User Manual

DA1468x Software Platform Reference

UM-B-044

Abstract

This document should be used as a reference guide to gain a deeper understanding of the SmartSnippets Software Development Kit (SDK). As such it covers a broad range of topics including a brief introduction to Bluetooth Low Energy (BLE), Operating System (OS) related material and a number of sections containing a more detailed technical analysis of hardware elements, for instance clock and power management

Contents

Figures

Tables

Codes

1 Terms and definitions

ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
AHB	AMBA High speed Bus
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
APU	Audio Processing Unit
ATT	Attribute Protocol
BR	Basic Rate
BD	Bluetooth Device
BIN	Binary
BLE	Bluetooth Low Energy
BOD	Brown-Out Detection
CBC	Cipher Block Chaining
CC	Constant Current
CCC	Client Characteristic Configuration
COC	Connection Oriented Channels
CPU	Central Processing Unit
CPM	Clock Power Manager
CRC	Cyclic Redundancy Check
CTR	Counter
CV	Constant Voltage

DCDC	Direct Current – to – Direct Current
DMA	Direct Memory Access
DMIPS	Dhrystone MIPS (Million Instructions Per Second)
ECB	Electronic Codebook
ECC	Elliptic Curve Cryptography
ELF	Extensible Linking Format
EEPROM	Electrically Erasable Programmable Read-Only Memory
EDR	Enhanced Data Rate
FreeRTOS	Free Real-Time Operating System
FW	Firmware
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GCC	GNU Compiler Collection
GDB	GNU Debugger
GFSK	Gaussian Frequency-Shift Keying
GPADC	General Purpose Analog-to-Digital Converter
GPIO	General-purpose input/output
HMAC	Hash-based Message Authentication Code
HID	Human Interface Device
HCI	Host Controller Interface
HTML	HyperText Markup Language
HW	Hardware
I2C	Inter-Integrated Circuit
IAS	Immediate Alert Service
IC	Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IVT	Interrupt Vector Table
LE	Low Energy
LL	Link Layer
L2CAP	Logical Link Control and Adaptation Protocol
LLD	Low-Level Drivers
MAC	Media Access Control
MCIF	Monitor and Control Interface
MITM	Man In The Middle
MPS	Maximum Payload Size
MTU	Maximum Transmission Unit
NVM	Non-volatile memory
OS	Operating System
OSAL	OS Abstraction Layer
OTP	One-Time Programmable
PDM	Pulse Density Modulation
PHY	Physical Layer
PLL	Phase-Locked Loop
PSM	Protocol Service Multiplexer

PCB	Printed Circuit Board
QSPI	Queued Serial Peripheral Interface
RAM	Random-Access memory
RC16	16 MHz Oscillate
RCX	10.5 kHz Oscillator
RF	Radio Frequency
ROM	Read-Only Memory
RTS/CTS	Request to Send / Clear to Send
SDIO	Secure Digital Input Output
SDK	Software Development Kit
SDU	Service Data Unit
SM	Security Manager
SMP	Security Manger Protocol
SIG	Special Interest Group
SIP	Serial Peripheral Interface
SW	Software
SoC	System on Chip
SRC	Sample Rate Converter
SUOTA	Software Upgrade Over The Air
TCS	Trim and Calibration Section
TRNG	True Random Number Generator
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VBAT	Battery supply voltage
VBUS	External supply voltage (from USB)
VES	Virtual EEPROM
XiP	Executing in Place
XTAL16	16 MHz Crystal oscillator

2 References

[1] DA14681_FS_v2.1, Datasheet, Dialog Semiconductor.

[2] UM-B-057-SmartSnippets Studio user guide, User manual, Dialog Semiconductor.

[3] UM-B-056 DA1468x Software Developer's Guide, User manual, Dialog Semiconductor.

[4] UM-B-047 DA1468x Getting Started, User manual, Dialog Semiconductor

[5] RFC 2104, HMAC: Keyed-Hashing for Message Authentication

[6] FIPS PUB 198-1, The Keyed-Hash Message Authentication Code (HMAC)

[7] NIST, Special Publication 800-56A, Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography Revision 2

[8] Bernstein, Daniel J. "Curve25519: New Diffie-Hellman Speed Records", in Proceedings of Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006.

[9] BLUETOOTH SPECIFICATION Version 4.2

[10] AN-B-045 Application Note: DA14681 Supported QSPI Flash Devices

[11] AN-B-035 Application Note DA1468x Battery Charging Version 1.1

[12] AN-B-075 Application Note DA1468x State of Charge Functionality Version 1.2

3 Prerequisites

- SmartSnippets™ Studio package
- Dialog's Semiconductor SmartSnippets™ DA1468x SDK
- Operating System (Windows or Linux)
- ProDK DA1468x and accessories

4 An Overview of Bluetooth® low energy Platform

Bluetooth® low energy technology was introduced in 2010 as part of the Bluetooth® version 4.0 Core Specification published by the Bluetooth Special Interest Group (SIG). Starting from Version 4.0 onwards, the Bluetooth standard supports two distinct wireless technology systems: the Bluetooth low energy and the Basic Rate (BR), often referred as Basic Rate / Enhanced Data Rate (BR/EDR).

During the early stages of Bluetooth low energy design, the SIG focus was on developing a low complexity radio standard with the lowest possible power consumption, offering low bandwidth optimization, thus enabling low cost applications. In this context, Bluetooth low energy was designed to transmit very small packets of data each time, while consuming significantly less power than similar BR/EDR devices. Moreover, its design also supports efficient implementations having a tight energy and silicon budget, facilitating applications to operate for an extended period of time using a single coin cell battery.

Note 1 The following sections are based on the book "Getting Started with Bluetooth Low Energy" by Kevin Townsend, Carles Cufí, Akiba, Robert Davidson.

4.1 Devices Mode

Devices that support Bluetooth® low energy and BR/EDR are referred as dual-mode devices and are branded as Bluetooth®. Typically, inside the Bluetooth ecosystem, a mobile phone or laptop computer is considered a dual-mode device, unless specifically stated otherwise. Devices that only support Bluetooth low energy are referred to as single-mode devices.

Figure 1: Bluetooth® Branding

4.1.1 Single Mode Devices

A Single-mode (Bluetooth low energy) device only implements Bluetooth low energy. It can communicate with both single-mode and dual-mode devices, however not with devices that only support BR/EDR. Bluetooth low energy support is a must-have for single-mode devices to handle incoming messages and issue a response.

4.1.2 Dual Mode Devices

A Dual-mode BR/EDR/LE, Bluetooth low energy device, implements both BR/EDR and Bluetooth low energy and can communicate with any Bluetooth device.

4.2 Main Building Blocks

In the classic Bluetooth standard, the protocol stack consists of two blocks; the Controller and the Host. In Bluetooth BR/EDR devices, these two are usually implemented separately. However, more recent Bluetooth devices include an increased level of integration. The main building blocks that exist in almost every Bluetooth device are the following:

- The Application that uses the Bluetooth protocol stack interface to implement a particular use case.
- The Host that contains the upper layers of the Bluetooth protocol stack.
- The Controller that contains the lower layers of the Bluetooth protocol stack, including the radio.

Bluetooth specifications also offer a standard communication protocol between the host and the controller called Host Controller Interface (HCI), which allows interoperability between hosts and controllers when these are developed by different entities.

4.3 Hardware configurations

These main building blocks can be implemented in a single integrated circuit (IC) or System on Chip (SoC) device, or they can be split and executed in more than one ICs that are connected through a suitable communication interface and protocol (UART, USB, SPI, or other).

4.3.1 Integrated Processor

Most sensor applications tend to use the SoC hardware configuration as it reduces overall system complexity and associated printed circuit board (PCB) realization costs.

4.3.2 External Processor

Powerful computing devices like smartphones and tablets usually opt for the external processor, with the corresponding HCI protocol which may be either proprietary or standard. This approach also allows additional Bluetooth low energy connectivity with specialized microcontrollers to be integrated without modifying the overall design.

Figure 2 shows a comparison between the two approaches when Bluetooth is implemented:

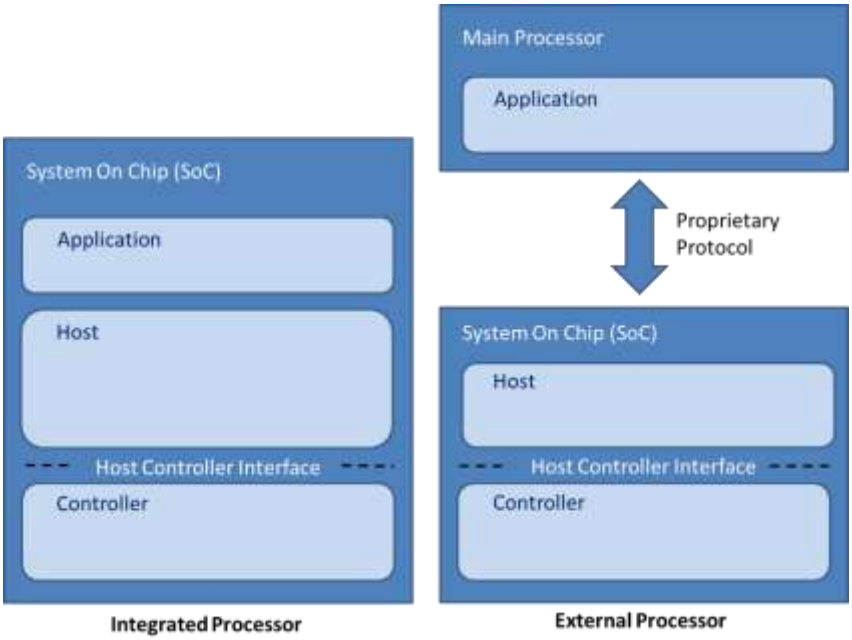

Figure 2: Integrated vs external processor BLE hardware configurations

4.4 Network Modes

Bluetooth low energy devices use two distinct communication methods, each with certain benefits and limitations: Broadcasting and Connecting. Both methods follow certain procedures established by the Generic Access Profile (GAP) as described in Section 4.5.1.

4.4.1 Broadcasting

When using connectionless broadcasting, a Bluetooth low energy device sends data out to any scanning device or receiver that is within acceptable listening range. Essentially, this mechanism

allows a Bluetooth low energy device to send data out one-way to anyone or anything that is able to pick up the transmission.

Broadcasting defines two separate roles:

- **Broadcaster**: Sends non-connectable advertising packets periodically to anyone willing to receive them.
- **Observer**: Repeatedly scans the pre-set frequencies to receive any non-connectable advertising packets.

Broadcasting is the only way for a device to transmit data to more than one peer at a time. The broadcast data is sent out using the advertising features of Bluetooth low energy.

4.4.2 Connecting

For bi-directional data transmission in Bluetooth low energy a connection needs to be present. A connection in Bluetooth low energy is nothing more than an established, periodical exchange of data at certain specific points in time (connection events) between the two Bluetooth low energy peers involved in it. Typically, the data are exchanged only between the two Bluetooth low energy connection peers, and no other device is involved. Connections define two separate roles:

- **Central (master):** Repeatedly scans the pre-set Bluetooth low energy frequencies for connectable advertising packets and, when suitable, initiates a connection. Once the connection is established, the central manages the timing and initiates the periodical data exchanges.
- **Peripheral (slave):** A device that sends connectable advertising packets periodically and accepts incoming connections. Once in an active connection, the peripheral follows the central's timing and exchanges data regularly with it.

For a connection to be initiated, the central device picks up the connectable advertising packets from a peripheral and then sends a request to the peripheral device to establish an exclusive connection between the two devices. Once the connection is established, the peripheral stops advertising and the two devices can begin exchanging data in both directions. Although the central is the device that manages the connection establishment, data can be sent independently by either device during each connection event, and the roles do not impose restrictions in data throughput or priority. It is therefore possible for a device to act as a central and a peripheral at the same time, for a central device to be connected to multiple peripherals as well as for a peripheral device to be connected to multiple centrals.

Connections provide the ability to organize the data with much finer-grained control over each field or property using additional protocol layers, more specifically, the Generic Attribute Profile (GATT). GATT organizes data around units called services and characteristics. Moreover, connections allow for higher throughput and support the establishment of a secure encrypted link, as well as negotiation of connection parameters to fit the data model.

A Bluetooth low energy device can have multiple services and characteristics, organized in a meaningful structure called a GATT Table. Services can contain multiple characteristics, each with their own access rights and descriptive metadata.

4.5 Profiles

The Bluetooth specification clearly separates the concept of Protocol and Profile. This distinction is made due to the different purposes each concept serves and the overall specifications are divided into:

- **Protocols:** They are the building blocks used by all devices conforming to the Bluetooth specification; protocols are essentially forming the layers that implement the different packet formats, routing, multiplexing, encoding, and decoding that allow data to be sent effectively between peers.

- **Profiles:** which are vertical slices of functionality defining either basic modes of operation required by all devices (such as the Generic Access Profile and the Generic Attribute Profile) or specific use cases (Proximity Profile, Glucose Profile). Profiles essentially specify how protocols should be used to achieve an objective, whether generic or specific.

4.5.1 Generic Profiles

Generic profiles are defined by the Bluetooth specification and two of them are fundamental as they ensure the interoperability between Bluetooth low energy devices from different vendors:

- **Generic Access Profile (GAP):** Specifies the usage model of the lower-level radio protocols to define roles, procedures, and modes that allow devices to broadcast data, discover devices, establish connections, manage connections, and negotiate security levels; GAP is essentially, the uppermost control layer of Bluetooth low energy. This profile is mandatory for all Bluetooth low energy devices, and all must comply with it.
- **Generic Attribute Profile (GATT):** Addresses data exchanges in Bluetooth low energy and specifies the basic data model and procedures to allow devices to discover, read, write, and push data elements between them. It is basically, the topmost data layer of Bluetooth low energy.

