User manual

DA14580 Software architecture

UM-B-015

Abstract

This document describes the software architecture of the DA14580 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

BLE	Bluetooth Low Energy
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 Datasheet, 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. ARM Cortex-M0, ARM
5. UM-B-014, DA14580 Development Kit, 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 Software development guide, Dialog Semiconductor
11. UM-B-008, DA14580 Production test tool, Dialog Semiconductor
12. UM-B-013, DA14580 External processor interface over SPI, Dialog Semiconductor
13. UM-B-007, DA14580 Software Patching over the Air (SPotA), Dialog Semiconductor
14. UM-B-011, DA14580 Memory file and scatter file, Dialog Semiconductor
15. UM-B-004, DA14580 Peripheral drivers, Dialog Semiconductor
16. UM-B-012, DA14580 Creation of a secondary boot loader, Dialog Semiconductor
17. UM-B-004, DA14580 Peripheral examples, Dialog Semiconductor
18. UM-B-008, DA14580 Sleep mode configuration, Dialog Semiconductor
19. UM-B-010, DA14580 Proximity application, Dialog Semiconductor

3 Introduction

One of the software components of the Dialog DA14580 development kit is the BLE Software Development Kit (SDK). The SDK implements the Bluetooth low energy (BLE) 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.

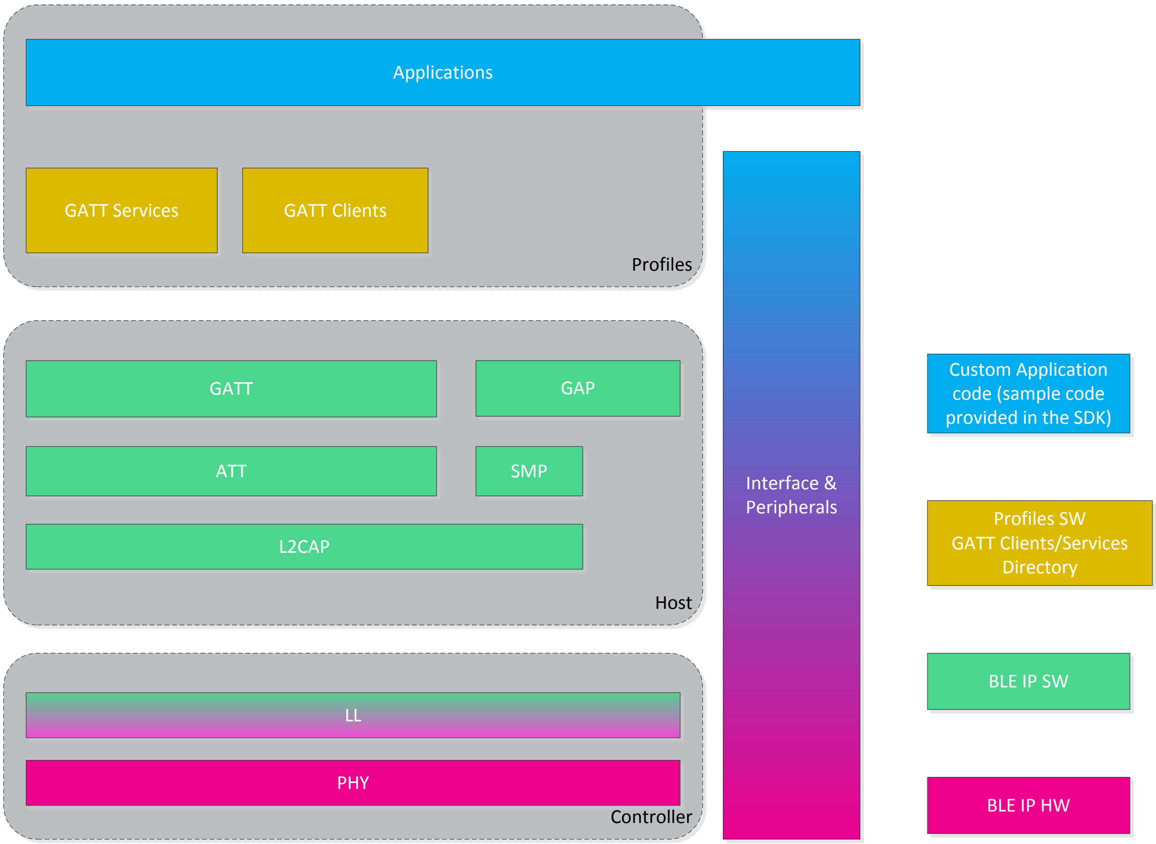

Figure 1: BLE protocol stack

5 BLE software development kit

The DA14580 SDK is a complete software platform for developing single-mode BLE applications. It is based on the DA14580, complete System-on-Chip (SoC) solutions. The DA14580 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 refer to the data sheet [1].

