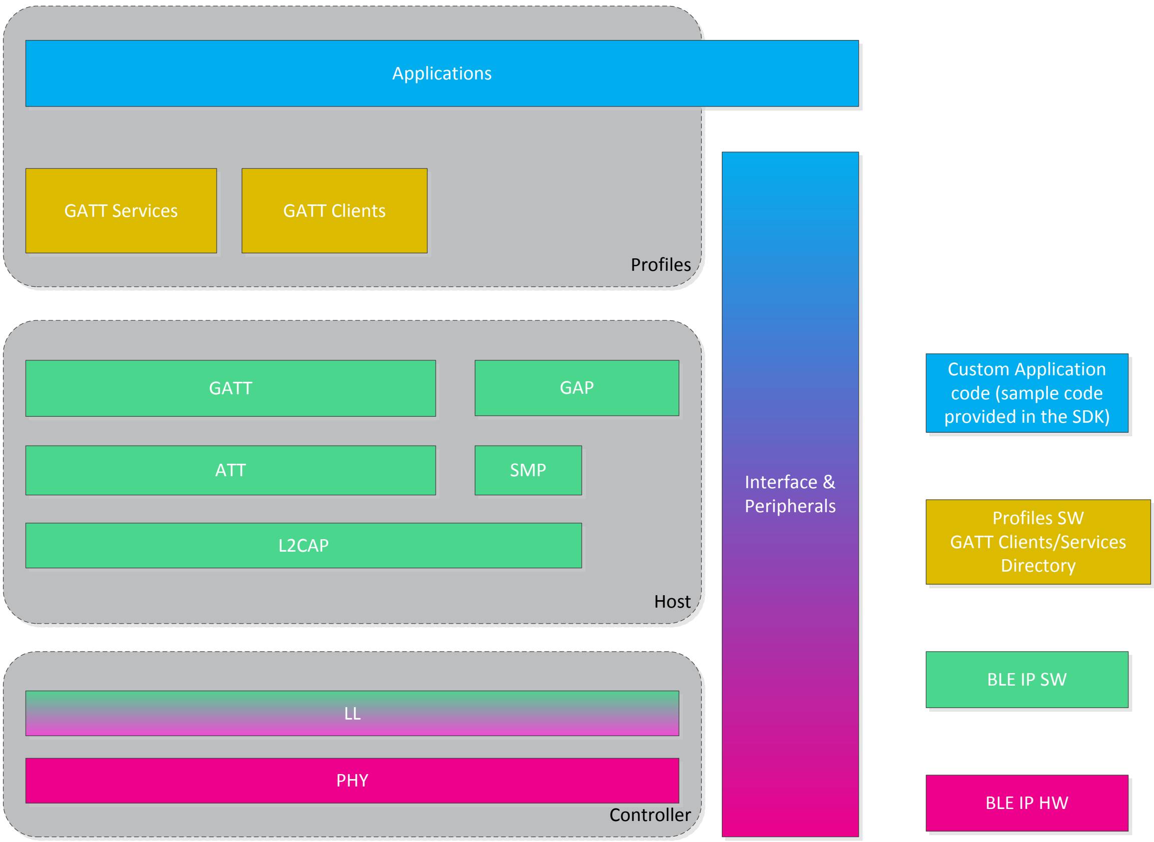


Figure 1: BLE protocol stack

5 BLE software development kit

The DA14580/581/583 SDK is a complete software platform for developing single-mode BLE applications. It is based on the DA14580/581 complete System on Chip (SoC) solutions. The DA14580/581/583 comprises the ultra-low power ARM Cortex M0, dedicated hardware for the Link-Layer implementation of the BLE, a 2.4 GHz RF transceiver, 84 kB ROM, 32 kB One-Time-Programmable memory (OTP) for storing Bluetooth profiles as well as custom application code, up to 42 kB of SRAM (8 kB retention RAM) and a full range of peripheral interfaces. For more information on the DA14580/581/583 refer to the data sheet [1].

Depending on the application HW processor configuration, the DA14580/581/583 SDK proposes different SW configurations.

Integrated processor configuration: The application and BLE layers (control and host) are implemented in DA14580/581/583 chip. It corresponds to Fully_Hosted configuration mode as it's described in Riviera Waves documents. The proximity reporter project and the template project are typical examples of the integrated processor configuration.

External processor configuration: The application is implemented in an external processor while the link layer and host protocols and profiles are implemented in DA14580/581/583 chip. It corresponds to Fully_Embedded configuration mode as it's described in Riviera Waves documents. Projects with names containing **'_ext'** are based on the external processor configuration.

5.1 Integrated processor configuration

The associated SW configuration is straightforward: all SW components, lower layers (controller), higher layers (host), profiles and application run on the DA14580/581/583 as a single chip solution.

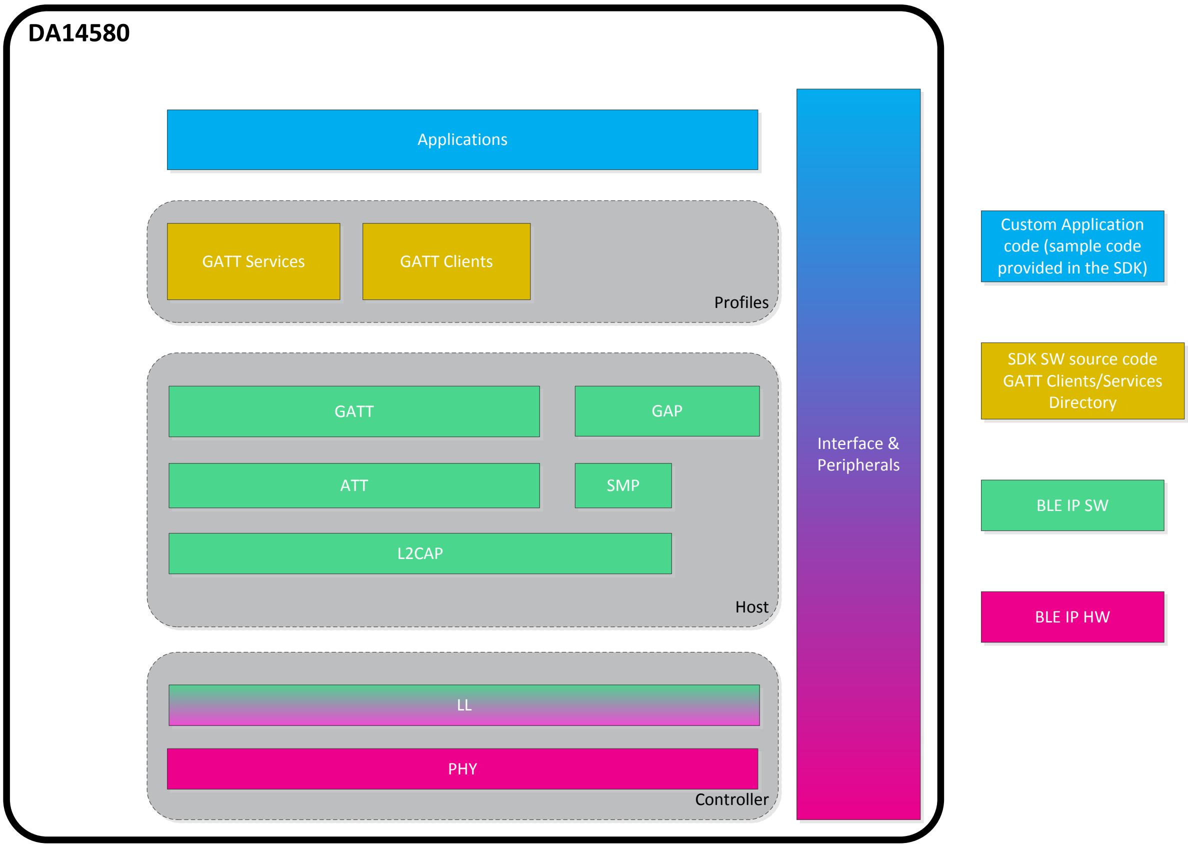


Figure 2: Integrated processor SW configuration

5.2 External processor configuration

In the external processor configuration, the application is implemented in an external processor while the link layer, the host protocols and the profiles are implemented in DA14580/581/583 chip. These two components communicate via a proprietary HCI [2], i.e. Generic Transport Layer (GTL) over UART. This configuration is useful for applications that run on an external microcontroller.

More information on the external processor configuration as well as an example application is described in [2].