GAP and GATT are so fundamental to Bluetooth low energy that they are often used as the base for the provision of application programming interfaces (APIs) that act as the entry point for the application to interact with the protocol stack.

4.5.2 Use-Case-Specific Profiles

Use-case-specific profiles are usually limited to GATT-based profiles. Typically these profiles use the procedures and operating models of the GATT profile as a base building block for all further extensions. However, in version 4.1 of the specification, Logical Link Control and Adaptation Protocol (L2CAP) connection-oriented channels have been introduced, which indicates that GATT-less profiles are also possible.

4.5.2.1 SIG-defined GATT-based profiles

In addition to providing a solid reference framework for the control and data layers of devices involved in a Bluetooth low energy network, the Bluetooth SIG also provides a predefined set of use-case profiles based on GATT. These completely cover all procedures and data formats required to implement a wide range of specific use cases such as:

- **Find Me Profile:** it allows devices to physically locate other devices (for example using a smartphone to find a Bluetooth low energy enabled keyring, or vice versa).
- **Proximity Profile:** it detects the presence or absence of nearby devices (beep if an item is forgotten when leaving an area like a room).
- **HID over GATT Profile:** it transfers Human Interface Device (HID) data over Bluetooth low energy (for keyboards, mice, remote controls).
- **Glucose Profile:** it securely transfers glucose levels over Bluetooth low energy.
- **Health Thermometer Profile:** it transfers body temperature readings over Bluetooth low energy.

The Bluetooth SIG's Specification in its Adopted Documents page provides a full list of SIG-approved profiles (for more information please visit https://www.bluetooth.com/specifications/adopted-specifications). A developer can also browse directly the list of all currently adopted services for the Bluetooth services and characteristics at the Bluetooth Developer Portal.

4.5.2.2 Vendor-Specific Profiles

Vendors are allowed by the Bluetooth specification to define their own profiles for use cases that are not covered by the SIG-defined profiles. Those profiles can be kept private to the two peers involved in the use case (for example, a new sensor accessory and a Smartphone application), or they can

also be published by the vendor so that other parties can provide implementations of the profile based on the vendor-supplied specification. An example of a published vendor-specific profile is Apple's iBeacon.

4.5.3 Generic Access Profile Layer

The **Generic Access Profile (GAP)** layer is responsible for the overall connection functionality; it handles the device's access modes and procedures including device discovery, directly interfacing with the application and/or profiles, and handling device discovery and connection-related services for the device. In addition, GAP takes care of the initiation of security features.

Essentially, GAP can be considered as Bluetooth low energy's upper control layer, given that it specifies how devices perform control procedures such as device discovery and secure connection establishment. This ensures interoperability and thus allows data exchange between devices from different vendors.

GAP specifies four roles that a device can adopt in a Bluetooth low energy network:

- **Broadcaster:** The device is advertising with specific data, letting any initiating devices know for example that it is a connectable device. This advertisement contains the device address and optional additional data such as the device name.
- **Observer:** When a scanning device receives an advertisement it sends a "scan request" to the advertiser. The advertiser responds with a "scan response". This is the process of device discovery, after which the scanning device is aware of the presence of the advertising device, and knows that it is possible to establish a connection with it.
- **Central:** when initiating a connection, the central must specify a peer device address to connect to. If an Advertisement is received which matches the peer device's address, the central device will then send out a request to establish a connection (link) with the advertising device with a set of connection parameters.
- **Peripheral:** once a connection is established, the device will function as a slave if it was the advertiser and as master if it was the initiator.

Fundamentally, GAP establishes different sets of rules and concepts that regulate and standardize the low-level operation of devices, in particular:

- The Roles and interaction between them.
- The Operational modes and transitions across those devices.
- The Operational procedures to achieve consistent and interoperable communication.
- All Security aspects, including security modes and procedures.
- Additional data formats for non-protocol data.

4.5.4 Generic Attribute Profile Layer

The **Generic Attribute Profile (GATT)** layer is a service framework that defines all sub-procedures for using the Attribute Protocol (ATT). It describes in detail how profile and user data is to be exchanged over a Bluetooth low energy connection. In contrast to GAP which defines the low-level interactions with devices, GATT deals only with actual data transfer procedures and formats.

GATT also provides the reference framework for all the GATT-based profiles defined by the SIG. Effectively by covering the precise use cases for the profiles, it ensures interoperability between devices from different vendors; all the standard Bluetooth low energy profiles are therefore based on GATT and must comply with it to operate correctly. This makes GATT a key section of the Bluetooth low energy specification, since every data collection that is relevant to applications and users must be formatted, packed, and transmitted according to its rules.

GATT defines two roles for the interacting Bluetooth low energy devices:

- **Client:** It sends requests to a server, receives responses and potentially server initiated updates and notifications as well. The GATT client does not know anything in advance about

the server's attributes, so it must first inquire about the presence and nature of those attributes by performing service discovery. After completing service discovery, it can start reading and writing the attributes found in the server, as well as receiving server-initiated updates and notifications. It corresponds to the ATT client.

- **Server:** It receives requests from a client and issues responses. It also sends server-initiated updates and notifications when configured to do so. The server role is responsible for organizing the user data in attributes and making them available to the client. Every Bluetooth low energy device sold must include at least a basic GATT server that can respond to client requests, even if only to return an error response. It corresponds to the ATT server.

It is worth mentioning once again that GATT and GAP roles are completely independent yet concurrently compatible to each other. For instance, it is possible for both a GAP central and a GAP peripheral to act as a GATT client or server, or even both at the same time.

GATT uses ATT as a transport protocol for data exchange between devices. This data is organized hierarchically in sections called services, which group conceptually related pieces of user data called characteristics.

4.6 Protocol Stack

A single-mode Bluetooth low energy device is from an architecture point of view similar to all Bluetooth devices in that it is divided into three blocks: controller, host, and application. These basic building blocks each consist of several layers which are tightly integrated in the so-called Protocol Stack as shown in Figure 3:

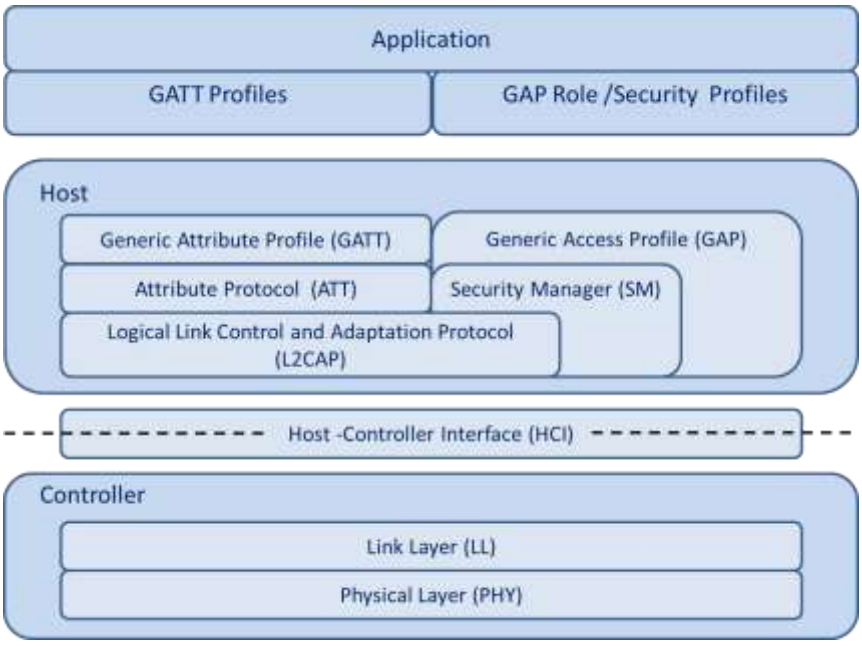

Figure 3: Bluetooth low energy Protocol Stack Layers

The following sections summarize each of these blocks along with the layers each one covers.

4.7 Controller

The **Controller** includes all the lower level functionality necessary for a Bluetooth low energy device to communicate; it consists of the Physical Layer (PHY), the Link Layer (LL) and the controller side of the Host Controller Interface (HCI).

4.7.1 Physical Layer (PHY)

In the **Physical Layer (PHY)** the key block is the 1Mbps adaptive frequency-hopping Gaussian Frequency-Shift Keying (GFSK) radio. This operates in the unlicensed 2.4 GHz Industrial, Scientific, and Medical (ISM) band.

4.7.2 Link Layer (LL)

The **Link Layer (LL)** directly interfaces with the PHY; it is the hard real-time layer of the protocol stack as it must comply with all the timing requirements defined in the specification. Given that many of the calculations performed by the LL are computationally expensive, they are usually implemented with hardware accelerators. This helps prevent overloading of the Central Processing Unit (CPU) that runs all software layers in the stack, therefore the LL implementation is a combination of custom hardware and software. The functionality provided by the LL usually includes Preamble, Access Address, air protocol framing, CRC generation and verification, data whitening, random number generation and AES encryption. It is usually kept isolated from the higher layers of the protocol stack by an interface that hides this complexity and its real-time requirements.

The LL principally controls the Radio Frequency (RF) state of the device and manages the link state of the radio which is how the device connects to other devices. A Bluetooth low energy device can be a **master**, a slave, or both depending on the use case and the corresponding requirements. A master can connect to multiple slaves and a slave can be connected to multiple masters. Typically, devices such as smartphones or tablets tend to act as a master, while smaller, simpler, more memory-constrained devices such as standalone sensors generally adopt the slave role. A device can only be in one of the following five states: **standby, advertising, scanning, initiating,** or **connected** as shown in Figure 4:

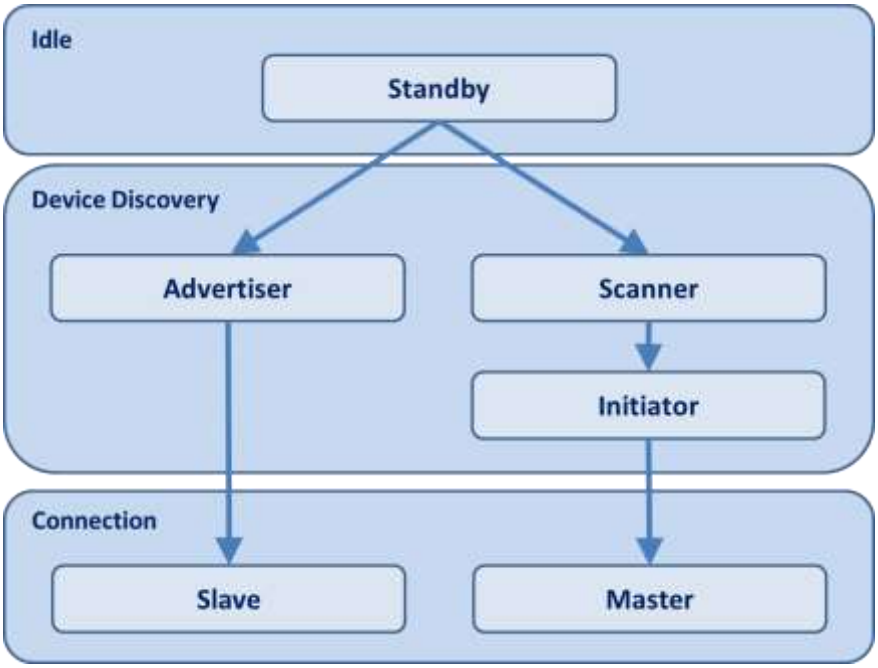

Figure 4: Link Layer States

Advertisers transmit data without being connected, while scanners listen for advertisers. An initiator is a device that is responding to an advertiser with a connection request. If the advertiser accepts the connection request, both the advertiser and initiator will enter a connected state. When a device is in a connection state, it will be connected in one of two roles: master or slave. Typically, devices that initiate connections will be masters and devices that advertise their availability and accept connections will be slaves. Therefore, the Link Layer defines the following roles:

- **Advertiser**: a device sending advertising packets.
- **Scanner**: a device scanning for advertising packets.
- **Master**: a device that initiates a connection (initiator) and manages it later.
- **Slave**: a device that accepts a connection request and follows the master's timing.

These roles can be logically grouped into two pairs: advertiser and scanner (when not in an active connection) and master and slave (when in a connection).

4.7.2.1 Bluetooth Device Address

The Bluetooth device address is the primary identifier of a Bluetooth device. This is just the same as an Ethernet Media Access Control (MAC) address which uniquely identifies a wired ethernet device. It is a 48-bit (6-byte) number that uniquely identifies a device among peers. There are two types of device addresses, and it is possible for a device to obtain one or both types:

- Public device address

This is the equivalent to a fixed, factory-programmed device address as used in BR/EDR devices as well. It must be registered with the Institute of Electrical and Electronics Engineers (IEEE) Registration Authority and should never change throughout the device's lifetime.

- Random device address

This address can either be pre-programmed or dynamically generated at runtime on the device. There are numerous use cases in which such addresses are useful in Bluetooth low energy.

4.7.2.2 Advertising and Scanning

The Bluetooth low energy specification allows only one packet format and two types of packets, **advertising** and **data**.

Advertising packets are used for two purposes:

- To broadcast data for applications that do not need the overhead of a full connection establishment. This is used in Beacon applications.
- To discover slaves and connect with them so that data can be exchanged.

Data packets are used for user data transport between the master and the slave devices, in a bi-directional manner.