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

Integrated processor configuration: The application and BLE layers (control and host) are implemented in DA14580 chip. It corresponds to Fully_Hosted configuration mode as it's described in Riviera Waves documents. Project names ending with **'_fh'** are based on 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 chip. It corresponds to Fully_Embedded configuration mode as it's described in Riviera Waves documents. Project names ending with **'_fe'** are based on 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 as a single chip solution.

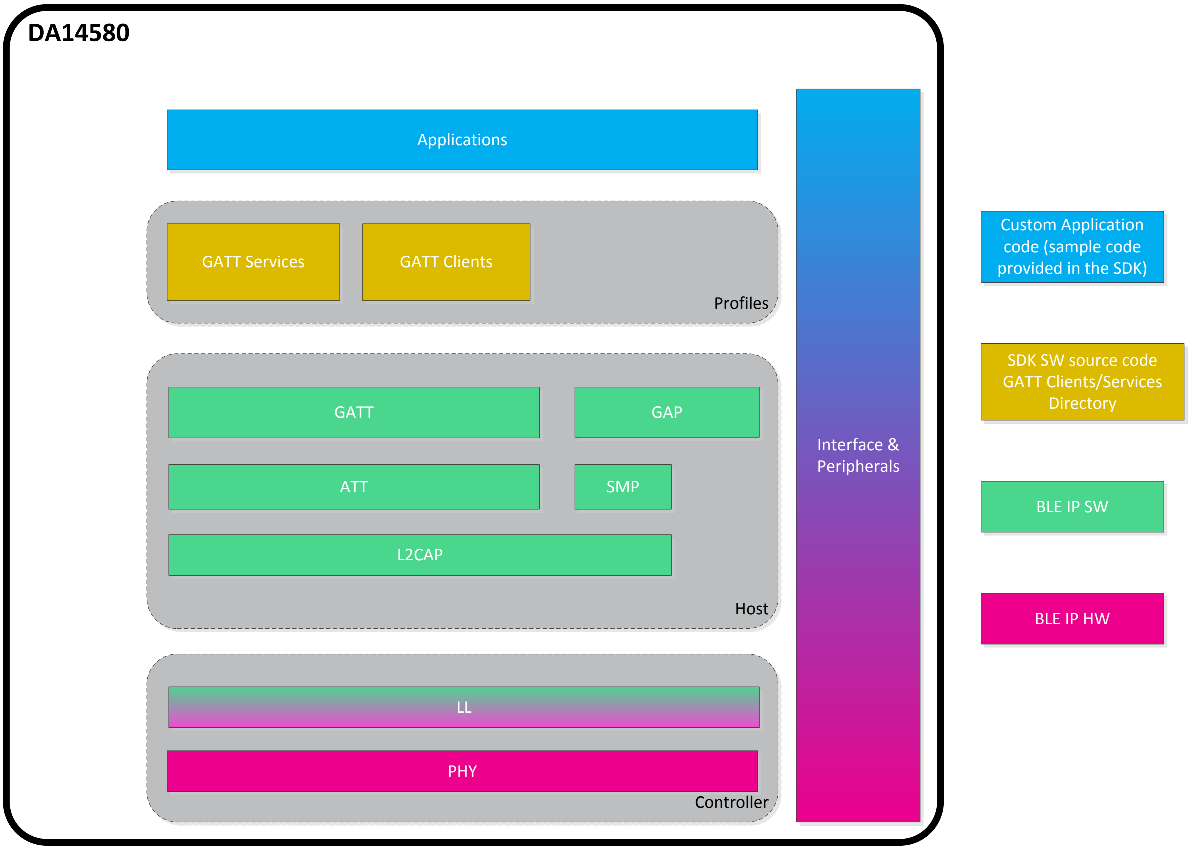

Figure 2: Integrated processor SW configuration

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 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].