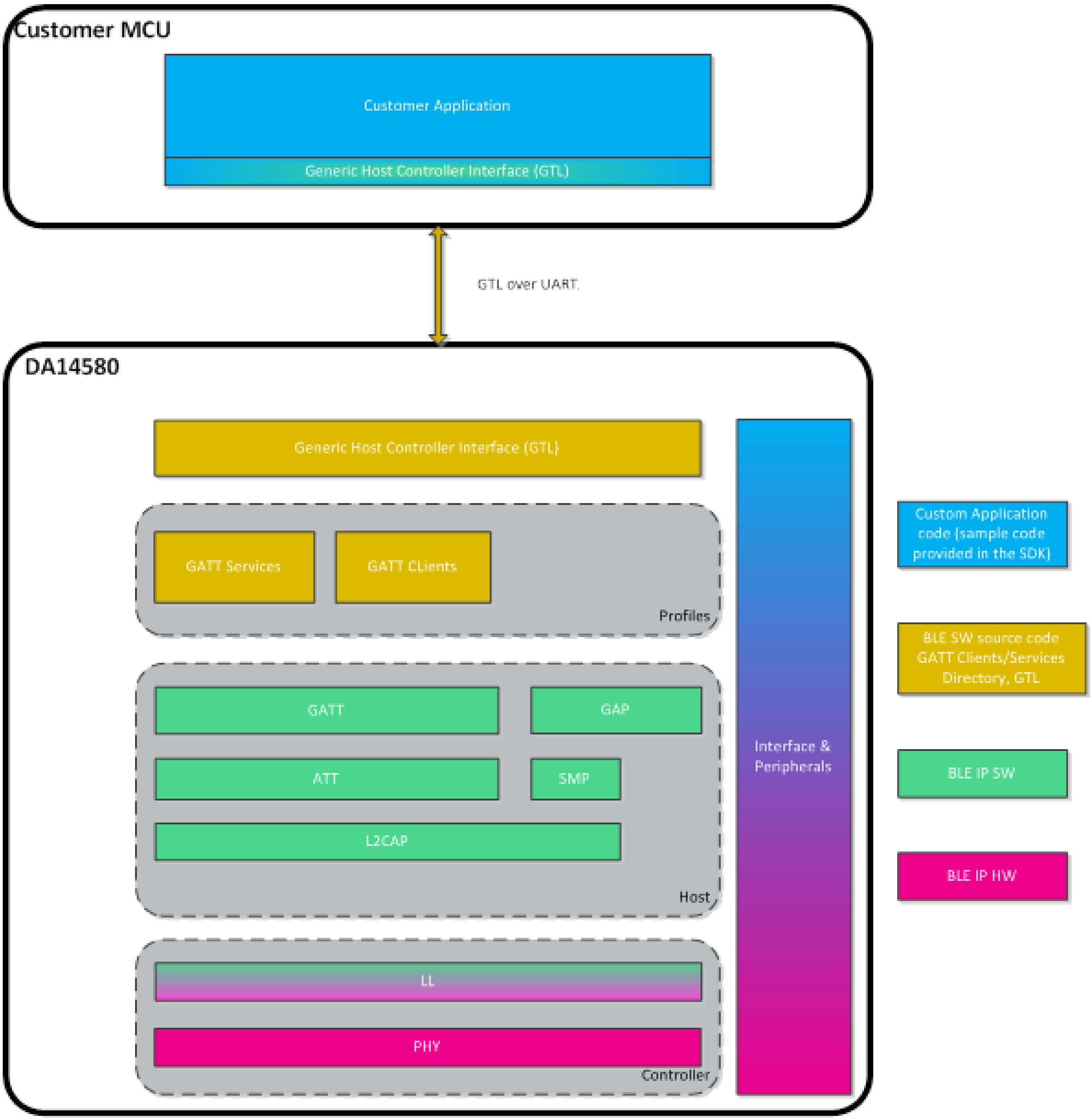


Figure 3: GTL interface

6 Software structure overview

6.1 ROM/RAM code

The DA14580/581/583 SDK stack consists of two major sections: the ROM code and the RAM code:

ROM code

This code resides in the DA14580/581/583's dedicated ROM and implements the BLE protocol stack from the GAP layer (inclusive) downwards. Since this code is already stored in ROM, only the symbol definitions are provided in the file *rom_symdef.txt* (dk_apps/misc/rom_symdef.txt) so that the entire project code can be linked into a single executable.

RAM code

This code will be loaded in the DA14580/581/583's RAM. It includes the various application profiles and the applications. The full source code of the sample applications is provided so that the application developer can use the API to develop specific applications and extend the functionality or develop new application profiles.

6.2 Code directory tree

This section presents an overview of the directory structure. The root directory of the SDK directory contains the subfolders shown in Figure 4. These directories are described in the following sections.

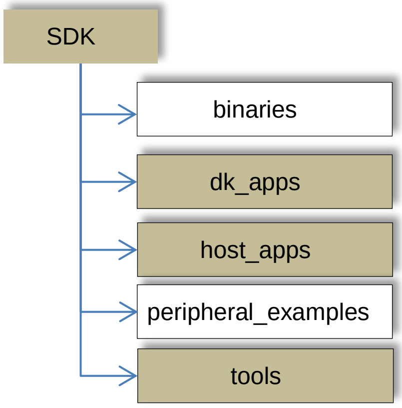


Figure 4: SDK root directory

6.2.1 binaries directory

This directory holds the executable binaries of the PC applications stored in host_apps directory as well as the binary file of the production test tool firmware. These binaries are provided so that the developer can run/test the applications with no need to compile the projects.

6.2.2 dk_apps directory

The Development Kit application directory (dk_apps) holds all the necessary folders (see Figure 5) needed for DA14580/581/583 application development. Below follows a short description for each folder.

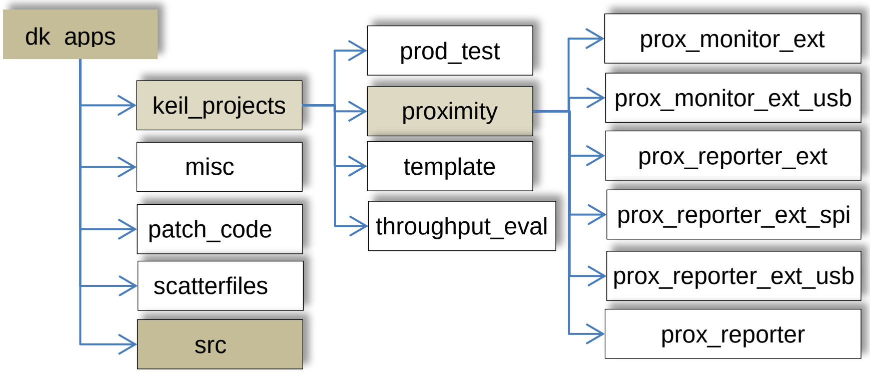


Figure 5: dk_apps directory

6.2.2.1 keil_projects directory

Under this directory, there are application name folders that hold the Keil environment projects for the applications supported by the SDK. Project files for Keil uVision 4 and Keil uVision 5 are provided.

- **prod_test**: Keil project for the production test tool firmware. More information for the production test tool is given in [11].
- **proximity**: Keil projects for the proximity applications provided as examples in SDK distribution.
  - **prox_monitor_ext**: Proximity monitor Keil project. The PC application is stored in host_apps directory.
  - **prox_monitor_ext_usb**: Proximity monitor Keil project for the USB dongle. The configuration settings are the only difference with the monitor_fe project.
  - **prox_reporter_ext**: Proximity reporter Keil project. The PC application is stored in host_apps directory.
  - **prox_reporter_ext_spi**: Proximity reporter Keil project. It supports the external processor over SPI interface. More information is given in [12].
  - **prox_reporter_ext_usb**: Proximity reporter Keil project for the USB dongle. The configuration settings are the only difference with the reporter_fe project.
- **throughput_eval**: Keil project for the throughput evaluation ventra and peripheral applications.
- **template**: This contains a template project used as an example in [10].

6.2.2.2 misc directory

The ROM code symbols file *rom_symdef.txt* is located in this directory. This file will be used as input into the linker to create the final executable. The executable file as well as the compilation outputs are saved in a newly created directory named **out**.

6.2.2.3 patch directory

This directory contains the object files of the patched ROM functions. More information for the patched functions is given the Release Notes of the SDK distribution.

6.2.2.4 scatterfiles directory

This directory contains the ARM M0 microprocessor scatter files. A scatter file is used for defining the memory layout in the microcontroller. This allows a more complex memory layout to be created. For more information regarding the M0 scatter files see [4]. The memory map and scatter file structure is described in details in [14].

6.2.2.5 src directory

The structure of the src directory is illustrated in Figure 6.

- **dialog:** This directory contains the SDK specific header files. ARM M0 header files and DA14580/581 register header files.
- **ip:** This directory contains the header files for the source code of the BLE core that is stored in ROM (the host, the controller, hci, rwble).
- **modules:** This directory contains the application API source code (app directory) and the sample applications [3]. It also contains the kernel API, the Non Volatile Data Storage (Appendix A) and the RF preferred settings (Appendix B). The app directory is described in a separate paragraph, below.
- **plf:** This directory contains platform specific code.
  - **arch**: This contains the system files and the main() application function.
  - **drivers**: This contains the peripheral drivers. More information is given in [15].

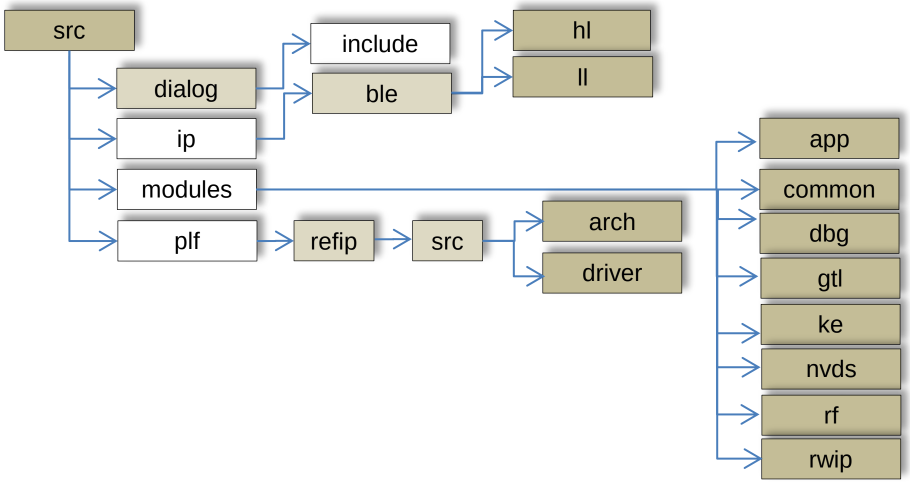


Figure 6: src directory

6.2.2.6 app directory

This directory holds the application projects, the profiles and some utilities common to all application projects. See Figure 7.