Finally, the Link Layer acts as a reliable data bearer since all received packets are checked against a 24-bit Cyclic Redundancy Check (CRC) and retransmissions are scheduled when the error checking mechanism detects a transmission failure. Since there is no pre-defined retransmission upper bound, the Link Layer will continuously resend the packet until it is finally acknowledged by the receiver.

4.7.3 Host Controller Interface – Controller side

The **Host Controller Interface (HCI)** interface at the Controller side, provides a mean of communication to the host via a standardized interface; the Bluetooth specification defines HCI as a set of commands and events for the host and the controller to interact with each other, along with a data packet format and a set of rules for flow control and other procedures. Additionally, the spec defines several transports, each of which augments the HCI protocol for a specific physical transport (UART, USB, SDIO, etc.).

4.8 Host

The Host block consists of a set of layers, each with specific role and functionality, which communicate with each other to make the overall block operate. As shown in Figure 3 these layers are the Logical Link Control and Adaptation Protocol (L2CAP), the Attribute Protocol (ATT), the Security Manager (SM) and finally the Generic Attribute Profile (GATT) and Generic Access Profile (GAP).

4.8.1 Host Controller Interface – Host Side

The HCI interface at the Host side provides a mean of communication to the controller via a standardized interface. As it matches the Controller Side HCI, this layer can be implemented either through a software API or over a hardware interface (UART, SDIO, USB etc).

4.8.2 Logical Link Control and Adaptation Protocol

The **Logical Link Control** and **Adaptation Protocol (L2CAP)** layer provides data encapsulation services to the upper layers, thus allowing logical end-to-end communication using data transfer. Essentially, it serves as a protocol multiplexer that takes multiple protocols from the upper layers and encapsulates them into the standard Bluetooth low energy packet format and vice versa. L2CAP is also responsible for package fragmentation and reassembly. During this process large packets originating from the upper layers of the transmitting side are fitted into the 27-byte maximum payload size of the Bluetooth low energy packets. The reverse process takes place at the receiving end, where the fragmented large upper layer packets are reassembled from multiple small Bluetooth low energy packets and transmitted up towards the appropriate upper level entity.

The L2CAP layer is in charge of routing two main protocols: the Attribute Protocol (ATT) and the Security Manager Protocol (SMP). Moreover, L2CAP can create its own user-defined channels for high-throughput data transfer, a feature called LE Credit Based Flow Control Mode.

4.8.3 Attribute Protocol

The **ATT** layer enables a Bluetooth low energy device to provide certain pieces of data, known as **attributes**, to another Bluetooth low energy device via a standardized interface. In the context of ATT, the device exposing attributes is referred to as the **server** and the peer device interested in working with these attributes is referred to as the **client**. The Link Layer state (**master** or **slave**) of the device is independent from the ATT role of the device. For example, a master device may either be an ATT server or an ATT client, while a slave device may also be either an ATT server or an ATT client. It is also possible for a device to be both an ATT server and an ATT client simultaneously.

Essentially ATT is a simple client/server stateless protocol based on the attributes presented by a device. A client requests data from a server, and a server sends data to clients. The protocol is strict which means that in case of a pending request (i.e. no response has yet been received for a previously issued request), no further requests can be submitted until the response to the first request is received and processed. This applies to both directions independently in the case where two peers are acting both as a client and server.

Each ATT server contains data organized in the form of attributes, each of which is assigned a 16-bit **attribute handle**, called a **Universally Unique Identifier (UUID)**, a set of permissions, and finally a value. Effectively, the attribute handle is an identifier used to access an attribute value. The UUID specifies the type and nature of the data contained in the value. When a client wants to read or write attribute values from or to a server, it issues a read or write request to the server using the attribute handle. The server will respond with the attribute value or an acknowledgement. In the case of a read operation, it is up to the client to parse the value and understand the data type based on the UUID of the attribute. On the other hand, during a write operation, the client is expected to provide data that is consistent with the attribute type and the server is free to reject the write operation if the data is not in the specified format.

4.8.4 Security Manager

The **Security Manager (SM)** layer defines the method for pairing and key distribution and provides functions for the other layers of the protocol stack to securely connect and exchange data with another Bluetooth low energy device. It includes both a protocol and a series of security algorithms that are designed to provide the Bluetooth low energy protocol stack with the ability to generate and exchange security keys. This allows the peers to communicate securely over an encrypted link, to trust the identity of the remote device, and if required, to hide the public Bluetooth Address. It defines two roles:

- **Initiator:** which always corresponds to the Link Layer master
- **Responder:** which always corresponds to the Link Layer slave

Moreover, it provides support for the following three procedures:

- **Pairing**: the procedure by which a security encryption key is generated and manipulated to enable a secure encrypted link. This key is temporary and not stored or available for subsequent connections.
- **Bonding**: a sequence of pairing followed by the generation and exchange of permanent security keys. These are typically stored in non-volatile memory and therefore enable the creation of a permanent bond between two devices, which will allow them to quickly set up a secure link in subsequent connections without having to perform a bonding procedure again.
- **Encryption Reestablishment**: after a bonding procedure is complete, keys might have been stored on both sides of the connection. If encryption keys have been stored, this procedure defines how to use those keys in subsequent connections to re-establish a secure, encrypted connection without having to go through the pairing (or bonding) procedure again.

Pairing can therefore create a secure link that will only last for the lifetime of the connection. Bonding will create a permanent association (also called **bond**) in the form of shared security keys that will be used in later connections until either side decides to delete them. Sometimes documentation and APIs use the term **pairing with bonding** instead of simply bonding, since a bonding procedure always includes an initial pairing phase.

Although it is always up to the initiator to request the start of a specific security procedure, the responder can asynchronously request the start of any of the procedures listed above. There are no guarantees however for the responder that the initiator will fulfil the request. Therefore, the request is optional rather than binding. This security request can logically be issued only by the slave or the peripheral end of the connection.

5 The DA1468x Software Platform Overview

SmartSnippets™ is Dialog's Software Development Platform provided with the Smartbond™ DA1468x devices family. It can be considered as a complete platform, targeted at single-CPU Bluetooth low energy applications development.

The SmartSnippets™ Studio software development platform provides a complete Bluetooth low energy Application development environment to enable the full potential of the DA1468x architecture. It is based on a GNU Compiler Collection (GCC)/ Debugger (GDB) toolchain, a preconfigured Eclipse CDT IDE and a set of utilities.

The SDK provides a Bluetooth stack, a Board Support Package and Middleware Services which all run within an RTOS environment which ensures that all real-time requirements are met.

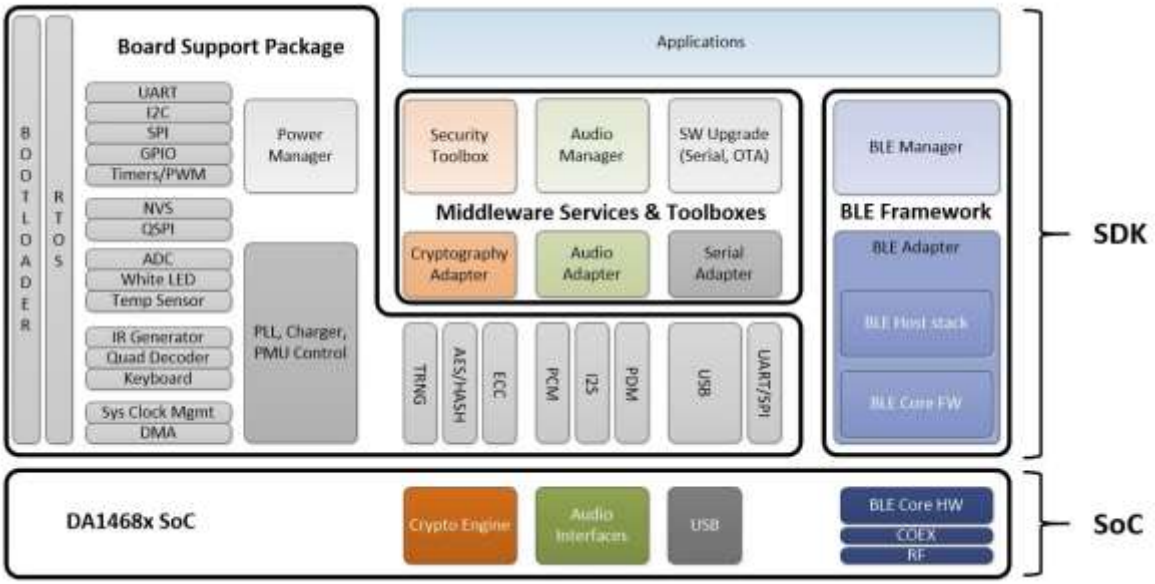

Figure 5: SmartSnippets™ Bluetooth low energy development platform overview

DA1468x also provides a System Management driver that optimizes Power Consumption by automatically putting the system in the lowest sleep mode whenever it is inactive. The application can alter the default behavior by defining which sleep mode to use when there is no activity or even specify that the system stays awake.

The SmartSnippets™ DA1468x SDK contains three main software packages that allow developers to easily implement complex applications and fully exploit the hardware capabilities of a DA1468x board (or Development Kit):

- **Board Support Package**: Platform-specific source files for Peripherals Drivers, OS, system configuration/management and memory management.
- **Bluetooth low energy Framework:** a Framework of tasks and queues that allow access to the BLE interface through a simple, comprehensible API.
- **Middleware Services & Toolboxes**: Features frequently needed services for Bluetooth low energy based application like Monitor and Control Interface (MCIF) and Software Upgrade over the Air (SUOTA).

5.1 Board Support Package Overview

5.1.1 Low-level Drivers

Board Support Package (BSP) contains Low-level Drivers (LLDs) for all peripherals, interfaces, timers and HW accelerators. These are Hardware Libraries providing the lowest level API to drive a HW resource. LLDs are not Thread safe and do not rely on any OS.

5.1.2 RTOS

The SmartSnippets™ DA1468x SDK is based on a real-time preemptive Operating System named FreeRTOS (www.freertos.org). FreeRTOS is a light-weight open source OS, widely used by many embedded systems across different microcontroller architectures.

Note 2 An OS Abstraction Layer (OSAL) is provided so another RTOS could be used instead.

5.1.3 System Manager

The System Manager is responsible for providing the following services:

- A Clock and Power Management Service. It implements an automated configurable sleep/wake-up engine.
- A Watchdog Service that monitors the system's status by checking the state of all registered tasks.
- A Real-Time Clock service that can be used from any execution context including Interrupt context.

5.1.4 Adapters

As the SmartSnippets™ DA1468x SDK relies on a multi-tasking environment, resource sharing is critical and is achieved by following a multi-layered architecture which introduces the concept of adapters which manage the access to resources between different tasks.

An application task that accesses shared resources such as cryptographic engines must use adapters and must not directly call Low-level drivers. Adapters also handle power management operations related to the controlled resource, hiding power management details from the application, such as blocking system sleep when the controlled HW resource is busy, or restoring HW configuration upon system wake up.

5.1.5 The BLE Framework

The BLE framework provides an abstraction layer that simplifies application development by hiding low level details of the specification. It allows performing any of the standard Central and/or Peripheral operations including security.

5.2 Middleware Services

A set of Middleware Services allow faster development by providing a high level API for the user application into a management framework that handles the state machines and driver calls for a specific service.

5.2.1 SUOTA

The Bluetooth low energy platform allows the user to update the software of the device wirelessly. This process is called Software Upgrade Over The Air (SUOTA). For more information about SUOTA refer to chapter 9 of Software Developers Guide [3].

5.2.2 Security Toolbox

The Security Toolbox provides a collection of high-level cryptographic algorithms. It follows a layered software architecture approach that consists of the LLDs for the hardware cryptographic engines, the system services and adapters that provide higher level APIs for using the engines and the algorithm implementations. For more information refer to section 8.

6 Using the Operating System

6.1 FreeRTOS

The SmartSnippets™ DA1468x SDK contains the FreeRTOS v8.2.0 kernel, updated with a few back-ported bug fixes from v8.2.1 and v8.2.3. The FreeRTOS API documentation is available online at http://www.freertos.org/modules.html#API_reference.

The SmartSnippets™ DA1468x SDK contains several FreeRTOS-based example applications. The corresponding Eclipse projects use (as a convention) the folder `<sdk_root_directory>/sdk/bsp/free_rtos/` to include the FreeRTOS kernel source files.

6.1.1 FreeRTOS Source Files

The FreeRTOS kernel source files are in the following directory tree:

```
<sdk_root_directory>/sdk/bsp/free_rtos/
├── include
└── portable
    ├── GCC
    │   └── ARM_CM0
    └── MemMang
```

The kernel source files are in `<sdk_root_directory>/sdk/bsp/free_rtos/`:

Table 1: Kernel source files for FreeRTOS

Source Files	Description
croutine.c	Not used in the SmartSnippets™ DA1468x SDK
event_groups.c	Implementation of the Event Groups API
list.c	List data structure implementation, used internally by FreeRTOS
queue.c	Implementation of the Queue and QueueSet API
tasks.c	Implementation of the Tasks API
timers.c	Implementation of the Timers API

The FreeRTOS header files are in `<sdk_root_directory>/sdk/bsp/free_rtos/include/`:

Table 2: Header files for FreeRTOS

Header files	Description
FreeRTOS.h	Must be always included first.
FreeRTOSConfig.h	Configuration options, included by FreeRTOS.h.
StackMacros.h	FreeRTOS internal.
croutine.h	Not used in the SmartSnippets™ DA1468x SDK.
deprecated_definitions.h	FreeRTOS legacy definitions.
event_groups.h	Event Groups API.
list.h	FreeRTOS internal.
mpu_wrappers.h	FreeRTOS internal.
portable.h	API that is platform-dependent.
projdefs.h	FreeRTOS basic constants and macros.
queue.h	Queue and QueueSet API.
semphr.h	Semaphore API.
task.h	Tasks API.
timers.h	Timers API.