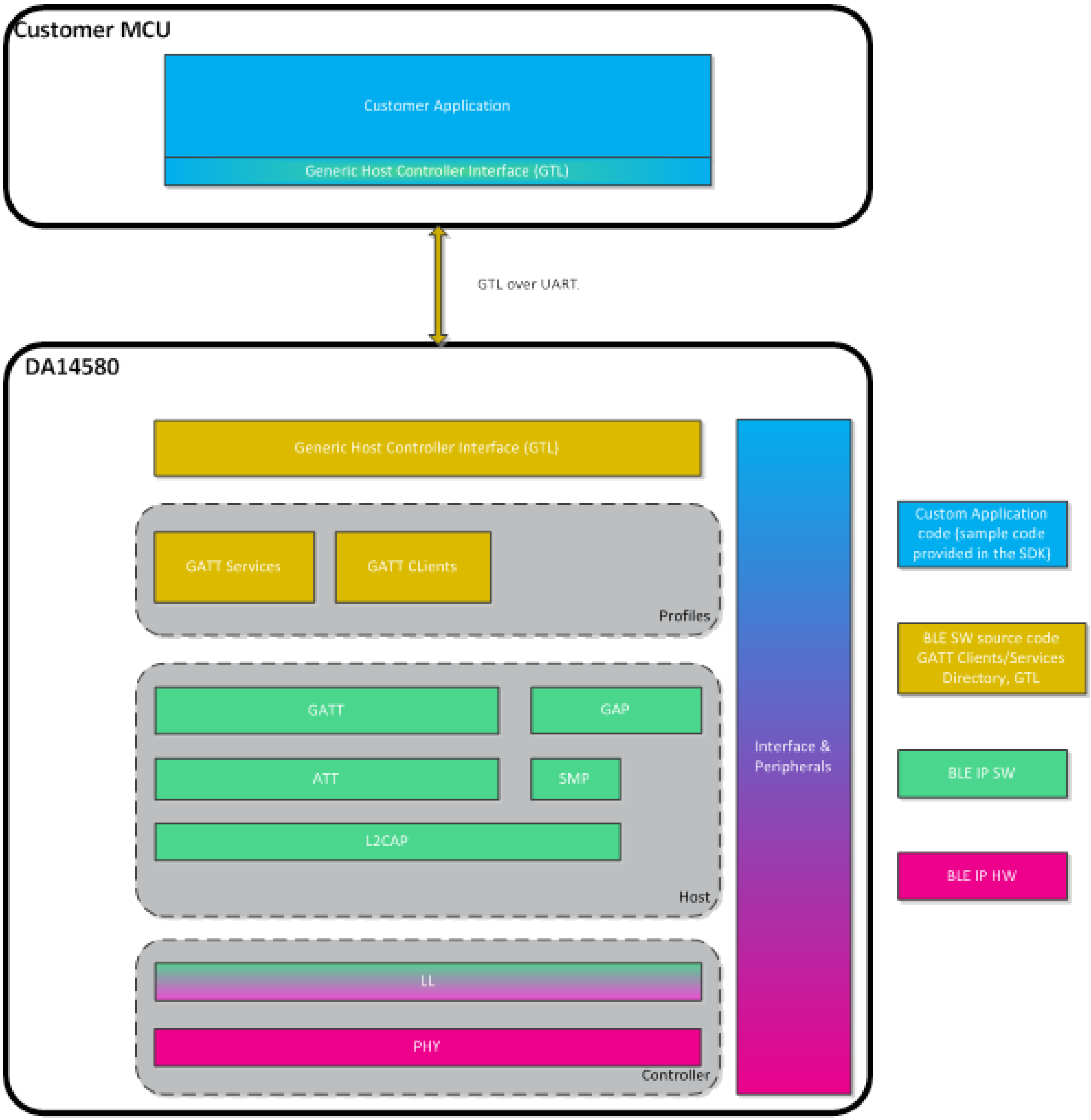

Figure 3: GTL Interface

6 Software structure overview

6.1 ROM/RAM code

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

ROM code

This code resides in the DA14580'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'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 6. These directories are described in the following paragraphs.

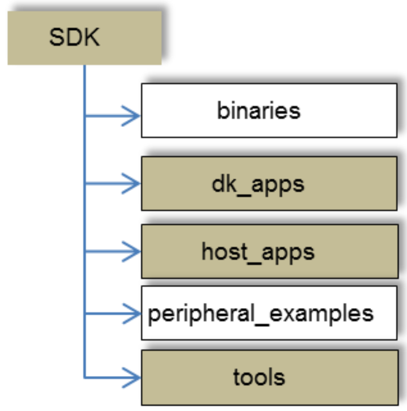

Figure 4: SDK root directory

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 application development. Below follows a short description for each folder.