- **api**: This contains common header files for all user applications.
- **src**: This holds applications project specific code and handling functions for operations like connect, encryption, advertise, etc
- **src/app_profiles**: This holds the source code of the supported profiles. A list of the certified profiles is given in the Release Notes.
- **src/app_project**: This holds the Keil projects of the user application examples. The subfolder **system** contains the configuration settings for the peripherals and the API for the sleep mode configuration [18].
- **src/app_utils**: This holds a set of utilities for storing bonding data, handling LEDs and buttons, enabling debug console.

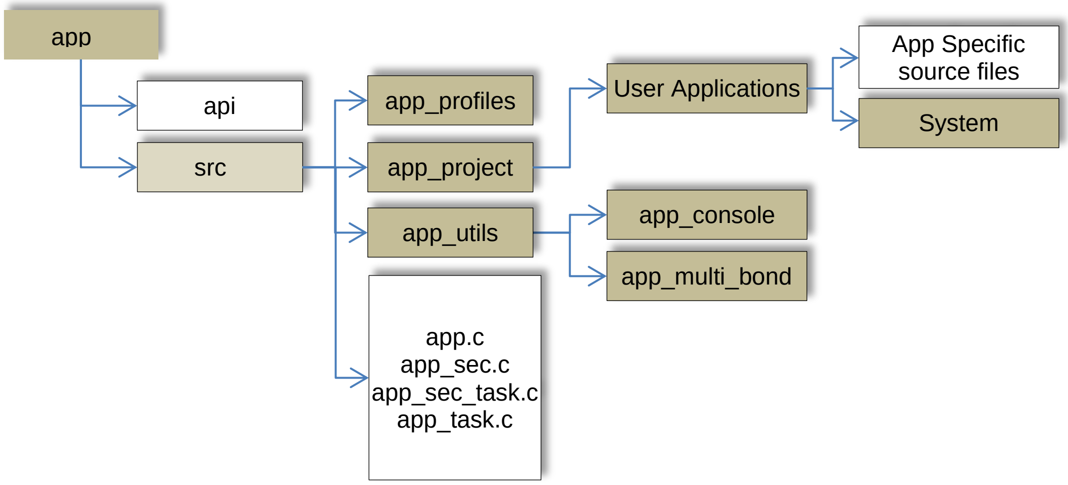


Figure 7: app directory

6.2.3 host_*apps* directory

This directory holds the applications that run on an external processor (PC or other CPU). Basically it contains the proximity, SPotA and SUOTA initiator applications that run on PCs and the application example for the proximity reporter over proprietary SPI interface [12].

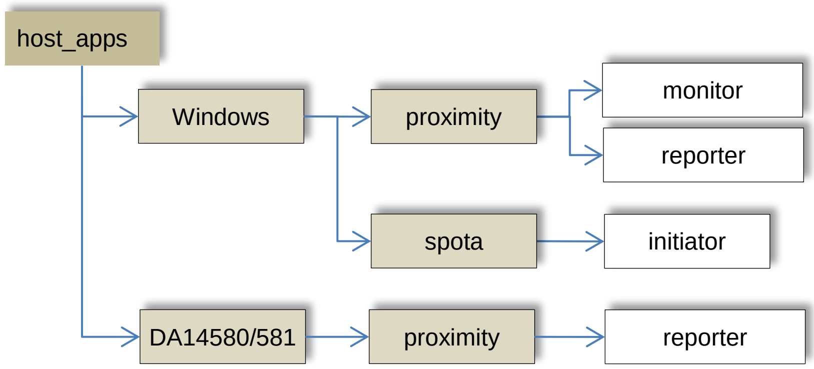


Figure 8: host_apps directory

6.2.4 peripheral_examples directory

This directory holds the peripheral examples application. It provides a set of useful examples for the main peripherals and device drivers supported by the DA14580/581 SDK. More information is given in [17].

6.2.5 tools directory

This directory holds the Keil projects of the tool applications: secondary_bootloader [16], flash_programmer and prod_test [11]. See Figure 9.

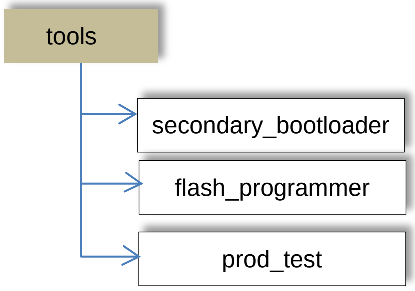


Figure 9: tools directory

6.3 Software configuration

6.3.1 Integrated or external processor operation mode

In the previous sections the **integrated processor** and **external processor** software configurations have been described. These two modes correspond to **full-hosted** and **full-embedded** configuration modes as they are described in Riviera Wave documents.

The application developer can configure the mode of operation at compile time using the first element of the jump table as follows:

```
const uint32_t* const jump_table_base[88] __attribute__((section
("jump_table_mem_area"))) =
{
        #if (BLE_APP_PRESENT)
          (const uint32_t*) TASK_APP,      // Integrated processor
        #else
          (const uint32_t*) TASK_GTL,      // External processor
        #endif
```

As explained in the comments section, if an application has been compiled in, the first item of the array is set to TASK_APP and the integrated processor mode is selected. When no application has been compiled in, the first item of the array is set to TASK_GTL (Generic Transport Layer) and the External processor mode is selected. During run-time, the software execution will check the value of the first element of the array and the relevant code will be executed.

6.3.2 Configuration directives

All DA14580/581/583 SDK projects pre-include a configuration header file (*da14580_config.h*) residing in Keil project's directory. Directives defined in *da14580_config.h* modify various settings of the application.

Table 1: Project configuration

Directive	Defined	Undefined
CFG_APP	Integrated host application	External processor host application
CFG_PRF_<profile>	Profile included	Profile not included
CFG_APP_<application>	Application identifier. Must be defined for all integrated host applications.	
CFG_NVDS	Non Volatile Data Storage (NVDS) structure used (Appendix A)	NVDS structure not used
CFG_APP_SEC	Includes BLE security	Excludes BLE security

Directive	Defined	Undefined
CFG_LUT_PATCH	Performs the calibration of the Voltage Controlled Oscillator of the radio PLL. It must **not** be altered by the customer.	Calibration disabled
CFG_WDOG	Watchdog timer enabled	Watchdog timer disabled
CFG_EXT_SLEEP CFG_DEEP_SLEEP	Default sleep mode. Only one must be defined	
BLE_CONNECTION_MAX_USER	Max connections number (1-6, 8 for DA14581)	
DEVELOPMENT_DEBUG	Project in development + debug information is enabled. DEEP sleep cannot be allowed.	It must be set to 0 before the production if the system boots from OTP or Flash. DEEP sleep can be allowed.
APP_BOOT_FROM_OTP	Defines if the application starts from OTP. OTP memory and OTP header are copied into SRAM during the boot-loader's OTP copy process.	Application is downloaded to System RAM from a communication interface or Debugger. OTP header is not copied into SRAM and the application accesses it from OTP.
READ_NVDS_STRUCT_FROM_OTP	If defined the NVDS structure is padded with zeros and it must be written in OTP otherwise it contains the hardcoded vaues	
CFG_LP_CLK	Low power clock selection (XTAL32 or RCX20) (Appendix B)	
USE_POWER_OPTIMIZATIONS	Enable power optimizations	Disable power optimizations
CFG_USE_DEFAULT_ XTAL16M_TRIM_ VALUE_IF_NOT_CALIBRATED	**Use a default trim value calculated for Dialogs DK for XTAL16M if a trim value has not been programmed in OTP.** **Define it only during the development if a trim value has not been programmed in OTP.**	**Must be undefined for final calibrated products and the XTAL calibration value to be programmed in OTP memory**
MEM_LEAK_PATCHED_ENABLED	Includes the software patch of a memory leak issue of BLE stack software.	Excludes the software patch
REINIT_DESCRIPT_BUF	Memory Map/Scatter File configuration. More information is given in [14].	
USE_MEMORY_MAP		
DB_HEAP_SZ		
ENV_HEAP_SZ		
MSG_HEAP_SZ		
NON_RET_HEAP_SZ		

Projects in the DA14580/581/583 SDK use two additional configuration header files:

- *da14580_scatter_config.h*: Scatter file and memory map configuration. More information is given in [14].
- *da14580_stack_config.h*: BLE stack and kernel definitions.

However, these files must **not** be altered by the customer.

Additional configurable parameters of the stack are set in the following files:

- *dk_apps\src\ip\ble\hl\src\rwble_hl\rwble_hl_config.h*
- *dk_apps\src\modules\rwip\api\rwip_config.h*

6.4 Integrated processor mode API

The proximity reporter sample application, which is implemented in the dk_apps\keil_projects\ proximity\Keil_4\prox_reporter or dk_apps\keil_projects\ proximity\Keil_5\prox_reporter, will be used as a reference to describe the software API for the integrated processor applications. Please refer to the user manuals [5] and [17] for more information on how to open and execute this project.