Some FreeRTOS code is specific for the compiler and the platform processor. The corresponding files are located under `<sdk_root_directory>/sdk/bsp/free_rtos/portable/`:

```
├── GCC
│   └── ARM_CM0
│       ├── port.c (DA1468x-specific customizations for timers and interrupts)
│       └── portmacro.h (DA1468x-specific customizations for constants and sections)
└── MemMang
    └── heap_4.c (heap memory manager)
```

The FreeRTOS kernel provides different heap memory algorithms (see http://www.freertos.org/a00111.html for descriptions). The SmartSnippets™ DA1468x SDK employs the `heap_4` algorithm, which consolidates adjacent free blocks to avoid fragmentation. The memory area allocated for the heap is assumed to be contiguous (i.e. there are no “holes”).

Note 3 The SmartSnippets™ DA1468x SDK has only been tested with `heap_4` algorithm.

6.1.2 FreeRTOS Configuration

Several FreeRTOS configuration options (see http://www.freertos.org/a00110.html for description) for SmartSnippets™ SDK are specified in `<sdk_root_directory>/sdk/bsp/free_rtos/include/FreeRTOSConfig.h` using macro definitions which are described below. For the rest of the configuration options, the defaults are used.

Table 3: Macro Definitions for the FreeRTOSConfig.h

Name	Default Definitions	Default Value
`configUSE_PREEMPTION`	Use the preemptive scheduler.	`1`
`configUSE_IDLE_HOOK`	Do not use an idle hook (callback).	`0`
`configUSE_TICK_HOOK`	Do not use a system-tick hook (callback).	`0`
`configIDLE_SHOULD_YIELD`	The idle task immediately yields if there are any tasks that can be scheduled.	`1`
`configUSE_MUTEXES`	The mMutex functionality is enabled.	`1`
`configCHECK_FOR_STACK_OVERFLOW`	The extended stack overflow detection is enabled.	`2`
`configUSE_RECURSIVE_MUTEXES`	The recursive mutex functionality is enabled.	`1`
`configUSE_MALLOC_FAILED_HOOK`	Trigger hook (callback) on memory allocation failure.	`1`
`configUSE_COUNTING_SEMAPHORES`	The counting semaphore functionality is enabled.	`1`
`configGENERATE_RUN_TIME_STATS`	No run-time statistics are generated.	`0`
`configUSE_QUEUE_SETS`	The QueueSet functionality is enabled.	`1`
`configUSE_TICKLESS_IDLE`	The periodic tick interrupt is disabled during idle intervals, to save energy.	`2`

The hardware timer “Timer1” is dedicated to FreeRTOS and runs as the system timer. Due to the width of the Timer1 period and the available pre-scaling options, the maximum time that a DA1468x-based system can stay in sleep mode is 8 seconds.

Finally, the macro `config_ASSERT()` is defined in:

<sdk_root_directory>/sdk/bsp/free_rtos/include/FreeRTOSConfig.h as:

```
/* Normal assert() semantics without relying on the provision of an assert.h header file.
*/

#if (dg_configIMAGE_SETUP == DEVELOPMENT_MODE)

# define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS();
hw_watchdog_freeze(); do {} while(1); }

#else

# define configASSERT( x ) do { } while (0)

#endif
```

Code 1: Code defining the `config_ASSERT()` macro

Therefore, in development builds it generates a breakpoint, while in production builds it has no effect.

6.1.3 Platform-specific Definitions

As mentioned in Section 6.1.1 the `portmacro.h` file in `<sdk_root_directory>/sdk/bsp/free_rtos/portable/GCC/ARM_CM0/` path customizes certain definitions (attributes) for DA1468x.

FreeRTOS uses the concept of *"privileged"* code and data which is meaningless for DA1468x as an ARM Cortex-M0 does not provide different execution levels (e.g. supervisor and user).

Therefore, on DA1468x the concept of privileged code/data is redefined to denote code/data that should be placed in retained RAM (i.e. RAM areas that stay active even during sleep periods).

The relevant definitions are:

```
/* Code and Data Retention specific constants. */
#if ((dg_configCODE_LOCATION == NON_VOLATILE_IS_FLASH) && (dg_configEXEC_MODE ==
MODE_IS_CACHED))
#define PRIVILEGED_APP_FUNCTION                 __attribute__((section("text_retained")))
#else
#define PRIVILEGED_APP_FUNCTION
#endif

// RetRAM0
#define PRIVILEGED_DATA
__attribute__((section("privileged_data_zi")))
#define INITIALISED_PRIVILEGED_DATA
__attribute__((section("privileged_data_init")))

// RetRAM1
#define PRIVILEGED_DATA_1
__attribute__((section("privileged_data_1_zi")))
```

Code 2: Code and Data Retention specific constants

Symbols (i.e. functions and variables) that are defined with one of the above attributes will be placed in special sections by the linker and will be moved in their appropriate RAM locations at run-time.

6.1.4 FreeRTOS Task Priorities

Note 4 Please do not modify priorities or corresponding settings, as there is a high risk of affecting the stability of time-critical parts of your application.

The priority (FreeRTOS: `uxPriority`) parameter of the `OS_TASK_CREATE` (FreeRTOS: `xTaskCreate()`) API function initially assigns a priority to every newly created task. It is possible to change this priority later, after the scheduler has been started, with the use of the `vTaskPrioritySet()` API function.

The priority that a task can be assigned to is in the range of 0 to (`configMAX_PRIORITIES` – 1). The `configMAX_PRIORITIES` constant is defined in the `FreeRTOSconfig.h` file which is in the `<sdk_root_directory>/sdk/FreeRTOS/include` folder of the project. For example, Code 3 shows the currently configured value for `configMAX_PRIORITIES` for the pxp_reporter demo application.

```
#define configMAX_PRIORITIES                    ( 7 )
```

Code 3: currently configured value for the configMAX_PRIORITIES

The only restriction in the maximum value that `configMAX_PRIORITIES` constant can take is when the port in use implements a port optimized task selection mechanism that uses a 'count leading zeros' type instruction (for task selection in a single instruction) and `configUSE_PORT_OPTIMISED_TASK_SELECTION` is set to 1 in `FreeRTOSconfig.h` file. In this case `configMAX_PRIORITIES` cannot be higher than 32. Otherwise there is no restriction in the maximum value that this constant can take but is advisable to set its value to minimum necessary for RAM efficiency reasons.

The lowest priority that a task can have is zero. The priority of the idle task, which is defined by the `tskIDLE_PRIORITY constant`, is zero. In SmartSnippets™ DA1468x SDK the priority of the idle task is defined in task.h file located in the `<sdk_root_directory>/sdk/FreeRTOS/include` folder as shown in Code 4. A priority assigned to a task is not unique and can be shared among many tasks.

```
/**
 * Defines the priority used by the idle task.  This must not be modified.
 *
 * \ingroup TaskUtils
 */
#define tskIDLE_PRIORITY                ( ( UBaseType_t ) 0U )
```

Code 4: Idle task's priority

As described in the FreeRTOS documentation (http://www.freertos.org/) the states in which a task can exist are:

- Running
- Ready
- Blocked
- Suspended

The task that is in the Running state, is the task with the highest priority among the tasks that are placed in the Ready state. The FreeRTOS scheduler ensures that tasks in the Ready or Running state will get CPU time in comparison with tasks that are also placed in the Ready state but have lower priorities.

Tasks that are in the Ready state and share the same priority will be sharing the available processing time, using a time sliced round robin schedule, unless `configUSE_TIME_SLICING` is defined or `configUSE_TIME_SLICING` is set to 0.

Code 5 below shows the task priorities as defined in the `osal.h` file which is in the folder `<sdk_root_directory>/sdk/bsp/osal`. Table 6 provides information about the wrappers of the Operating System Abstraction Layer.


```
#define OS_TASK_PRIORITY_LOWEST    tskIDLE_PRIORITY
#define OS_TASK_PRIORITY_NORMAL    tskIDLE_PRIORITY + 1
#define OS_TASK_PRIORITY_HIGHEST   configMAX_PRIORITIES - 1
```

Code 5: Task priorities

Table 4 below shows the tasks of the `pxp_reporter` demo application, starting with the lowest priority task. The same information is illustrated in a different way in Figure 6.

Table 4: pxp_reporter tasks

Task	Priority	File	Description
prvIdleTask	tskIDLE_PRIORITY	tasks.c	The idle task.
rcx_calibration_task	tskIDLE_PRIORITY	sys_clock_mgr.c	The RCX calibration task.
pxp_reporter_task	mainPXP_REPORTER_TASK_PRIORITY (OS_TASK_PRIORITY_NORMAL)	main.c	The PXP reporter application task.
ble_mgr_task	mainBLE_MGR_PRIORITY (OS_TASK_PRIORITY_HIGHEST - 4)	ble_mgr.c	The ble manager task.
ad_ble_task	mainBLE_TASK_PRIORITY (OS_TASK_PRIORITY_HIGHEST - 3)	ad_ble.c	The ble adapter task.
usb_charger_task	OS_TASK_PRIORITY_HIGHEST - 2	sys_charger.c	The USB Charger task.
usb_charger_fsm_task	OS_TASK_PRIORITY_HIGHEST - 2	sys_charger.c	The USB Charger FSM task.
prvTimerTask	configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1)	timers.c	The timer task.
system_init	OS_TASK_PRIORITY_HIGHEST	main.c	The System Initialization task.

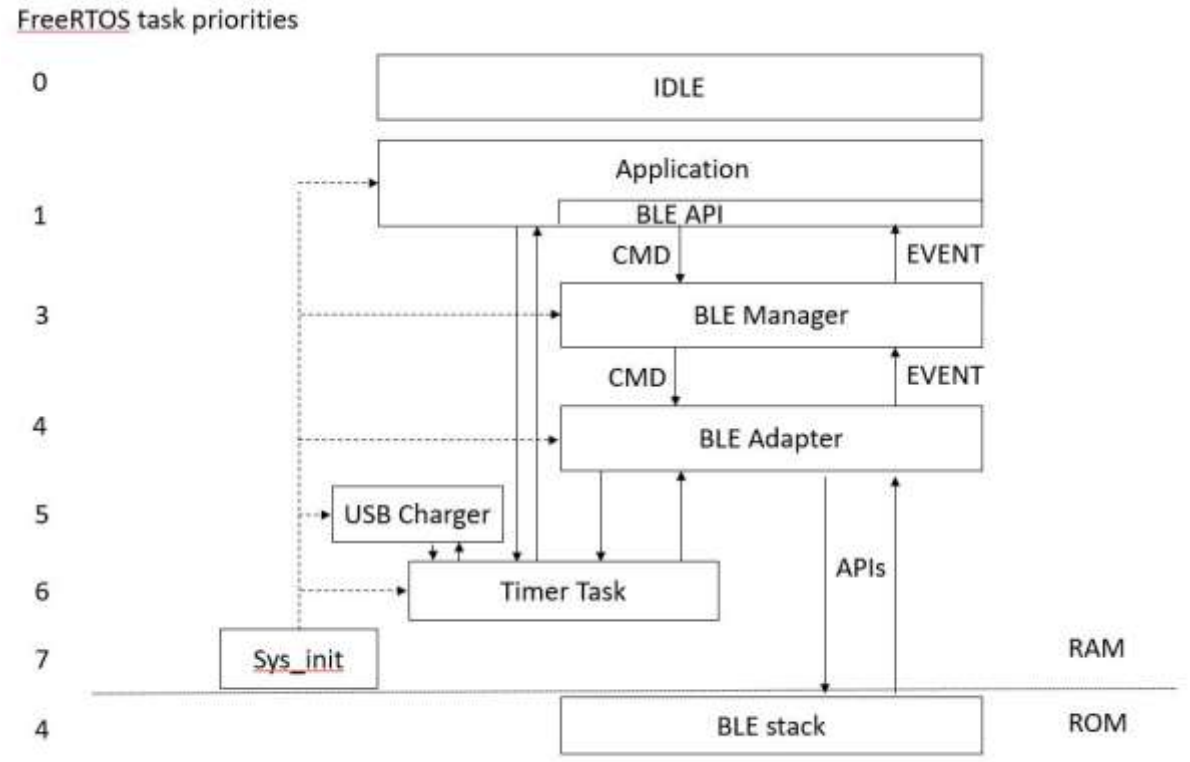

Figure 6: Pxp_reporter task priorities

6.1.5 Delaying the execution of a FreeRTOS Task

The following two functions can be used to delay a FreeRTOS task. They are intended for different lengths of delays:

- `hw_cpm_delay_usec()` can be used to add a delay of a specific number of microseconds in the code execution by executing a loop in assembly code. The function is as accurate as possible (the error is less than or equal to 1usec for clocks higher than or equal to 16MHz) and it will never generate a delay less than requested. It operates correctly with all possible clock setups. As long as the delay is of a few microseconds it is possible to call this function with the interrupts disabled as it will not impact the real time behavior of any other tasks. It does not call the RTOS scheduler as it just creates the delay in the current task.
- `OS_DELAY_MS()` can be used to add a delay in the range of milliseconds in the execution of a task. It will block the task and so the RTOS can schedule other lower priority tasks in the delay period. The delay time requested is converted to system ticks, so its accuracy depends on the tick period which is by default ~2msec. So, a call like `OS_DELAY_MS(5)` would be rounded down to a 2 tick delay which is ~4ms, leading to an error of ~1msec. The bigger the delay setting, the less important is the "tick error" to the actual delay. In addition, this function will block a task for a specific number of OS ticks. Even though it will be unblocked on time, this does not guarantee that it will be executed immediately as this depends on its priority. If a higher priority task is running then the task will be left in the "ready-to-run" state, waiting for the OS scheduler to allow it to run. Finally, since calling this function results in the blocking of the running task, it does not make any sense to call it with the interrupts disabled. Operating System Abstraction Layer

6.1.6 Scope

The SmartSnippets™ DA1468X SDK offers an Operating System Abstraction Layer (OSAL), which:

1. Facilitates porting to an RTOS other than FreeRTOS, by providing RTOS-agnostic wrappers around FreeRTOS Application Programming Interface (API).
2. Provides resource management (i.e. exclusive access) capabilities.
3. Provides a message-queue data structure.