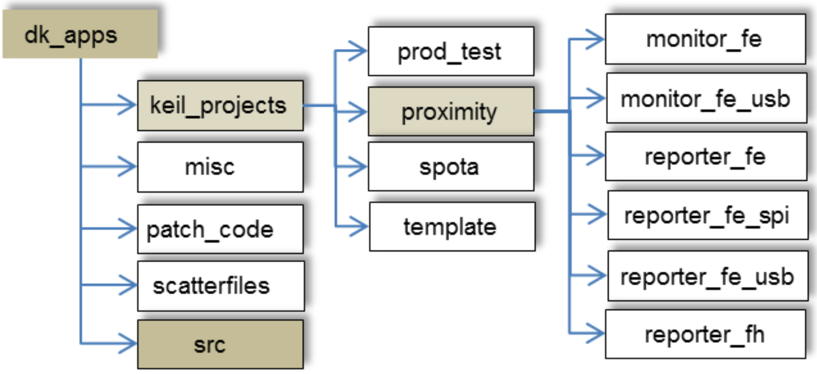

Figure 5: dk_apps directory

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.

- **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.
  - **monitor_fe**: Proximity monitor Keil project. The PC application is stored in host_apps directory.
  - **monitor_fe_usb**: Proximity monitor Keil project for the USB dongle. The configuration settings are the only difference with the monitor_fe project.
  - **reporter_fe**: Proximity reporter Keil project. The PC application is stored in host_apps directory.
  - **reporter_fe_spi**: Proximity reporter Keil project. It supports the external processor over SPI interface. More information is given in [12].
  - **reporter_fe_usb**: Proximity reporter Keil project for the USB dongle. The configuration settings are the only difference with the reporter_fe project.
- **spota**: Keil project for the Software Patching over the Air application. More information is given in [13].
- **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 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].

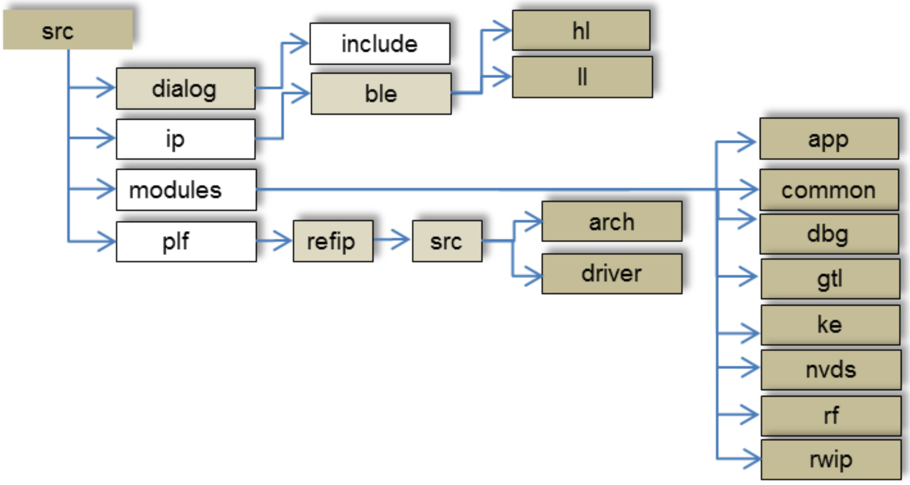

Figure 6: src directory

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.