6.4.1 Application to kernel API

The RivieraWaves Kernel is fully described in [7]. It is a small and efficient Real Time Operating System, offering the following features:

- Exchange of messages
- Message saving
- Timer functionality
- Event functionality (used to defer actions)

In order to use the services offered by the kernel the user should include the following files:

- *ke_task.h*
- *ke_timer.h*

Adding an application task

In the header file *dk_apps\src\modules\rwip\api\rwip_config.h* the `KE_TASK_TYPE` enumeration is defined, which contains all the kernel tasks. In file *dk_apps\src\modules\app\src\app.c*, the application task descriptor is defined:

```
// Application Task Descriptor
static const struct ke_task_desc TASK_DESC_APP = {NULL, &app_default_handler,
app_state, APP_STATE_MAX, APP_IDX_MAX};
```

Note that the task descriptor `TASK_DESC_APP` is of type struct ke_task_desc:

```
/// Task descriptor grouping all information required by the kernel for the
scheduling.
struct ke_task_desc
{
    /// Pointer to the state handler table (one element for each state).
    const struct ke_state_handler* state_handler;
    /// Pointer to the default state handler (element parsed after the current
state).
    const struct ke_state_handler* default_handler;
    /// Pointer to the state table (one element for each instance).
    ke_state_t* state;
    /// Maximum number of states in the task.
    uint16_t state_max;
    /// Maximum index of supported instances of the task.
    uint16_t idx_max;
};
```

The application developer needs to define the state handler table for each state (`NULL`), the default handler `app_default_handler()`, provide a place holder for the states of all the task instances (`app_state`), specify the maximum task states (`APP_STATE_MAX`) and the maximum stack instances (`APP_IDX_MAX`) in accordance with the task descriptor structure. In the application examples, this is done in the files *app_task.c* and *app_task.h*.

Creating an application environment

An environment is needed to store some important data for the application; like the connection handle and security flag. The structure of this environment is defined in the file *app.h*:

```
/// Application environment structure
struct app_env_tag
{
    /// Connection handle
    uint16_t conhdl;
    uint8_t  conidx; // Should be used only with KE_BUILD_ID()
    /// Last initialised profile
    uint8_t next_prf_init;
    /// Security enable
    bool sec_en;
    // Last paired peer address type
    uint8_t peer_addr_type;
    // Last paired peer address
    struct bd_addr peer_addr;
    #if BLE_HID_DEVICE
       uint8_t app_state;
       uint8_t app_flags;
    #endif
};
```

The application environment is defined in *app.c*:

```
struct app_env_tag app_env;
```

System startup

Although the system's main function is not part of the application API, it is important to understand the system startup process so that the software flow can be followed.

The `main()` function of the sample application is the `int main_func(void)`. After the system boots up, the `main()` function, which is stored in ROM, will call the function:

```
PtrFunc = (my_function)(jump_table_struct[main_pos]);
```

which is translated to the RAM function:

```
int main_func(void)
```

The source code of this function can be found in the file *dk_apps\src\plf\refip\src\arch\main\ble\arch_main.c*.

At the beginning of this function the DA14580/581 platform initialisation takes place, followed by BLE stack initialisation. Next, when the code is compiled for integrated processor configuration, the application is initialised:

```
#if (BLE_APP_PRESENT)
    {
        app_init();         // Initialise APP
    }
#endif /* #if (BLE_APP_PRESENT) */
```

and finally, the main `while(1)` is entered. In this while loop, the BLE scheduler is called to schedule all pending BLE events:

```
rwip_schedule()
```

and then a decision is made which sleep mode is entered by reading the sleep mode (defined by the enumeration `sleep_mode_t`) and executing the relevant code:

```
sleep_mode = rwip_sleep();
```

Finally, the `WFI()` function is called at the end of the while loop which suspends the execution until an event occurs.

6.4.2 Application initialisation

As described in the previous section, the main function calls the `app_init()` function to initialise the application. The following initialisations are required:

- Initialise the list of the profiles that the application requires. In the Proximity Reporter project, the profiles needed are defined in *app_api.h* in an enumerator with first value `APP_PRF_LIST_START`. The task names of the profiles are listed in this enumerator:
  - APP_DIS_TASK // Device Information Service profile
  - APP_PROXR_TASK // Proximity Reporter profile
  - APP_BASS_TASK // Battery Server profile
- The application task needs to be created and to be initialised:
  - `ke_task_create(TASK_APP, &TASK_DESC_APP);`
- The security task is initialised when `CFG_APP_SEC` is enabled.

6.4.3 Application to GAP API

The RW-BLE Generic Access Profile (GAP) defines the basic procedures related to the discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices. Furthermore, it defines procedures related to the use of different LE security modes and levels. For a detailed description of the API refer to [8].

Adding GAP event handlers

As described in the previous section, the BLE stack initialisation takes place in `main_func()`. When the GAP entity has been initialised and is ready to provide services to the upper layers, the event `GAPM_DEVICE_READY_IND` is sent to the upper layers. Since the application task has defined a handler for this event, the kernel scheduler will call the function `gapm_device_ready_ind_handler()`.

In the example application code the default state handlers definition is in the *app_task_handlers.h* file as shown below:

```
EXTERN const struct ke_msg_handler app_default_state[] = {
    {GAPM_DEVICE_READY_IND,(ke_msg_func_t)gapm_device_ready_ind_handler},
    {GAPM_CMP_EVT,(ke_msg_func_t)gapm_cmp_evt_handler},
    {GAPC_CMP_EVT,(ke_msg_func_t)gapc_cmp_evt_handler},
    {GAPC_CONNECTION_REQ_IND,(ke_msg_func_t)gapc_connection_req_ind_handler},
    {GAPC_DISCONNECT_IND,(ke_msg_func_t)gapc_disconnect_ind_handler},
    {APP_MODULE_INIT_CMP_EVT,(ke_msg_func_t)app_module_init_cmp_evt_handler},
```

In the above definition, handlers are also defined for those GAP events that the application needs to be aware of. In a similar way, the application developer can add more GAP event handlers for any of the GAP events in the state, that the application needs to act upon.

Note: The GAP module consists of two tasks:

- GAP Manager (TASK_GAPM)
- GAP Controller (TASK_GAPC)

GAP setup

The first action of the proximity reporter application after receiving the `GAPM_DEVICE_READY_IND` message, is to send the `GAPM_RESET_CMD` command to TASK_GAPM. The GAPM will respond with `GAPM_CMP_EVT` and the handler `gapm_cmp_evt_handler()` will be called, resulting in sending the command `GAPM_SET_DEV_CONFIG_CMD` to TASK_GAPM. This will cause GAPM to respond with `GAPM_CMP_EVT`, indicating that the previous command has been completed and that the initialisation of TASK_GAPM has been completed.

After GAPM initialisation and when the `GAPM_CMP_EVT` message has been received in TASK_APP, the function `app_db_init()` is called to initialise the profile database. **Note:** This function will be called for every profile in the list described in section 6.4.2.

After each database has been initialised, the profile task will send the `XXX_CREATE_DB_CFM` message (where XXX is the name of the profile) to the application. Then the `xxx_create_db_cfm_handler()` will be called and send the `APP_MODULE_INIT_CMP_EVT` message from TASK_APP to TASK_APP. The handler `app_module_init_cmp_evt_handler()` will be called and the function `app_db_init()` will be called for the next profile, if any. Otherwise `app_adv_start()` will be called to start the advertising procedure.

Advertising data

The advertising data are defined in the file *app_proxr_proj.h*. In the proximity reporter application code, the default advertising data are defined as follows:

```
#define APP_ADV_DATA         "\x07\x03\x03\x18\x02\x18\x04\x18"
#define APP_ADV_DATA_LEN     (8)
```

This means (data decoding as per [5]):

```
x07           Length
x03           Complete list of 16-bit UUIDs available
x03\x18       Link Loss Service UUID
x02\x18       Immediate Alert Service UUID
x04\x18       Tx Power Service UUID
```

Advertising procedure

In the function `app_adv_start()`, the application function `app_adv_func()` is called to send the `GAPM_START_ADVERTISE_CMD` message to GAPM. An example is given below how this message should be filled:

```
cmd->op.code         = GAPM_ADV_UNDIRECT;
cmd->op.addr_src     = GAPM_PUBLIC_ADDR;
cmd->intv_min        = APP_ADV_INT_MIN;
cmd->intv_max        = APP_ADV_INT_MAX;
cmd->channel_map     = APP_ADV_CHMAP;
cmd->info.host.mode = GAP_GEN_DISCOVERABLE;
```

The advertising data are also copied into the message. **Note:** The parameter

```
cmd->info.host.adv_data_len
```

has to specify the length of the advertising data exactly. GAPM will check the size and when it does not match the size of the advertising data, the message will be ignored.

The application can stop the advertising procedure by calling the function `app_adv_stop()`.

Device connected

After advertising has started and the device enters the connected state, GAPC will send the `GAPC_CONNECTION_REQ_IND` message and the handler `gapc_connection_req_ind_handler()` is called, which calls the application API function `app_connection_func ()`.

The application will confirm the connection indication message to GAPC by sending the `GAPC_CONNECTION_CFM` message. If security is required, the function `app_security_enable()` is called to set up the security mode and pass the security parameters to GAPC.

At this point, the device is connected and the application can use the profile services.

6.4.4 Application to profile API

The proximity reporter profile API is documented in [9] and [10]. Refer to header file *proxr_task.h* for the implementation of this API.

6.5 External processor API

As described in section 5, in an external processor configuration the link layer, host protocols and profiles run on the DA14580/581 (embedded), the application runs in a separate CPU (host application) and these two components communicate via a proprietary HCI [2].