The OSAL source files are located under `<sdk_root_directory>/sdk/bsp/osal/`:

Table 5: Source files for OSAL

Source files	Description
msg_queues.c	Message queues implementation.
msg_queues.h	Message queues API.
osal.h	RTOS-agnostic API wrappers.
resmgmt.c	Resource management implementation.
resmgmt.h	Resource management API.

6.1.7 RTOS-agnostic API

OSAL provides wrappers of the FreeRTOS API for the following RTOS primitives, presented in Table 6.

Table 6: OSAL wrappers of the FreeRTOS API

Description	API
Tasks/Threads	`OS_TASK_*`
Mutexes	`OS_MUTEX_*`
Events/Notifications	`OS_EVENT_*`
Thread-safe queues	`OS_QUEUE_*`
Memory allocation	`OS_MALLOC_*`
Memory de-allocation	`OS_FREE_*`
Timers	`OS_TIMER_*`

Note 5 The OSAL API does not yet completely cover the FreeRTOS API being used in the SmartSnippets™ DA1468x SDK.

6.1.8 Resource Management API

The OSAL resource management API is:

Table 7: OSAL resource management API

Function	Description
`resource_init()`	Initialize resource management structures.
`resource_acquire(mask, timeout)`	Attempt to acquire exclusive access to a set of resources.

Function	Description
`resource_release(mask)`	Release exclusive access to a set of resources.
`resource_add()`	Add new resources at run-time (compile-time configurable).

The OSAL resource management capability is used by the adapters to provide exclusive access from hardware resources. The low-level drivers layer are bare-metal drivers, so they do not have the notions of resource locking, synchronization etc. The adapter layer is RTOS-aware, and can provide such capabilities.

6.1.9 Message Queues API

OSAL message queues can be used to exchange messages between tasks. Although they build on top of the queue primitive, they add the capability to define the message allocation/de-allocation functions per message queue or even per message.

The available APIs are:

Table 8: OSAL message queues functions

Functions
`msg_queue_create(queue, size, allocator)`
`msg_queue_delete(queue)`
`msg_queue_put(queue, msg, timeout)`
`msg_queue_get(queue, msg, timeout)`
`msg_init(msg, id, type, buf, size, free_cb)`
`msg_release(msg)`
`msg_queue_init_msg(queue, msg, id, type, size)`
`msg_queue_send(queue, id, type, buf, size, timeout)`
`msq_queue_send_zero_copy(queue, id, type, buf, size, timeout, free_cb)`

The exact prototypes of these functions with types of arguments and returned values can be found in the source code or in Doxygen.

Usage of message queues is demonstrated in the `peripherals_demo/uart` demo.

7 The BLE Framework

Figure 7 depicts the general architectural scheme used.

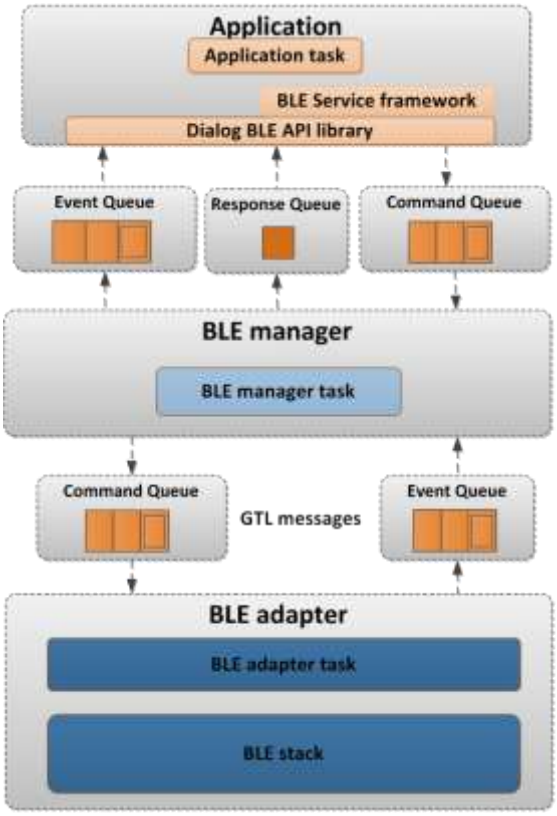

Figure 7: BLE framework architecture

Using a top-down approach, the layers that build-up the BLE framework functionality can be identified as the following:

1. The BLE service framework which is a library that provides implemented BLE services that the application can use “out-of-the-box”, using simple initialization functions and defining callbacks for the various BLE service events (like the change of an alert level attribute). The functionality of the BLE service framework is built on top of the Dialog BLE API library. The BLE service API header files can be found under `<sdk_root_directory>/sdk/interfaces/ble_services/include`.

   The BLE service framework is called in the context of the Application.
2. The Dialog BLE API is a set of functions that can be used to initiate BLE operations or respond to BLE events. The API header files can be found under the path `<sdk_root_directory>/sdk/interfaces/ble/include`. The API functions constitute a library that can be used to send messages (commands or replies to events) to the BLE manager, using queues between the application task and the BLE manager task which makes the application thread-safe. The BLE API is called in the context of the Application.
3. The BLE manager provides the interface to the BLE functionality of the chip. Application tasks that are based on BLE functionality use the Dialog BLE API to interface with the BLE manager. The BLE manager is a task that stands between the application and the BLE adapter. It uses the BLE adapter to interface with the BLE stack. The BLE manager uses a Generic Transport Layer (GTL) to communicate with the BLE adapter through a command and event queue.

   The BLE Manager runs in its own task.

4. The BLE adapter is the system task that provides the interface to the BLE stack which is in the ROM code. It runs the BLE stack internal scheduler, handles the BLE interrupts, receives the commands or the replies to events from the BLE manager, and passes BLE events to the BLE manager. BLE core functionality is implemented by the BLE adapter task.
5. The BLE stack is the software stack that interfaces with the BLE IP and implements the Link Layer and the host stack, specifically the Logical Link Control and Adaptation Protocol (L2CAP), the Security Manager Protocol (SMP), the Attribute Protocol (ATT), the Generic Attribute Profile (GATT) and the Generic Access Profile (GAP). The BLE stack software is stored in the system's ROM and its API header files can be found under `<sdk_root_directory>/sdk/interfaces/ble/src/stack`.

   The BLE stack default configuration can be modified by editing `<sdk_root_directory>/sdk/interfaces/ble/src/stack/config/ble_stack_config.h` although it is recommended to do this via project specific configuration files as described in Section 13.1. The BLE stack software is run in the context of the BLE adapter task which instantiates and initializes the BLE stack.

7.1 Developing BLE Applications

One of the main goals of the SmartSnippets™ DA1468x SDK is to simplify the development of Bluetooth low energy applications and achieve a fast time to market. The SmartSnippets™ DA1468x SDK separates the application logic from the BLE stack implementation and provide a clean and powerful API to interact with the Bluetooth low energy capabilities of the device. The BLE framework API gives easy access to configure the BLE manager, start air operations and set up an attribute database inside a GATT server. The BLE service API provides access to predefined Bluetooth SIG profiles with the definition of only a few call-back functions.

The Proximity Reporter (`pxp_reporter`) application described in Software Developer's Guide [3] is the most typical of the BLE applications that are included in the SmartSnippets™ DA1468x SDK. It is a complete and solid example of a BLE application developed on top of the SmartSnippets™ DA1468x SDK. It uses both the Dialog BLE API and the BLE service framework to implement the functionality of a Bluetooth low energy profile.

However, it may not be the simplest example or the best starting point to become familiar with the development of a Bluetooth low energy application from scratch. Instead, there are Bluetooth low energy projects specifically created to serve as starting points for specific Bluetooth low energy applications such as beacons (`ble_adv_demo`) or for specific roles such as a generic peripheral (`ble_peripheral`) or central (`ble_central`) device .

This section aims to introduce the various options and examples that exist in the SmartSnippets™ DA1468x SDK which can be used as building blocks for many applications. After a short introduction on where the API header files can be found, each section describes the functionality they implement along with guidance on how they differ from each other. This information is essential when starting a Bluetooth low energy application from scratch.

7.2 The BLE API header files

All demos and services API are found in the doxygen documentation. This can be found either at `<sdk_root_directory>/docs/html/index.html` or via the **Open** API Documentation button on the welcome page of SmartSnippets™ Studio.

7.2.1 Dialog BLE API

The Dialog BLE API header files are in `<sdk_root_directory>/sdk/interfaces/ble/include`.

In most projects these API header files are symbolically linked to

`<sdk_root_directory>/sdk/ble/include`.

The API functions are declared across several header files depending on their functionality:

- Common API (`ble_common.h`): Functions used for operations, not specific to a specific BLE host software component. For example:

Table 9: API Functions of the common BLE host software component

Function	Description
`ble_register_app()`	Register the application to the BLE framework so that it can receive BLE event notifications.
`ble_enable()`	Enable the BLE framework.
`ble_reset()`	Reset the BLE framework.
`ble_central_start()`	Start the device as a BLE central. This is actually a helper function, since it uses API calls ble_enable() and ble_gap_role_set().
`ble_peripheral_start()`	Start the device as a BLE peripheral. This is also a helper function that uses ble_enable() and ble_gap_role_set().
`ble_get_event()`	Get a BLE event from the BLE manager's event queue.
`ble_has_event()`	Check if there is an event pending at the BLE manager's event queue.
`ble_handle_event_default()`	Used to define handling of events that are not handled by the added services or the application defined handlers.

- GAP & L2CAP APIs (*ble_gap.h/ble_l2cap.h*): Covers a wide range of operations, like
  - Device parameters configuration: device role, MTU size, device name exposed in the GAP service attribute, etc.
  - Air operations: Advertise, scan, connect, respond to connection requests, initiate or respond to connection parameters update, etc.
  - Security operations: Initiate and respond to a pairing or bonding procedure, set the security level, unpair, etc.

Table 10: GAP and L2CAP API functions

Function	Description
BLE device configuration	
`ble_gap_role_set()`	Set the GAP role.
`ble_gap_mtu_size_get()`	Get the MTU size currently set.
`ble_gap_mtu_size_set()`	Set the MTU size.
`ble_gap_channel_map_get()`	Get the currently set channel map of the device (the device must be configured as central).
`ble_gap_channel_map_set()`	Set the channel map of the device (device must be configured as central).
`ble_gap_address_get()`	Get the currently set BD address of the device.
`ble_gap_address_set()`	Set the BD address of the device.

Function	Description
ble_gap_device_name_get()	Get the device name used for the GAP service.
ble_gap_device_name_set()	Set the device name used for the GAP service.
ble_gap_appearance_get()	Get the appearance used for the GAP service.
ble_gap_appearance_set()	Set the appearance used for the GAP service.
ble_gap_per_pref_conn_params_get()	Get the peripheral preferred connection parameters used for the GAP service.
ble_gap_per_pref_conn_params_set()	Set the peripheral preferred connection parameters used for the GAP service.
ble_gap_set_io_cap()	Set the I/O capabilities of the device (combined with the peer's I/O capabilities, this will determine which pairing algorithm will be used).
ble_gap_data_length_set()	Set the data length to be used for TX on new connections.
Advertising	
ble_gap_adv_start()	Start advertising.
ble_gap_adv_stop()	Stop advertising.
ble_gap_adv_data_set()	Set the Advertising Data and Scan Response Data used.
ble_gap_adv_intv_set()	Set the advertising intervals prior to advertising start.
ble_gap_adv_chnl_map_set()	Set the advertising channel map prior to advertising start.
ble_gap_adv_mode_set()	Set the discoverability mode used for advertising prior to advertising start.
ble_gap_adv_direct_address_set()	Set the peer address used for directed advertising prior to advertising start.
Scanning	
ble_gap_scan_start()	Start scanning for devices.
ble_gap_scan_stop()	Stop scanning for devices.

Function	Description
Connection management	
ble_gap_scan_params_get()	Get the scan parameters used for connections.
ble_gap_scan_params_set()	Set the scan parameters used for connections.
ble_gap_connect()	Initiate a direct connection to a device.
ble_gap_connect_ce()	Initiate a direct connection with an app-defined minimum and maximum connection event length
ble_gap_connect_cancel()	Cancel an initiated connection procedure.
ble_gap_disconnect()	Initiate a disconnection procedure on an established link.
ble_gap_conn_rssi_get()	Retrieve the RSSI of a connection.
ble_gap_conn_param_update()	Initiate a connection parameter update.
ble_gap_conn_param_update_reply()	Reply to a connection parameter update request.
ble_gap_data_length_set()	Set the data length used for TX for a specified connection.
Security	
ble_gap_pair()	Start pairing.
ble_gap_pair_reply()	Respond to a pairing request.
ble_gap_passkey_reply()	Respond to a passkey request.
ble_gap_numeric_reply()	Respond to a numeric comparison request (Secure Connections only).
ble_gap_get_sec_level()	Get connection security level.
ble_gap_set_sec_level()	Set connection security level.
ble_gap_unpair()	Unpair device (will also remove bond data from BLE storage).
Helper functions	
ble_gap_get_connected()	Get list of connected devices.
ble_gap_get_bonded()	Get list of bonded devices.
ble_gap_get_devices()	Return list of known devices based on

Function	Description
	filter.
ble_gap_get_device_by_addr()	Get the device object, given the device address.
ble_gap_get_device_by_conn_idx()	Get device object, given the connection index.
ble_gap_is_bonded()	Get bond state of device (by connection index).
ble_gap_is_addr_bonded()	Get bond state of device (by address).
Expert functions	
ble_gap_skip_latency()	Temporarily ignore the connection latency.