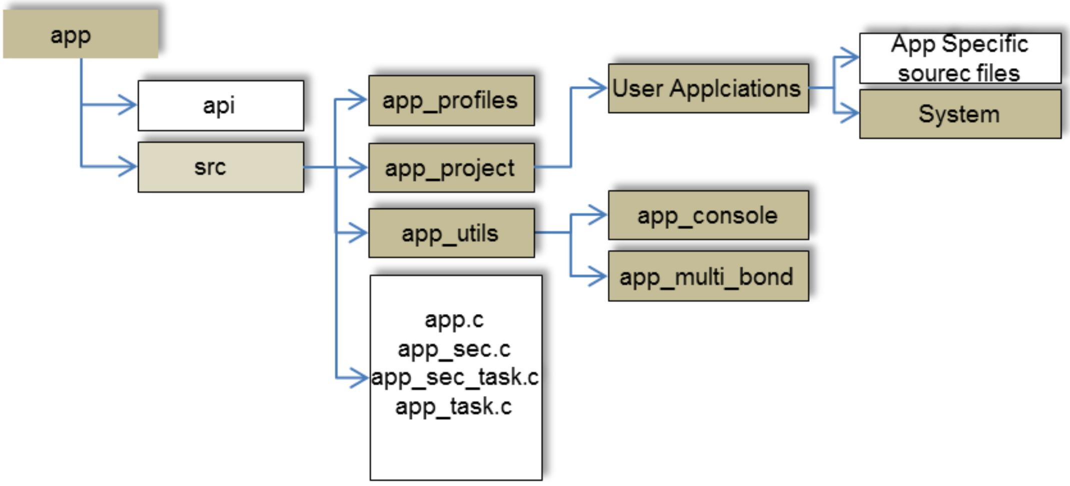

Figure 7: app directory

6.2.3 host_*apps* directory

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

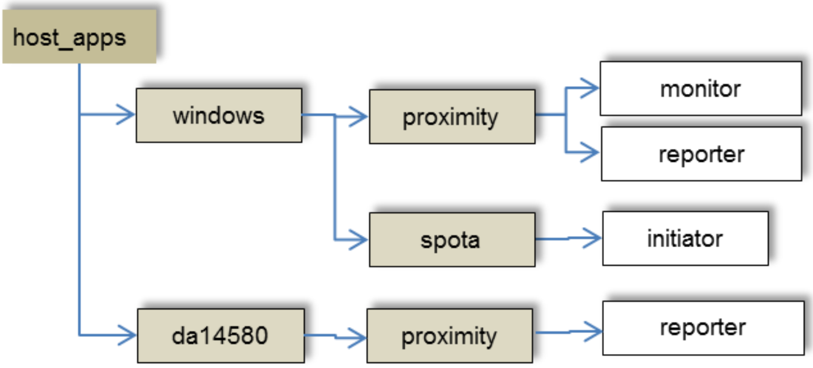

Figure 8: host_apps directory

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 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].

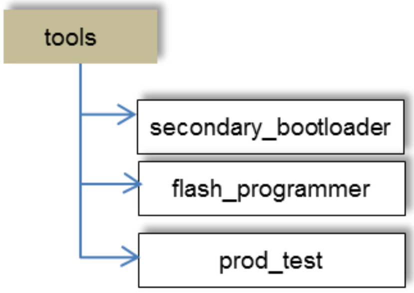

Figure 9: tools directory

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 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)	
DEVELOPMENT__NO_OTP	Development mode, OTP copy at system wakeup is disabled.	In the production, if the product loads the code from OTP.
CFG_LP_CLK	Low power clock selection (XTAL32 or RCX20) (Appendix B)	
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		
CFG_CALIBRATED_AT_FAB	Calibration values written in OTP Header	Un-calibrated device.

Projects in the DA14580 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\reporter_fh will be used as a reference to describe the software API for the integrated processor applications. Please refer to the user manuals [5, 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

The kernel also provides an 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 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 platform initialisation takes place, followed by BLE stack initialisation. Then, if 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() 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 paragraph, 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 list of the profiles needed are defined in *app.h*, (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
  - 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 if `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 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 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 is initialised and 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 and 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 that the GAP module consists of two tasks: the GAP Manager (TASK_GAPM) and the 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 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 has been received in TASK_APP, the app_db_init() is called to initialise the profile database (note that 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 (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 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 (decoding these data 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 bellow 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 in to the message. Note that the parameter

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

has to specify the length of the advertising data exactly, otherwise GAPM will check the size and if it does not match with 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 message GAPC_CONNECTION_REQ_IND and the 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 [9,10]. 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 (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_fe\ fe_proxm_sdk.uvproj
- **DA14580 project**: dk_apps\keil_projects\proximity\monitor_fe\ fe_proxm.uvproj

Please refer to the user guide [5,16] 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 layout:

```
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 at startup expects to receive a `GAPM_DEVICE_READY_IND` upon the DA14580 device startup. The host application 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 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 device Bluetooth address *(bdaddr)*. It will be notified of the completion or failure of the connection with 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 BT 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
```

The developer can use the OTP NVDS tool of the Smart Snippets toolkit to write the NVDS structure in OTP memory. The data written in the NVDS area of the OTP memory are copied to 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 (function nvds_read_bdaddr_from_otp() in *nvds.c*) of the OTP header, 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 corresponding field in OTP Header.

A calibration mechanism has been developed to measure the RCX clock frequency changes over temperature. This mechanism consists of 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*.

If RCX is selected as low power clock calibrate_rcx20() initiates the HW process to measure the number of XTAL16 (16 MHz) ticks elapsed during the countdown of the specified number of RCX ticks. RCX evaluation under temperature cycling proved that a calibration process of 20 RCX ticks provides 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 clocks ticks and calculates current RCX frequency. Function read_rcx_freq() is called at the end of BLE connection event before start entering sleep mode. The hardware calibration is completed at this point, hence there is no extension of wakeup period.

Appendix C Preferred RF settings

Preferred radio settings are stored in the file \ble_sw\dk_apps\src\plf\refip\src\arch\system_settings.h.

User should not change this file as the RF performance and compliance to Bluetooth specification might violated.

7 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 the terminology, new appendixes added.

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

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