Using the proximity monitor example code included in the SDK as a reference, the two components mentioned above are implemented in the following projects:

- **Host application**:
  dk_apps\keil_projects\proximity\monitor\host_proxm_sdk\host_proxm_sdk.vcxproj
- **DA14580/581/583 project**:
  Keil uVision4
  dk_apps\keil_projects\proximity\prox_reporter_ext\Keil_4\prox_reporter_ext.uvproj
  dk_apps\keil_projects\proximity\prox_reporter_ext\ Keil_4\prox_reporter_ext_581.uvproj
  dk_apps\keil_projects\proximity\prox_reporter_ext\ Keil_4\prox_reporter_ext_583.uvproj
  Keil uVision5
  dk_apps\keil_projects\proximity\prox_reporter_ext\Keil_5\prox_reporter_ext.uvprojx
  dk_apps\keil_projects\proximity\prox_reporter_ext\ Keil_5\prox_reporter_ext_581.uvprojx
  dk_apps\keil_projects\proximity\prox_reporter_ext\ Keil_5\prox_reporter_ext_583.uvprojx

Please refer to [5] and [16] for more information on how to open and execute this project.

6.5.1 Host application to external processor interface

The host application sends commands and receives confirmations, events and indications from the BLE stack and profile tasks. Commands, confirmations events and indications are encapsulated in HCI messages, which have the following structure:

```
typedef struct {
  unsigned short bType;// Command, confirmation,event,indication type
  unsigned short bDstid; // Destination Task Id. should be == TASK_APP
  unsigned short bSrcid; // Source Task Id.
  unsigned short bLength; //Paylod Data size
  unsigned char  bData[1]; //Message's data. Format depends to message type.
} ble_msg;
```

Initialisation

The host application expects to receive a `GAPM_DEVICE_READY_IND` message upon startup of the DA14580/581/583 device. The host application then sends a `GAPM_RESET_CMD` command to GAPM. The message flow is the same as described in section 6.4.3 for GAP setup.

Discovering devices

After GAPM has been set up and the `GAPM_SET_DEV_CONFIG` has been received by the host application, it can then send a `GAPM_START_SCAN_CMD` command to start scanning for devices within range. When the DA14580/581/583 discovers a device, it sends the event `GAPM_ADV_REPORT_IND` with the details of the discovered device.

Connecting

The host application must send a `GAPM_START_CONNECTION_CMD` message for the selected Bluetooth device address *(*`bdaddr`*)*. It will be notified of the successful completion or failure of the connection via a `GAPC_CONNECTION_REQ_IND` message.

Appendix A Non-Volatile Data Storage

The Non-Volatile Data Storage (NVDS) can used to keep system configuration settings such as Bluetooth device address, device name, advertise data, scan response data, etc.

```
struct nvds_data_struct {
	uint32_t	NVDS_VALIDATION_FLAG; // define which fields are valid
	uint32_t	NVDS_TAG_UART_BAUDRATE;// UART baudrate
	uint32_t	NVDS_TAG_DIAG_SW;	// Diagport configuration
	uint32_t	NVDS_TAG_DIAG_BLE_HW; // Diagport configuration
	uint16_t	NVDS_TAG_NEB_ID;	// Neb Session ID
	uint16_t	NVDS_TAG_LPCLK_DRIFT; // Low power clock accourancy
	uint8_t	NVDS_TAG_SLEEP_ENABLE;	// Enable sleep mode
	uint8_t	NVDS_TAG_EXT_WAKEUP_ENABLE; //External wakeup enable
	uint8_t	NVDS_TAG_SECURITY_ENABLE; //Enable security for BLE application
	uint8_t	ADV_DATA_TAG_LEN;	// Advertise data size
	uint8_t	SCAN_RESP_DATA_TAG_LEN; // Scan response data size
	uint8_t	DEVICE_NAME_TAG_LEN;	// Device name size
	uint8_t	NVDS_TAG_APP_BLE_ADV_DATA[32];	// Advertise data
	uint8_t	NVDS_TAG_APP_BLE_SCAN_RESP_DATA[32]; // Scan response data
	uint8_t	NVDS_TAG_DEVICE_NAME[62]; // Device name
	uint8_t	NVDS_TAG_BD_ADDRESS[6]; // Device Bluetooth address
	uint16_t	NVDS_TAG_BLE_CA_TIMER_DUR; // Default Channel Assessment Timer
	duration
	uint8_t	NVDS_TAG_BLE_CRA_TIMER_DUR; // Default Channel Reassessment Timer
	duration
	uint8_t	NVDS_TAG_BLE_CA_MIN_RSSI;// Default Minimal RSSI Threshold
	uint8_t	NVDS_TAG_BLE_CA_NB_PKT; // Default number of packets to receive
	for statistics
	uint8_t	NVDS_TAG_BLE_CA_NB_BAD_PKT;// Default number of bad packets
	needed to remove a channel
};
```

It is mapped to a constant system RAM position (0x20000340 when the system RAM is mapped to 0x20000000) as shown in the map file of an application Keil project, which corresponds to offset 0x340 in the OTP memory.

```
	nvds_data_storage			0x20000340 in DA14581 project
	nvds_data_storage			0x20000350 in DA14580/583 projects
```

The compilation option `READ_NVDS_STRUCT_FROM_OTP` can be used to define whether the NVDS will be read from OTP or will be initialised with hardcoded values by the application software.

The developer can use the OTP NVDS tool of the Smart Snippets toolkit to write the NVDS structure into OTP memory. The data written in the NVDS area of the OTP memory are copied to the corresponding system RAM position (0x20000340) during the OTP mirroring process [1].

An alternative way for configuring the Bluetooth Device address (BD address) is offered through the OTP header, which has priority over the NVDS. The device address can be written to offset 0x7FD4 of the OTP memory using the OTP header tool of Smart Snippets. The software reads the BD address field of the OTP header (function `nvds_read_bdaddr_from_otp()` in *nvds.c*), and when it is set (non-zero), copies it to the NVDS BD address field (function `custom_nvds_get_func()` in *nvds.c*).

Appendix B How to select the low power clock

Support of the RCX clock as low power clock source is added in SDK v3.0.2.

A configuration flag is added in projects' *da14580_config.h* for low power clock source selection:

```
#define CFG_LP_CLK   0x00
```

Where:

```
0x00   is used for XTAL32,
0xAA   is used for RCX,
0xFF   the low power clock is read from the corresponding field in OTP Header.
```

A calibration mechanism has been developed to measure the RCX clock frequency changes over temperature. This mechanism consists of the functions `calibrate_rcx20()` and `read_rcx_freq()`. Both functions are implemented in *\ble_sw\dk_apps\src\plf\refip\src\arch\main\ble\arch_system.c*.

When RCX is selected as low power clock, the function `calibrate_rcx20()` initiates the HW process to measure the number of XTAL16 ticks (16 MHz) elapsed during the countdown of a specified number of RCX ticks. RCX evaluation under temperature cycling proved that a calibration process of 20 RCX ticks gives adequate precision in current frequency calculation. Function `calibrate_rcx20()` is called in the sleep interrupt handler to start the HW calibration process, while the processor services the BLE event.

The function `read_rcx_freq()` checks that the calibration process is completed in HW, reads the number of XTAL16 clock ticks and calculates the RCX frequency. Function `read_rcx_freq()` is called at the end of a BLE connection event right before entering sleep mode. The hardware calibration is completed at this point, hence there is no extension of the wakeup period.

Appendix C Preferred RF settings

Preferred radio settings for the DA14580/581/583 are stored in the file *\ble_sw\dk_apps\src\plf\refip\src\arch\system_settings.h*.

The user should not modify this file as the RF performance and compliance to the Bluetooth specification may be violated.

Appendix D Opening your project for the first time

D.1 Keil IDE crashes when clicking on “J-LINK/J-TRACE Cortex” settings

When on a Keil uVision project some entries in file **.uvopt** are missing or the file itself is missing, uVision crashes when the user clicks on the button 'settings' (options{debug tag}) with the{J-LINK/J-TRACE Cortex} selected.

D.1.1 Possible causes

Some important information concerning the j-link driver is missing. Calling the driver’s DLL probably causes the crash.

D.1.2 Affected versions of Keil uVision

At least uVision versions 5.11.1.0 and 5.10.0.2 are affected.

D.1.3 Circumstances of the error

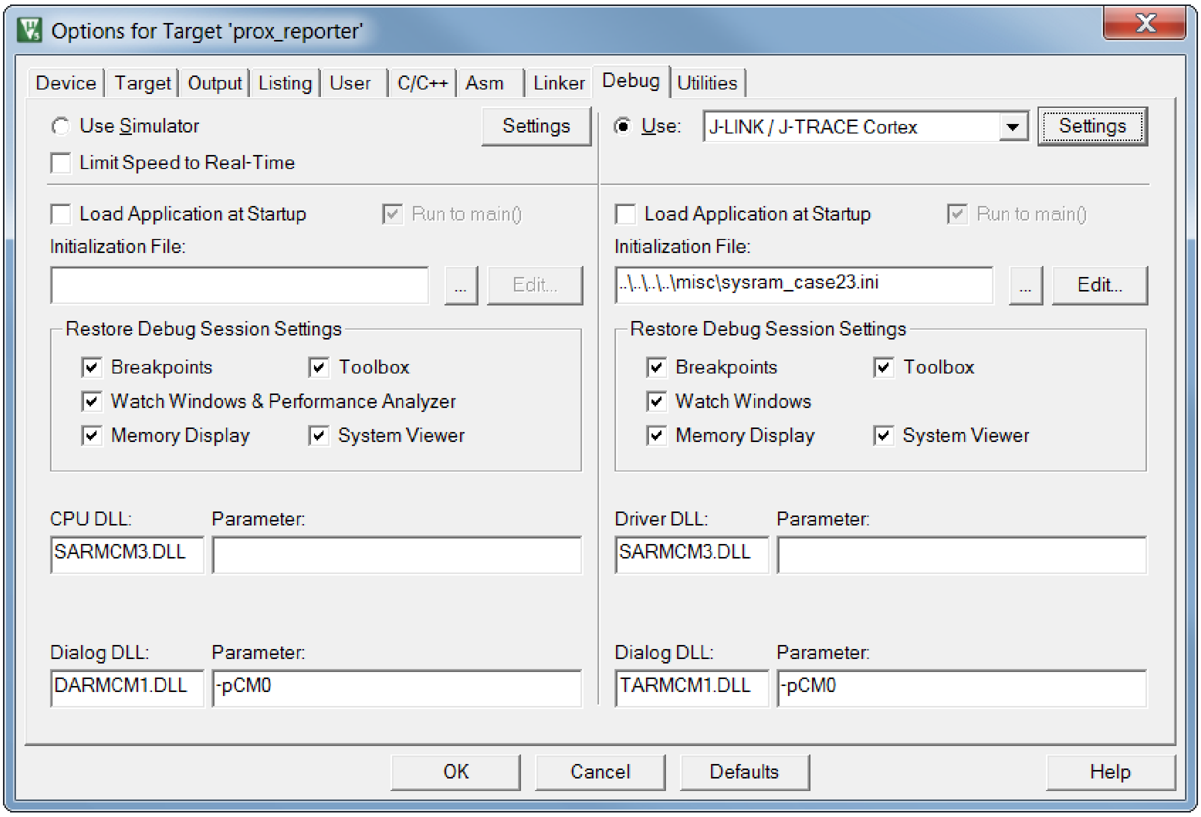
When a local GIT repository is first created, the file **.uvopt** does not exist, since it is not included in the remote repository. When the user opens the project for the first time, this file is created but some keys/values are missing.