- GATT server API (ble_gatts.h): Set up the attribute database, set attribute values, notify/indicate characteristic values, initiate MTU exchanges, respond to write and read requests, etc.

Table 11: GATT server API

Function	Description
ble_gatts_add_service()	Add a new GATT service to the ATT database. Subsequent calls to ble_gatts_add_include(), ble_gatts_add_characteristic() and ble_gatts_add_descriptor() will add attributes to the service added by this call.
ble_gatts_add_include()	Add an included service declaration to the service added by ble_gatts_add_service().
ble_gatts_add_characteristic()	Add a characteristic declaration to the service added by ble_gatts_add_service().
ble_gatts_add_descriptor()	Add a descriptor declaration to the service added by ble_gatts_add_service().
ble_gatts_register_service()	Add to the ATT database all attributes previously added to the service.
ble_gatts_enable_service()	Enable service in database.
ble_gatts_disable_service()	Disable service in database.
ble_gatts_get_characteristic_prop()	Read current characteristic properties and permissions.
ble_gatts_set_characteristic_prop()	Set characteristic properties and permissions.
ble_gatts_get_value()	Get attribute value.
ble_gatts_set_value()	Set attribute value.

Function	Description
ble_gatts_read_cfm()	Confirmation response to an attribute read request.
ble_gatts_write_cfm()	Confirmation response to an attribute write request.
ble_gatts_prepare_write_cfm()	Confirmation response to an attribute prepare write request.
ble_gatts_send_event()	Send a characteristic value notification or indication.
ble_gatts_service_changed_ind()	Send indication of the Service Changed Characteristic.
ble_gatts_get_num_attr()	Calculate the number of attributes required for a service.

- GATT client API (ble_gattc.h): Used by a device configured as a GATT client to discover the services, characteristics, etc. of a peer device, read or write its attributes, initiate MTU exchanges, confirm the reception of indications, etc.

Table 12: GATT client API

Function	Description
ble_gattc_browse()	Browse services on a remote GATT server.
ble_gattc_discover_svc()	Discover services on a remote GATT server.
ble_gattc_discover_include()	Discover included services on a remote GATT server.
ble_gattc_discover_char()	Discover characteristics on a remote GATT server.
ble_gattc_discover_desc()	Discover descriptors on a remote GATT server.
ble_gattc_read()	Read a characteristic value or a characteristic descriptor from the remote GATT server, depending on the attribute handle.
ble_gattc_write()	Write a characteristic value or a characteristic descriptor to the remote GATT server, depending on the attribute handle.
ble_gattc_write_no_resp()	Write attribute to remote GATT server without response.
ble_gattc_write_prepare()	Prepare long/reliable write to remote GATT server.
ble_gattc_write_execute()	Execute long/reliable write to remote GATT server.
ble_gattc_indication_cfm()	Send confirmation for received indication.
ble_gattc_get_mtu()	Get current TX MTU of peer.
ble_gattc_exchange_mtu()	Exchange MTU.

Note 6 That several GAP configuration functions must be called before the attribute database is created, because modifying the device's configuration can clear the attribute database created up to that point. This is noted in the Doxygen headers of the configuration functions that can have this effect.

7.2.2 Dialog BLE service API

The BLE service API header files are in `<sdk_root_directory>/sdk/interfaces/ble_services/include`.

In most projects the API header files are symbolically linked to `<sdk_root_directory>/sdk/ble_services/include`.

The services-specific API, call-back function prototypes and definitions are included in each service's header file. The services implemented are the following:

Table 13: Header files for the BLE services

Header file	Description
bas.h	Battery Service.
bcs.h	Body Composition Service.
ble_service	Services handling routines API.
bms.h	Bond Management Service.
cts.h	Current Time Service.
dis.h	Device Information Service.
dlg_debug	Debug service API.
dlg_suota	SUOTA service implementation API.
hids.h	Human Interface Device Service.
hrs.h	Heart Rate Service.
ias.h	Immediate Alert Service.
lls.h	Link Loss Service.
scps.h	Scan Parameter Service.
sps.h	Serial Port Service.
svc_defines	Common definitions for all services.
svc_types	Characteristics common types.
tps.h	Tx Power Service.
uds.h	User Data Service.
wss.h	Weight Scale Service.

All services have an initialization function defined. This function is called with arguments that vary for different services.

The most common argument is a pointer to one or more call-back functions that should be called upon a service-specific event. For example, the prototype for the initialization function of the Immediate Alert Service (`ias.h`) is the following:

```
ble_service_t *ias_init(ias_alert_level_cb_t alert_level_cb)
```

Code 6: Initialization code for Immediate Alert Service

Function `ias_init()` has only one argument. It is a pointer to the call-back function that will be called when a peer device has modified the value of the Immediate Alert Level characteristic. This callback function is part of the user application code and should provide the application handling required for the change to the Immediate Alert Level.

The return value from all initialization functions is the created service's handle which is used to reference the service in the application. So for example the handle will be used as an argument for function `ble_service_add()` to add the created service to the BLE service framework (in SDK release 1.0.10 and later this is seamlessly done by the service initialization function and there is no need to explicitly use `ble_service_add()`).

The application only needs to use the service handle. However, to understand how the service interacts with the BLE framework it is useful to know what the handle represents. The handle is a pointer to a generic structure (`ble_service_t`). that defines how the service should interact with the framework. Within each service there is an internal service definition (`XXX_service_t`) as shown in Figure 8. This contains the generic service structure plus a set of handles, one for each GATT characteristic that the service implements. This `XXX_service_t` structure is populated by `XXX_init()` for that service. The start_h and end_h handles will contain the start and end positions of the attributes for this service within the overall GATT table provided by the GATT server. So, when a GATT client requests a Service Discovery from the server these represent the start and end handles that the client would use to access service `XXX`.

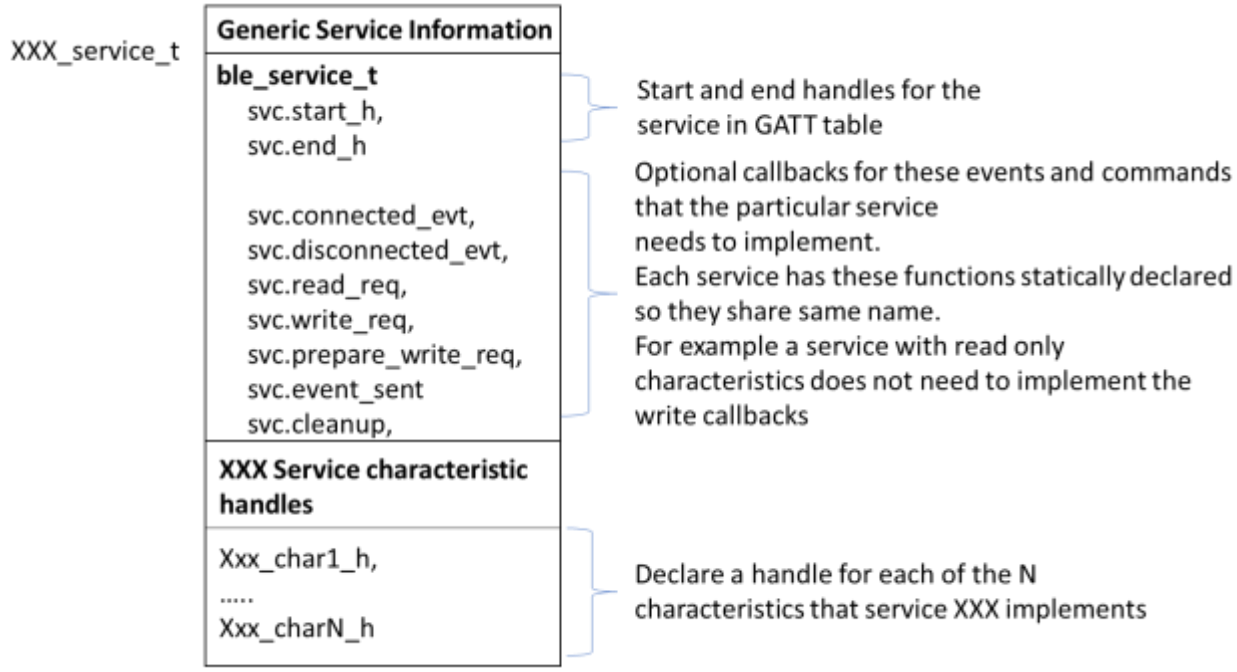

Figure 8: Structure of a service handle

The set of optional callbacks allow each service to specify if it wants to do some specific handling on a certain event received by the BLE framework. If the service wants to be informed when another Bluetooth low energy device has connected to this device then it can define its own `handle_connected_evt()` function and plug it into the `connected_evt` callback. Each service declares its `handle_connected_evt()` function as static in `xxx.c` and by convention in the SmartSnippets™ SDK they all share the same function names in each service.

As each service is initialized and thus added to the BLE services framework with `ble_service_add`(), its generic services structure is added to a structure of supported services as shown in Figure 9.

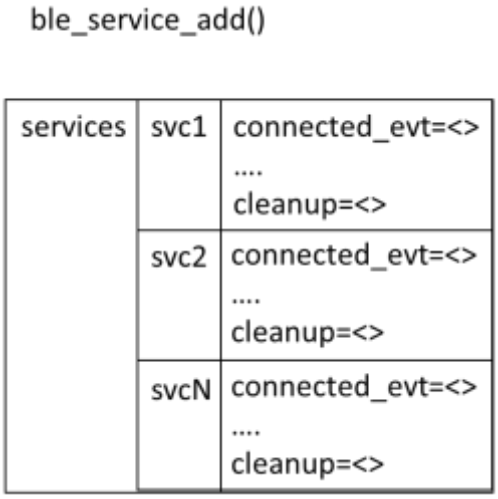

Figure 9: Structure of supported services

Now that the main internal services structure has been explained, it is easier to follow how the service initialization defines how the service operates.

Within the BLE service framework the main event handler is `ble_service_handle_event()` which is shown below.

```
bool ble_service_handle_event(const ble_evt_hdr_t *evt)
{
        switch (evt->evt_code) {
        case BLE_EVT_GAP_CONNECTED:
                connected_evt((const ble_evt_gap_connected_t *) evt);
                return false; // make it "not handled" so app can handle
        case BLE_EVT_GAP_DISCONNECTED:
                disconnected_evt((const ble_evt_gap_disconnected_t *) evt);
                return false; // make it "not handled" so app can handle
        case BLE_EVT_GATTS_READ_REQ:
                return read_req((const ble_evt_gatts_read_req_t *) evt);
        case BLE_EVT_GATTS_WRITE_REQ:
                return write_req((const ble_evt_gatts_write_req_t *) evt);
        case BLE_EVT_GATTS_PREPARE_WRITE_REQ:
                return prepare_write_req((const ble_evt_gatts_prepare_write_req_t *)
evt);
        case BLE_EVT_GATTS_EVENT_SENT:
                return event_sent((const ble_evt_gatts_event_sent_t *) evt);
        }
        return false;
}
```

Code 7: Handle BLE events using BLE service framework

Each of these sub-handlers inside `ble_service_handle_event()` search throughout the added services to find one that has defined a behavior for this event. There are two types of events that are handled differently

7.2.2.1 Connection Orientated Events

The connection and disconnection events are potentially of interest to all registered services and so all services can be informed. The cleanup on shutdown is handled in the same way.

For example a connection event will call the BLE service's statically defined `connected_evt()` function (`sdk/ble_services/src/ble_service.c`). This will loop through all the services registered in `services` array and if for each service a `connected_evt` callback has been registered during initialization the callback function will be called.

7.2.2.2 Attribute Orientated Events

These are events that are to do with a given attribute handle. As each attribute is related to a unique service the first step in one of these handlers is to identify which service the attribute belongs to.

For example a write request on a specific attribute will call the BLE service's statically defined `write_req()` function (`sdk/ble_services/src/ble_service.c`) as shown below. This will first identify which service owns that attribute with `find_service_by_handle()`. Then if it has a `write_req` callback defined it executes the callback.

```
static bool write_req(const ble_evt_gatts_write_req_t *evt)
{
        ble_service_t *svc = find_service_by_handle(evt->handle);
        if (!svc) {
                return false;
        }
        if (svc->write_req) {
                svc->write_req(svc, evt);
        }
        return true;
}
```

Code 8: Example of code for the Write Request

An example of this flow is the Write No Response procedure that can be applied to the Immediate Alert Level characteristic of the Immediate Alert Service. When a GATT client requests a write to that characteristic it will trigger the `write_req()` sub-handler under `ble_service_handle_event()`.

The `write_req()` sub-handler will use `find_service_by_handle()` to see if any of the added services are registered for that characteristic. It will match it with the Immediate Alert Service (IAS) and as the IAS has registered a Write Request handler the IAS `handle_write_req()` will be called (`<sdk_root_directory>\sdk\interfaces\ble_services\src\ias.c`).


```
static void handle_write_req(ble_service_t *svc, const
ble_evt_gatts_write_req_t *evt)
{
        ia_service_t *ias = (ia_service_t *) svc;
        att_error_t err = ATT_ERROR_OK;

        if (evt->length == 1) {
                uint8_t level;

                /*
                * This is write-only characteristic so we don't need to
store value written
                * anywhere, can just handle it here and reply.
                */

                level = get_u8(evt->value);

                if (level > 2) {
                        err = ATT_ERROR_APPLICATION_ERROR;
                } else if (ias->cb) {
                        ias->cb(evt->conn_idx, level);
                }
        }

        ble_gatts_write_cfm(evt->conn_idx, evt->handle, err);
}
```

Code 9: Example of code that handle the Write Request and match it with the appropriate instance

By calling `ias->cb()` function, this handler also actually calls the application supplied call-back function passed as an argument when `ias_init()` was called by the application. Finally, it sends a Write Confirmation to update the value at the attribute database maintained in the BLE stack.

This is only an example of the way the BLE service framework translates BLE events to BLE service events. Different events in different services can have different levels of complexity, but most of the times this complexity is contained within the BLE service API. The aim is that the application only needs to call the service's initialization function and define the appropriate call-back functions to handle all service's events.

In addition, some services define additional service-specific API calls. For instance, the Heart Rate Service implementation defines an API to trigger notifications of the Heart Rate Measurement characteristic, using functions `hrs_notify_measurement()` and `hrs_notify_measurement_all()` (the first is used to notify the characteristic to a specified peer, while the second is used to notify the characteristic to all subscribed peers). Some services also define some internal helper functions that are used to manipulate characteristic values, and some services require attribute initial values as arguments of the initialization function.

The BLE service API adds another layer to the general BLE API. Together the BLE adapter, BLE manager, BLE API library and BLE service framework results in the BLE framework.

The BLE services API provides an off the shelf solution to implement an application using many of the common adopted Bluetooth low energy services. The underlying BLE API and GATT server API can be used to create other adopted services or even custom services using the BLE services as a template.

The following sections will provide an overview of a generic application and then will describe in detail several of the example Bluetooth low energy projects included in the SmartSnippets™ DA1468x SDK:

Table 14: BLE projects included in the SmartSnippets™ DA1468x SDK

BLE projects	General description
ble_adv_demo	The simplest BLE project available in the SmartSnippets DA1468x SDK, which does not use the BLE service framework and exposes only the GAP and GATT services.
ble_peripheral	A project that can be used as a template for developing BLE peripheral applications. The project includes some of the services implemented under the BLE service framework.
ble_central	A project that can be used as a template for developing BLE central applications. The project starts by scanning and trying to connect to a device with a pre-defined BD address, and then discovers all services and characteristics of it.
ble_multi_link_demo	This project demonstrates the Bluetooth LE Topology feature, using a custom service that can be used to “force” a device to be master on one connection and slave on another at the same time.
ble_external_host	This project exposes an HCI controller interface on a UART. The BLE framework is bypassed in this case as the external host provides the Bluetooth stack.
ble_suota_client	This application is a SUOTA 1.2 client implementation and allows to update SUOTA-enabled devices over the air, using a simple serial console interface.
power_demo	This project is a simple connectable advertising demo. It is designed to let the user configure several parameters such as the advertising interval and the connection parameters using either UART commands or the GPIOs. User can select the various preconfigured settings and examine the effect on the power consumption that these settings have.

7.2.3 Configuring the project

In each project the BLE framework and BSP are configured via a set of custom config files that set the defines and macros used in that project. These files are found in the `config` directory of each project.

In the case of the `ble_adv_demo` project this file is `config/custom_config_qspi.h`

Any defines set in this file will override the default SDK values which are found in the following files

```
sdk/config/bsp_defaults.h
sdk/ble/config/ble_config.h
sdk/bsp/free_rtos/include/FreeRTOSconfig.h
```

7.2.4 BLE application structure

All the Bluetooth low energy application projects in the SmartSnippets™ DA1468x SDK have a similar structure. This involves several FreeRTOS tasks at different priority levels (illustrated in Figure 6) to ensure that the overall systems real time requirements are met.

The Bluetooth low energy application is implemented in a FreeRTOS task that is created by the `system_init()` function. `system_init()` runs at the highest priority at startup and is also responsible for starting the BLE manager and BLE adapter tasks that run the Bluetooth low energy stack.

The application task has the following flow:

1. Device initialization and configuration: Start-up BLE, setting device role, device name, appearance and other device parameters.

1. Attribute database creation (GATT server devices): Creation of services and attributes using the BLE service framework. This must be done after (1) to prevent deletion of the attribute database.
2. Air operation configuration and initiation: Bluetooth low energy peripheral devices usually end-up advertising and Bluetooth low energy central devices end-up scanning and/or attempting to connect to another device. This is the last operation to be done only once as the task then drops into its infinite loop.
3. The infinite `for(;;)` loop, which is the application's event handling loop. The application has been set-up as desired and now it is waiting for Bluetooth low energy events to respond to, like connections or write requests.
   - The BLE adapter (ad_ble) must have a higher priority than the application task(s) because it runs the lower stack scheduler. It handles time critical tasks, like applying connection parameter and channel map updates, and replying to Packet Data Units (PDU) from the peer on time. If the BLE adapter does not run in time because another task has a higher priority, BLE events and even connections may be lost.
   - Most of the Bluetooth low energy applications use the FreeRTOS task notifications mechanism to block. `ble_adv_demo` is the simplest application and is the only project that does not use this mechanism. Instead, it just blocks on the BLE manager's event queue.
   - In addition to the BLE-related functionality most projects also use other system modules, like software timers or sensors. In this case, the application usually defines call-back functions to be triggered on these system events or interrupts. These callback functions should use task notifications to unblock the application task which can then handle the event or interrupt in the context of the task's `for(;;)` loop. The reason for this is that the BLE framework must be accessed by only one application task.

   Calling a BLE API function inside a call-back function triggered on a timer expiry will execute the BLE API function in the timer's task context. Calling other functions in the callback functions also can have implications on real time performance or in corrupting the small stack used by the timer task.

7.3 Bluetooth low energy Security

The Bluetooth specification defines the security options for device authentication and link encryption. These aspects of security are handled seamlessly by the BLE Framework. The API in Table 15 is able to set-up security, initiate pairing, do a security request or set-up encryption using previously exchanged keys. Most details of the procedures will be handled internally by the BLE Framework and the application will be notified only if intervention is needed or when the procedure is completed. These options will be described in detail in sections 7.3.1 and 7.3.4.

The generation and storage of the security keys and other bonding information is also handled by the BLE Framework. Persistent storage can also be used to enable storage of the security keys and bonding data info in the flash. The BLE Framework can then retrieve this information after a power cycle and use it to restore connections with previously bonded devices. This is described in 7.3.5.

7.3.1 Functions

Table 15 summarizes the API functions that can be used by the application to set-up security features.

Table 15: BLE Security API functions

API call	Description
`ble_gap_pair()`	Initiate a pairing or bonding procedure with a connected peer. Depending on whether the device is master or slave on the connection, this call will result either in a pairing or a security request respectively.
`ble_gap_pair_reply()`	Reply to a received `BLE_EVT_GAP_PAIR_REQ` event. This event will only be received by a peripheral device when

API call	Description
	the central peer has initiated a pairing procedure, so this function should only be called by a peripheral application and only after a BLE_EVT_GAP_PAIR_REQ event has been received.
ble_gap_passkey_reply()	Reply to a received BLE_EVT_GAP_PASSKEY_REQUEST event. This event will be received if the combination of the devices' input/output capabilities results in a passkey entry pairing algorithm. The application should use this function to submit the passkey for the pairing procedure to proceed.
ble_gap_numeric_reply()	Reply to a received BLE_EVT_GAP_NUMERIC_REQUEST event. This event will be received if the combination of the devices' input/output capabilities results in a numeric comparison pairing algorithm. The application should use this function to accept or reject the numeric key for the pairing procedure to proceed. This should be only used if LE Secure Connections are enabled.
ble_gap_set_sec_level()	Set the security level for a connection. If the device is already bonded, the existing Long Term Key (LTK) will be used to set-up encryption. If the device is not bonded, a pairing or a security request will be triggered (depending on whether the device is master or slave on the connection) with the bond flag set to false.
ble_gap_get_sec_level()	Get the security level currently established on a connection.
ble_gap_unpair()	Unpair a previously paired or bonded device. This will also remove security keys and bonding data info currently present in BLE storage.

7.3.2 Events

Table 16 describes the BLE events related to security that may be received by the application and the proper API functions to respond to them.

Table 16: BLE Security API events

Event	Argument	Description
BLE_EVT_GAP_PAIR_REQ	ble_evt_gap_pair_req_t	Pairing request received by a connected peer. Member <bond> indicates if the peer has requested a bond (that is, exchange of long term security keys). The application should use ble_gap_pair_reply() to respond to this request.
BLE_EVT_GAP_PAIR_COMPLETED	ble_evt_gap_pair_completed_t	A previously requested pairing procedure has been completed. Member <status> indicates the completion status of the procedure, while members <bond> and <MITM> indicate if a bond was established with the peer and if MITM (Man In The Middle) protection has been enabled on the connected link.
BLE_EVT_GAP_SECURITY_REQUEST	ble_evt_gap_security_request_t	Security request received by a connected peripheral peer. Members <bond> and

Event	Argument	Description
		<MITM> indicate if a bond and MITM protection have been requested by the peer. The application may use ble_gap_pair() to initiate pairing with the peer.
BLE_EVT_GAP_PASSKEY_NOTIFY	ble_evt_gap_passkey_notify_t	A passkey has been generated during a pairing procedure. This event will be received if the application has display capability. Member <passkey> contains the passkey that should be displayed to the user and entered by the peer for the pairing procedure to continue.
BLE_EVT_GAP_PASSKEY_REQUEST	ble_evt_gap_passkey_request_t	A passkey has been requested during a pairing procedure. This event will be received if the application has keyboard capability. The application should use ble_gap_passkey_reply() to respond to this request using the passkey entered by the user.
BLE_EVT_GAP_NUMERIC_REQUEST	ble_evt_gap_numeric_request_t	A numeric comparison has been requested during a pairing procedure. This event will be received if the application has keyboard or Yes/No and display capability. The application should use ble_gap_numeric_reply() to respond to this request using the accept or reject input entered by the user.
BLE_EVT_GAP_SEC_LEVEL_CHANGED	ble_evt_gap_sec_level_changed_t	The security level has been changed on an established link. Member <level> contains the security level that has been reached. This will be received after a pairing or an encryption procedure has been successfully completed.
BLE_EVT_GAP_SET_SEC_LEVEL_FAILED	ble_evt_gap_set_sec_level_failed_t	Setting the security level on an established link using ble_gap_set_sec_level() has failed. Member <status> indicates the reason for the failure. This will be received after an initiated encryption procedure has been unsuccessful. This may indicate that pairing should be requested again for the connected peer (for example, the peer may have lost the previously exchanged security keys).

7.3.3 Macros

Table 17 contains the configuration macros related to BLE security.

Table 17: BLE Security API macros

Macro	Default	Description
dg_configBLE_SECURE_CONNECTIONS	1	Set to 1 to use LE Secure Connections pairing if the peer supports the feature or to 0 to always use LE Legacy Pairing.

Macro	Default	Description
dg_configBLE_PAIR_INIT_KEY_DIST	GAP_KDIST_ENCKEY | GAP_KDIST_IDKEY | GAP_KDIST_SIGNKEY	Set the security keys requested to be distributed by the pairing initiator during a pairing feature exchange procedure.
dg_configBLE_PAIR_RESP_KEY_DIST	GAP_KDIST_ENCKEY | GAP_KDIST_IDKEY | GAP_KDIST_SIGNKEY	Set the security keys requested to be distributed by the pairing responder during a pairing feature exchange procedure.