D.1.4 Proposed solution

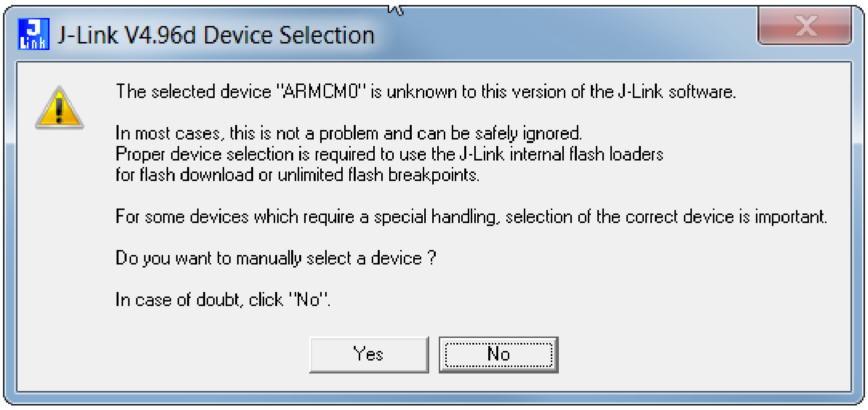
1. Ensure that the .uvopt file does not exist in the folder of your project. If the file exists and a crash has been identified to happen, delete the .uvopt file.
2. Open the Keil project and close it. The .uvopt file is created automatically in the project folder where the .uvproj is located.
3. Open the .uvopt file, using your favourite text editor.
4. Under the key <TargetOption> add the flowing lines:
```
<TargetDriverDllRegistry>
<SetRegEntry>
<Number>0</Number>
<Key>JL2CM3</Key>
<Name>-U228202424 -O78 -S0 -A0 -C0 -JU1 -JI127.0.0.1 -JP0 -RST0 -N00("ARM
CoreSight SW-DP") -D00(0BB11477) -L00(0) -TO18 -TC10000000 -TP21 -TDS8007 -TDT0 -
TDC1F -TIEFFFFFFFF -TIP8 -TB1 -TFE0 -FO7 -FD20000000 -FC800 -FN0</Name>
</SetRegEntry>
</TargetDriverDllRegistry>
```
5. Save the .uvopt file and close the text editor.
6. Open the Keil project in uVision.
7. Click on *Project→Options for Project ‘XXX’.*
8. On the ‘Debug’ Tab, select J-Link / J-TRACE Cortex debugger and click on the ‘Settings’ button for the debugger (not the simulator). **This is the instance where the crash would happen.**
9. The ‘Cortex JLink/JTrace Target Driver Setup’ Dialog opens*. Select your debugger as you would normally do.*
10. Close the dialog windows by clicking OK.
11. *Now, normal operation of j-link debugger is resumed.* After you have finished your work, close the Keil uVision IDE to allow for updates to the .uvopt file to be saved.

D.2 Keil 5 ARMCM0 device is not recognized by J-Link

The issue occurs the first time that the “J-LINK / J-TRACE” settings are accessed in a DA14580/581/583 Keil 5 project, by clicking the “Settings” button as shown below:



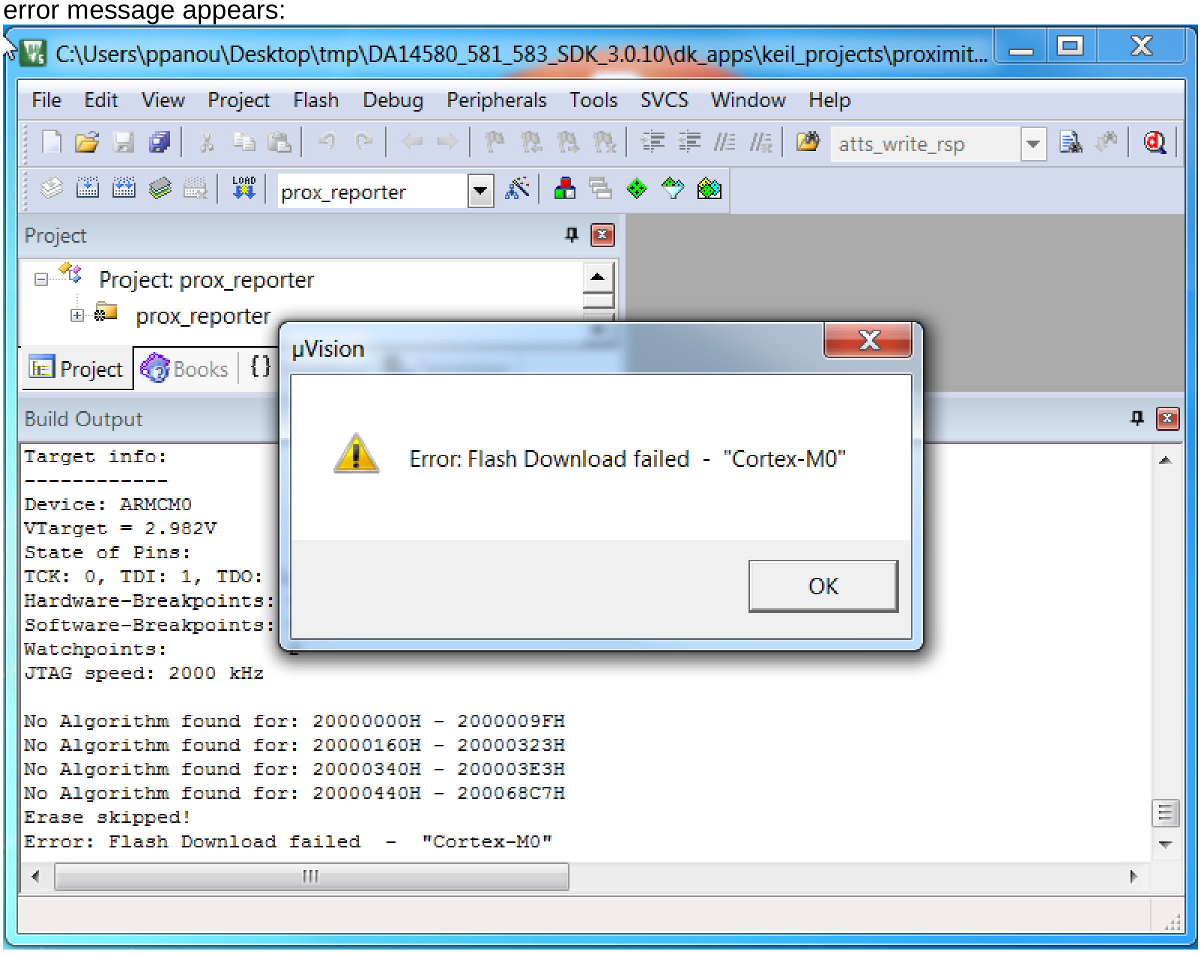
J-Link reports the ARMCM0 device as an unknown device.



The solution is to select “No”.

D.3 Keil 5 IDE reports flash download failure

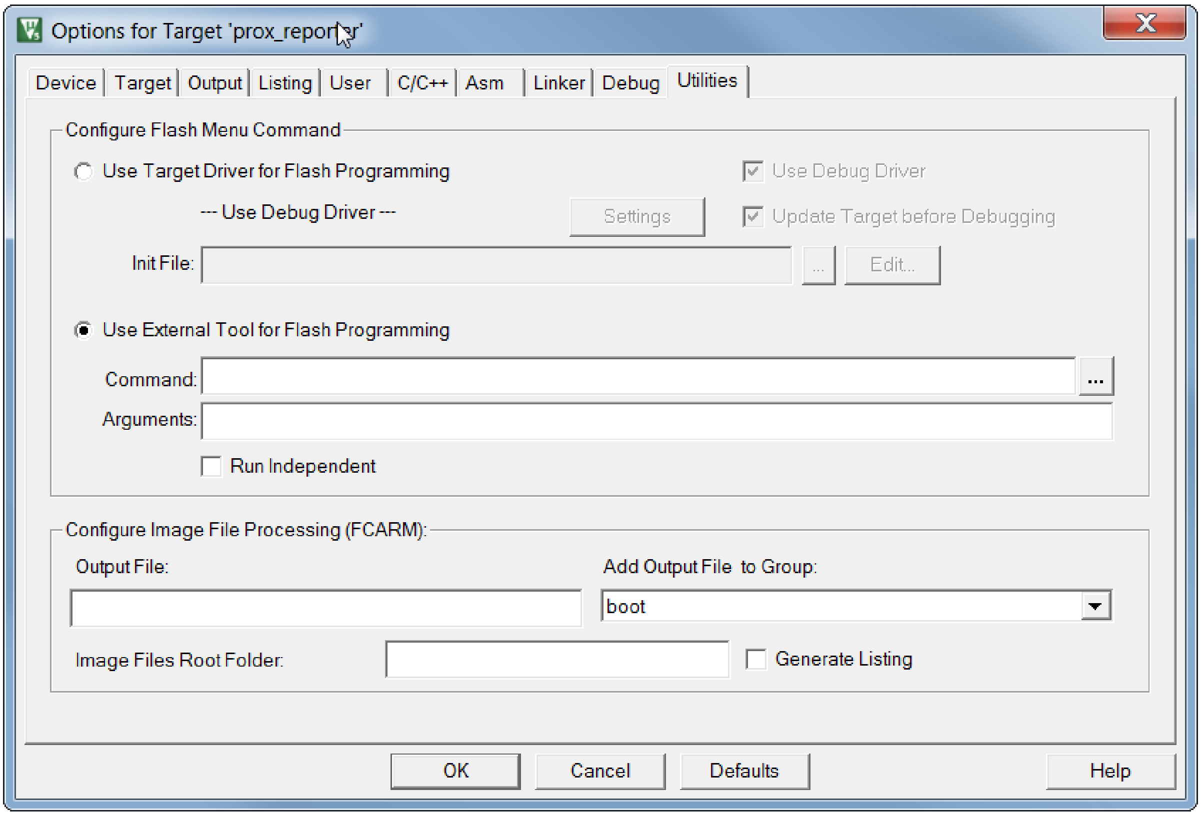
The issue occurs when a new DA14580/581/583 Keil 5 project is created or when the project’s .uvoptx file is deleted. When the developer attempts to start a new debugging session the following error message appears:



The solution is the following:

Open menu Project -> “Options for Target ’xxx’” and go to the “Utilities” tab page.

There select the “Use External Tool for Flash Programming” radio button:



Appendix E True Random Number Generator (TRNG)

This Appendix describes the function `trng_acquire()` which generates a 128-bit random number. The function is implemented in the file *trng.c* in the folder: dk_apps\src\plf\refip\src\driver\trng\.

Function name	`void` **`trng_acquire`** `(uint8_t *trng_bits_ptr)`
Function description	Acquires 128-bit random data from the Radio
Parameters	`trng_bits_ptr` stores the 128-bit number
Return values	None
Notes	

The function `trng_acquire()`:

- Initialises the system and radio, sets preferred settings and performs radio calibration: `rfpt_init();`
- Implements Save-Modify-Restore for the preferred settings that will be changed in TRNG mode:

```
save_TEST_CTRL_REG=GetWord16(TEST_CTRL_REG);
save_RF_ENABLE_CONFIG1_REG=GetWord16(RF_ENABLE_CONFIG1_REG);   // LNA off
save_RF_ENABLE_CONFIG2_REG=GetWord16(RF_ENABLE_CONFIG2_REG);   // Mixer off
save_RF_DC_OFFSET_CTRL1_RE=GetWord16(RF_DC_OFFSET_CTRL1_REG);  // Fixed DC offset
    compensation values for I and Q
save_RF_DC_OFFSET_CTRL2_REG=GetWord16(RF_DC_OFFSET_CTRL2_REG); // Use the manual
    DC offset compensation values
save_RF_ENABLE_CONFIG4_REG=GetWord16(RF_ENABLE_CONFIG4_REG);   // VCO_LDO_EN=0,
    MD_LDO_EN=0. You need this for more isolation from the RF input
save_RF_ENABLE_CONFIG14_REG=GetWord16(RF_ENABLE_CONFIG14_REG); // LOBUF_RXIQ_EN=0,
    DIV2_EN=0. This increases the noise floor for some reason. So you get more
    entropy. Need to understand it and then decide...
```



```
save_RF_SPARE1_REG=GetWord16(RF_SPARE1_REG);  // Set the IFF in REAL transfer
    function, to remove I-Q correlation. But it affects the DC offsets!
save_RF_AGC_CTRL2_REG=GetWord16(RF_AGC_CTRL2_REG);  // AGC=0 i.e. max RX gain
```

- Configures the radio for TRNG mode (modifies some preferred settings, starts RX in overrule): `trng_init();`
- Starts acquiring raw IQ RFADC data and then extracts the random bits:

```
for (i_acq=0; i_acq < 128/(NUM_POINTS*2/16); i_acq++)
{
  trng_get_raw_data((uint32)&rfadc_data[0], NUM_POINTS/2-1); // acquires the raw
     RFADC IQ samples
  bit_cnt=0;
  rnd_byte=0;
  for (i=0;i<=NUM_POINTS_MUL_2_M_4;i=i+16)
  {
        single_rnd_bit = (vq_uint8[i] & 0x01) ^ (vi_uint8[i] & 0x01) ; // This way
            it can pass ALL the NIST tests! This solves a small bias in 1s or 0s
            which appears due to the actual value of the DC offset...
        rnd_byte= rnd_byte | (single_rnd_bit<<bit_cnt++);
        if(bit_cnt==8)
        {
            trng_bits_ptr[byte_idx++] = rnd_byte;
            bit_cnt=0;
            rnd_byte=0;
        }
        #if (USE_WDOG)
          SetWord16(WATCHDOG_REG, 0xC8);// Reset WDOG! 200 * 10.24 ms active time
            for normal mode!
        #endif
  }
```

}Restores the modified registers

```
SetWord16(TEST_CTRL_REG, save_TEST_CTRL_REG);
SetWord16(RF_OVERRULE_REG,0x0);
SetWord16(RF_ENABLE_CONFIG1_REG,save_RF_ENABLE_CONFIG1_REG);   // LNA off
SetWord16(RF_ENABLE_CONFIG2_REG,save_RF_ENABLE_CONFIG2_REG);   // Mixer off
SetWord16(RF_DC_OFFSET_CTRL1_REG,save_RF_DC_OFFSET_CTRL1_REG); // Fixed DC offset
    compensation values for I and Q
SetWord16(RF_DC_OFFSET_CTRL2_REG,save_RF_DC_OFFSET_CTRL2_REG); // Use the manual
    DC offset compensation values
SetWord16(RF_ENABLE_CONFIG4_REG,save_RF_ENABLE_CONFIG4_REG);   // VCO_LDO_EN=0,
    MD_LDO_EN=0. You need this for more isolation from the RF input
SetWord16(RF_ENABLE_CONFIG14_REG,save_RF_ENABLE_CONFIG14_REG); // LOBUF_RXIQ_EN=0,
    DIV2_EN=0. This increases the noise floor for some reason. So you get more
    entropy. Need to understand it and then decide...
SetWord16(RF_SPARE1_REG, save_RF_SPARE1_REG);  // Set the IFF in REAL transfer
    function, to remove I-Q correlation. But it affects the DC offsets!
SetWord16(RF_AGC_CTRL2_REG, save_RF_AGC_CTRL2_REG);  // AGC=0 i.e. max RX gain
SetBits16 (CLK_AMBA_REG, OTP_ENABLE, 0);             // disables the OTP
```

The function `trng_acquire()` needs 1.3 ms to generate the 128-bits random number.

Appendix F DCDC_VBAT3V API

The following function has been added in SDK 3.0.8 or later for setting the nominal VBAT3V output voltage of the boost converter.

Function name	`void` **`syscntl_set_dcdc_vbat3v_level`** `(enum SYSCNTL_DCDC_VBAT3V_LEVEL level)`
Function description	Sets the nominal VBAT3V output voltage of the boost converter
Parameters	`level` DCDC VBAT3V output voltage
Return values	none

Notes	`enum SYSCNTL_DCDC_VBAT3V_LEVEL`
	`{`
	`    SYSCNTL_DCDC_VBAT3V_LEVEL_2V4  = 4,  // 2.4 V`
	`    SYSCNTL_DCDC_VBAT3V_LEVEL_2V5  = 5,  // 2.5 V`
	`    SYSCNTL_DCDC_VBAT3V_LEVEL_2V62 = 6,  // 2.62 V`
	`    SYSCNTL_DCDC_VBAT3V_LEVEL_2V76 = 7,  // 2.76 V`
	`}`

Appendix G Near Field API

The following functions have been added in SDK 3.0.8 for enabling and disabling Near Field mode (output power -20 dBm).

Function name	`void` **`rf_nfm_enable`** `(void)`
Function description	Enables Near Field mode for all connections.
Parameters	none
Return values	none
Notes	

Function name	`void` **`rf_nfm_disable`** `(void)`
Function description	Disables Near Field mode for all connections.
Parameters	none
Return values	none
Notes	

Appendix H Crypto API

The files for the Crypto API are stored in the folder: dk_apps\src\modules\crypto.

File	Description
aes.c, aes.h	Initialisation functions, API functions
aes_api.c, aes_api.h	Functions for accessing DA14580/581 registers (Native API) `aes_set_key()`, `aes_enc_dec()`
aes_task.c, aes_task.h	TASK_AES related functions
sw_aes.c, sw_aes.h, os_int.h , os_port.h	Software implementation of the AES

Flag `USE_AES` must be defined in the file *da14580_config.h* for the Crypto API to be included in a BLE application.

Function name	`void` **`aes_init`**`(bool reset, void (*aes_done_cb)(uint8_t status))`
Function description	Initiates the AES operation
Parameters	`reset` `FALSE:` create the task, `TRUE:` reset the environment. `aes_done_cb` The callback function to be called at the end of each operation.
Return values	none
Notes	This function will create the task when called with `reset = FALSE` or just setup the environment when called with `reset = TRUE`. It will also set the callback function to be called when triggered by an AES_USE_ENC_BLOCK_CMD message.

Function name	`int` **`aes_operation`**`(unsigned char *key, int key_len, unsigned char *in, int in_len, unsigned char *out, int out_len, int enc_dec, void (*aes_done_cb)(uint8_t status), unsigned char ble_flags)`
Function description	Starts an AES encrypting/decrypting operation
Parameters	`key` The key data. `key_len` The key data length in bytes. Should be 16. `in` The input data block. `in_len` The input data block length. `out` The output data block. `out_len` The output data block length. `enc_dec` 0: decrypt, 1: encrypt `aes_done_cb` The callback function to be called at the end of each operation. `ble_flags` Specifies whether the encryption/decryption will be performed synchronously or asynchronously (message based). Also specifies, when `ble_safe` is specified, whether function `rwip_schedule()` will be called to avoid losing any BLE events.
Return values	0 successful -1 userKey or key is NULL -2 AES task is busy -3 `enc_dec` is not 0/1 -4 `key_len` is not 16
Notes	

The following function can be used as an example of the above crypto API functions. It must be called in *arch_main.c*:

```
#if (BLE_APP_PRESENT)
        app_init();         // Initialize APP
#endif /* #if (BLE_APP_PRESENT) */
#if (USE_AES)
        aes_test();
#endif

unsigned char key[16]={
  0x06,0xa9,0x21,0x40,0x36,0xb8,0xa1,0x5b,0x51,0x2e,0x03,0xd5,0x34,0x12,0x00,0x06};
unsigned char Plaintext[16]={
  0x53,0x69,0x6e,0x67,0x6c,0x65,0x20,0x62,0x6c,0x6f,0x63,0x6b,0x20,0x6d,0x73,0x67};
unsigned char aes_result[16];

static void aes_done_cb(uint8_t status)
{
        //insert code to read the aes_result[] bytes in reversed order
        while(1);
}

void aes_test(void)
{
memcpy(aes_env.aes_key.iv, IV, 16);
        rwip_schedule();
        aes_init(false, NULL);

aes_operation(key, sizeof(key), Plaintext, sizeof(Plaintext), aes_out,
sizeof(aes_out), 1, NULL, 0);
        rwip_schedule();
        aes_operation(key, sizeof(key), aes_out, 16, aes_result, 16, 0, NULL, 0);
        rwip_schedule();
}
```

The Native Crypto API includes the following functions:

Function name	`int` **`aes_set_key`**`(const unsigned char *userKey, const int bits, AES_KEY *key, int enc_dec)`
Function description	Sets the AES encryption/decryption key
Parameters	`userKey` The key data. `bits` Key number of bits. Should be 128. `key` `AES_KEY` structure pointer. `enc_dec` 0: set decryption key, 1: set encryption key
Return values	0 successful -1 `userKey` or `key` is `NULL` -2 `bits` is not 128
Notes	

Function name	`int aes_enc_dec(unsigned char *in, unsigned char *out, AES_KEY *key, int enc_dec, unsigned char ble_flags)`
Function description	AES encryption/decryption block
Parameters	`in` The data block (16 bytes) out The encrypted/decrypted output of the operation (16 bytes) `key` AES_KEY structure pointer `enc_dec` 0: decrypt, 1: encrypt `ble_flags` Specifies whether the encryption/decryption will be performed synchronously or asynchronously (message based). Also specifies, when `ble_safe` is specified, whether function `rwip_schedule()` will be called to avoid losing any BLE events.
Return values	0 successful -1 SMPM uses the HW block
Notes	

The software implementation for the AES decryption includes the following functions:

Function name	`void AES_set_key(AES_CTX *ctx, const uint8_t *key, const uint8_t *iv, AES_MODE mode)`
Function description	Sets up AES with the key/iv and cipher size
Parameters	`ctx` key info storage `key` key information `iv` IV information `mode` cipher size (128, 256)
Return values	None
Notes	

Function name	`void AES_convert_key(AES_CTX *ctx)`
Function description	Prepares the key for decryption
Parameters	`ctx` key info storage
Return values	None
Notes	

Function name	`void AES_decrypt(const AES_CTX *ctx, uint32_t *data)`
Function description	Decrypts a single block (16 bytes) of data
Parameters	`ctx` key info storage `data` data to be decrypted
Return values	None
Notes	

Function name	`void **AES_cbc_decrypt**(AES_CTX *ctx, const uint8_t *msg, uint8_t *out, int length)`
Function description	Decrypts a byte sequence (block size: 16 bytes) using the AES CBC cipher
Parameters	`ctx` key info `msg` data to be decrypted `out` buffer to save the result `length` size of the `msg`
Return values	None
Notes	

A software implementation of the encryption/decryption based on the **axTLS** open source package (http://axtls.sourceforge.net/index.htm) is used for the encrypted firmware image in the following applications:

- mkimage: Software encryption for `AES-CBC()`
- secondary_bootloader: Software decryption for `AES-CBC()`

Appendix I Coexistence API

The following functions have been added in SDK 3.0.8 or later for enabling the WLAN Coexistence handling.

Function name	`void` **`wlan_coex_init`** `(void)`
Function description	Initialises the `wlan_coex` module and enables it
Parameters	None
Return values	None
Notes	Called once from `arch_main`.

Function name	`void` **`wlan_coex_enable`** `(void)`
Function description	Configures and enables the `wlan_coex` module.
Parameters	None
Return values	none
Notes	Called from `wlan_coex_init` and after each wakeup.

Function name	`void` **`wlan_coex_reservations`** `(void)`
Function description	Reserves `wlan_coex` related GPIOs.
Parameters	none
Return values	none
Notes	Called from `GPIO_reservations`.

Function name	`void` **`wlan_coex_prio_criteria_add`** `(uint16_t type, uint16_t conhdl, uint16_t missed)`
Function description	Adds priority case for a specific connection.

Parameters	`type` event type that has priority.
	Defined types are:
	`#define BLEMPRIO_SCAN 0x01 //active scan`
	`#define BLEMPRIO_ADV 0x02 //advertise`
	`#define BLEMPRIO_CONREQ 0x04 //connection request`
	`#define BLEMPRIO_LLCP 0x10 //control packet`
	`#define BLEMPRIO_DATA 0x20 //data packet`
	`#define BLEMPRIO_MISSED 0x40 //missed events`
	`conhdl` connection handle that the event will belong to
	`missed` number of missed connection events that will trigger the priority (only applicable for `type = BLEMPRIO_MISSED`).

Return values	none
Notes	

Function name	`void` **`wlan_coex_prio_criteria_del`** `(uint16_t type, uint16_t conhdl, uint16_t missed)`
Function description	Delete priority case for a specific connection.
Parameters	`type` event type will be deleted `conhdl` connection handle that the event will belong to `missed` not used
Return values	none
Notes	

The following steps must be followed for adding the **wlan_coex** module in an application:

1. Add dk_apps\src\modules\wlan_coex\wlan_coex.c in Keil project.
2. Add dk_apps\src\modules\wlan_coex in Keil project's include path.
3. Add and customise the following defines in *da14580_config.h*:

```
#define WLAN_COEX_ENABLED
#define WLAN_COEX_BLE_EVENT        7
#define WLAN_COEX_PORT             GPIO_PORT_0
#define WLAN_COEX_PIN              GPIO_PIN_0
#define WLAN_COEX_IRQ              1
#define WLAN_COEX_PORT_2           GPIO_PORT_2
#define WLAN_COEX_PIN_2            GPIO_PIN_6
#define WLAN_COEX_IRQ_2            2
#define WLAN_COEX_PRIO_PORT        GPIO_PORT_0
#define WLAN_COEX_PRIO_PIN         GPIO_PIN_6
#define WLAN_COEX_DEBUG            0
```

4. Include *wlan_coex.h* in *periph_setup.c*.
5. Reserve GPIOs used by the module by calling `wlan_coex_reservations()` from `GPIO_reservations()` in *periph_setup.c*.
6. Initialise GPIOs used by the module by calling `wlan_coex_init()` from `set_pad_functions()` in *periph_setup.c*.

Appendix J Packet Error Rate (PER)

An application that needs Packet Error Rate metrics must:

1. Add the `METRICS` flag in file *da14580_config.h*:
   `#define METRICS`
2. Define the following hook function:

Function name	`void` **`metrics_packet_rx_func`**`(uint8_t packet_error_status)`
Function description	The `metrics_packet_rx_func()` hook function is called for each received packet and it is passed the packet error status as an argument.
Parameters	`error_status` 0: no error in packet, other values: error in packet
Return values	none
Notes	

Revision history

Revision	Date	Description
1.0	02-May-2013	Initial version.
2.0	11-Oct-2013	Update for the SDK ver. 2.0.1
3.0	26-Mar-2014	New template, major changes in terminology, new appendixes added.
4.0	17-Jun-2014	Update for the SDK 3.0.2.1
5.0	14-Apr-2015	Added Appendices D, E, F, G, H, I and J. Updated for SDK 3.0.8 release.
6.0	24-Apr-2015	Update table 1
7.0	05-June-2015	Updated for SDK 3.0.10 release.

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

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

United Kingdom (Headquarters)
Dialog Semiconductor PLC
Phone: +44 1793 757700

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

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