7.3.4 Message Sequence Charts (MSCs)

7.3.4.1 Central

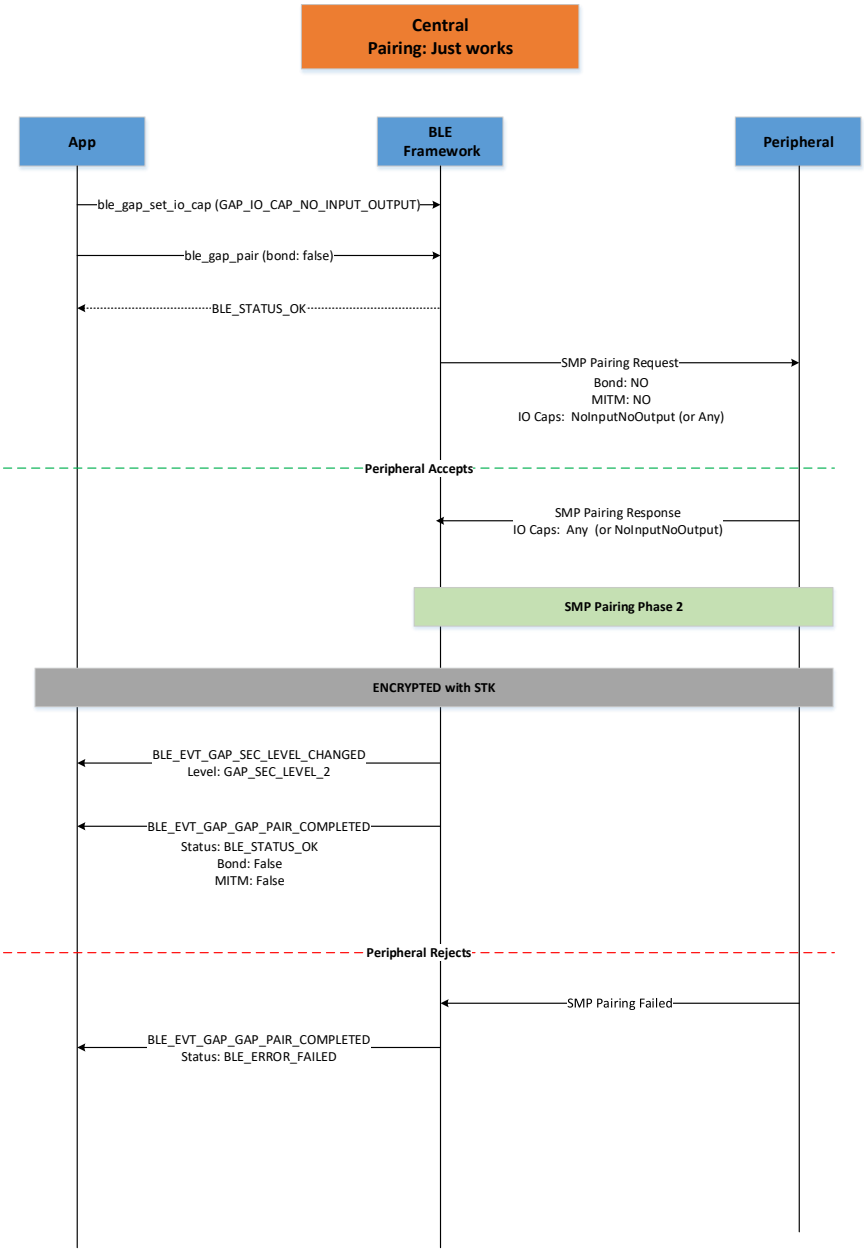

Figure 10: Pairing Just Works

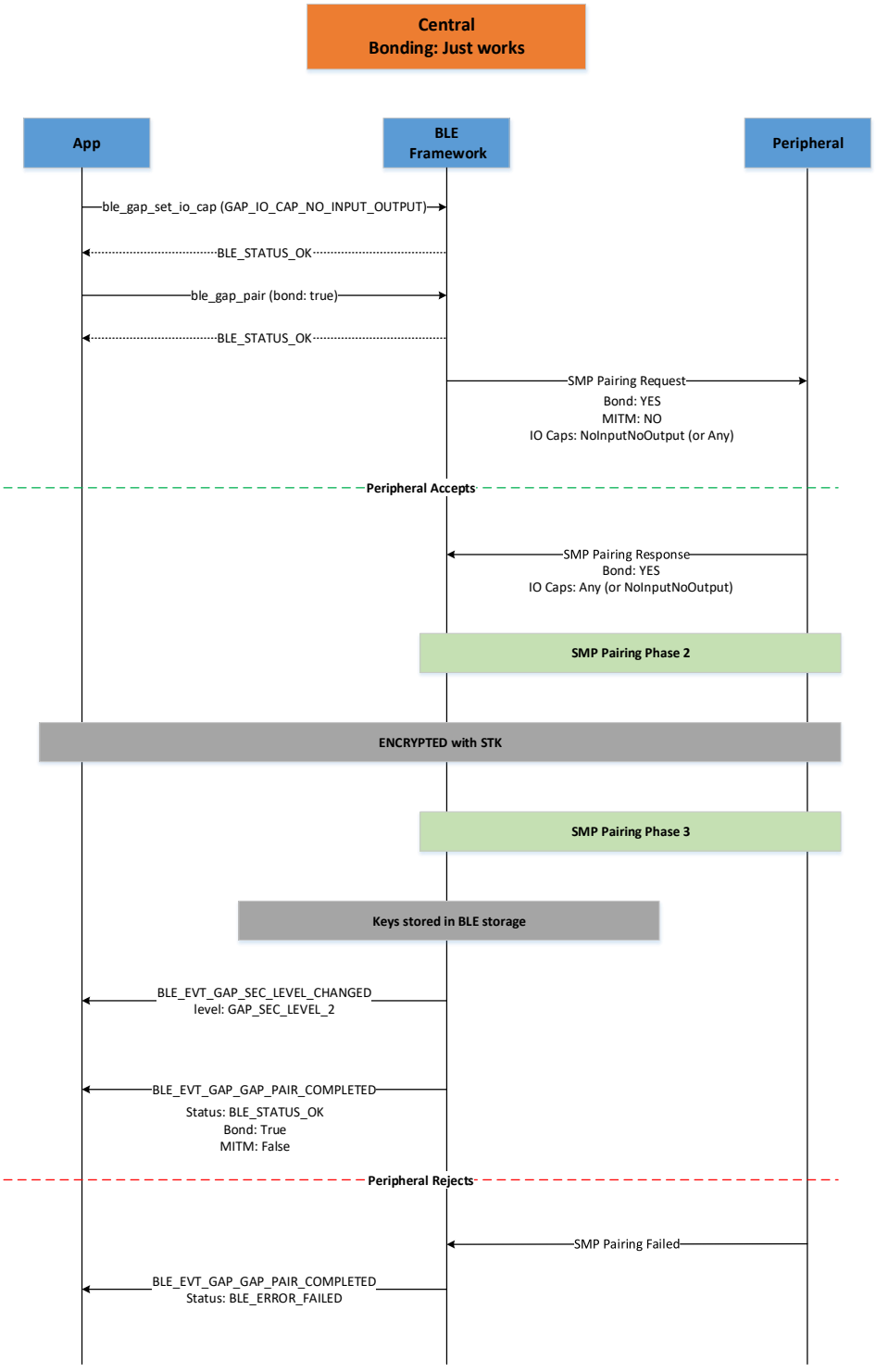

Figure 11: Bonding Just Works

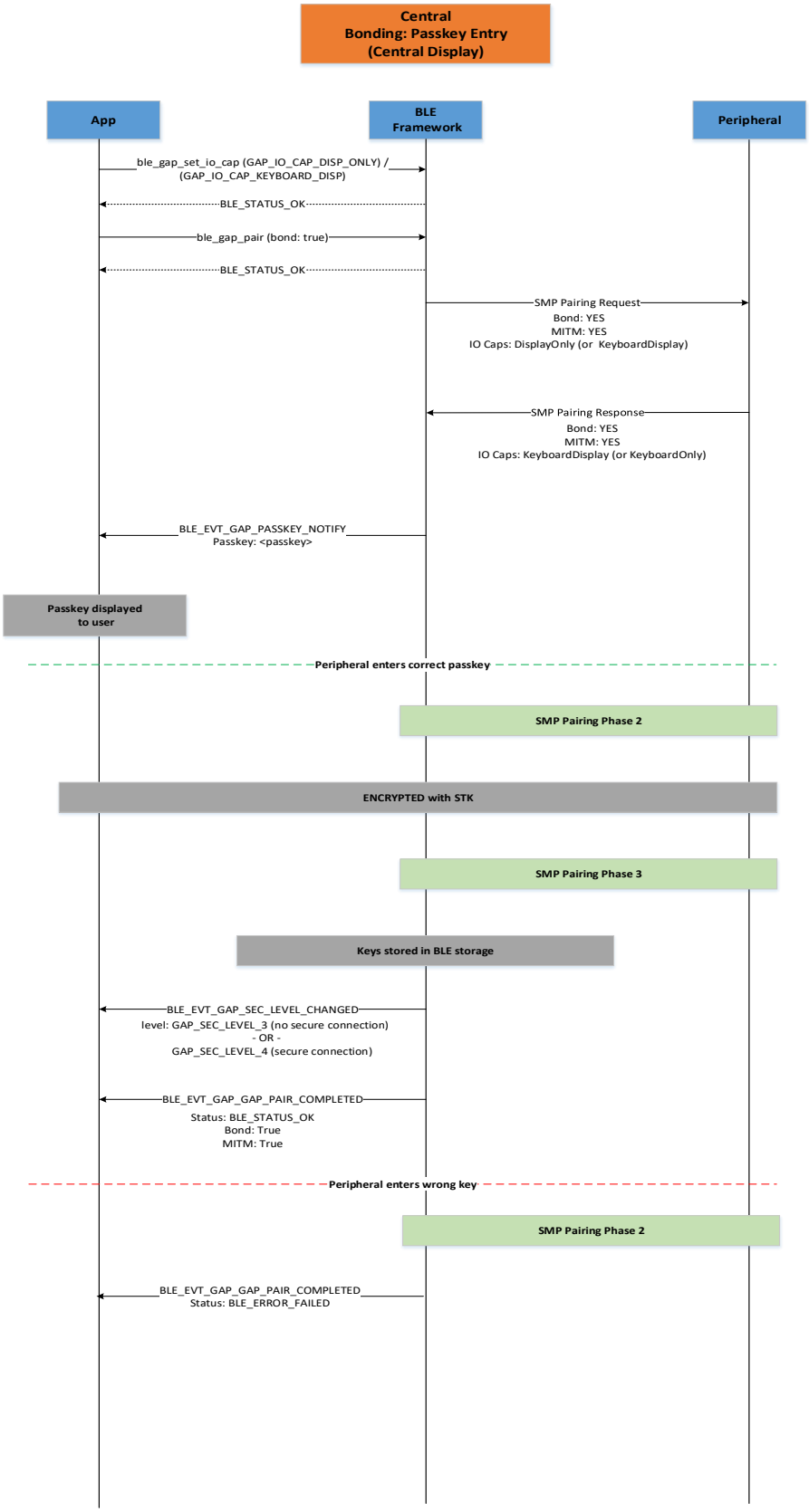

Figure 12: Bonding Passkey Entry (Central Display)

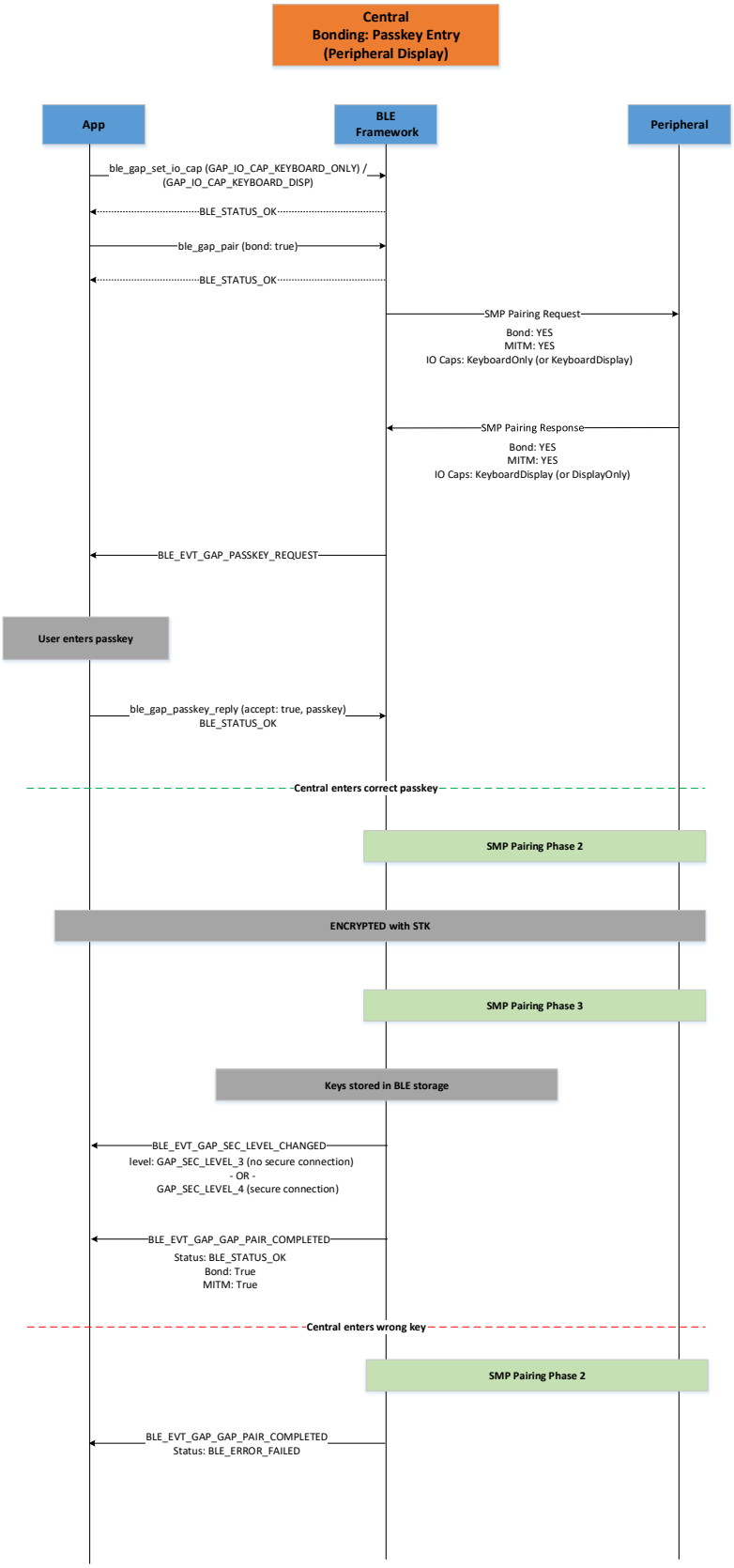

Figure 13: Bonding Passkey Entry (Peripheral Display)

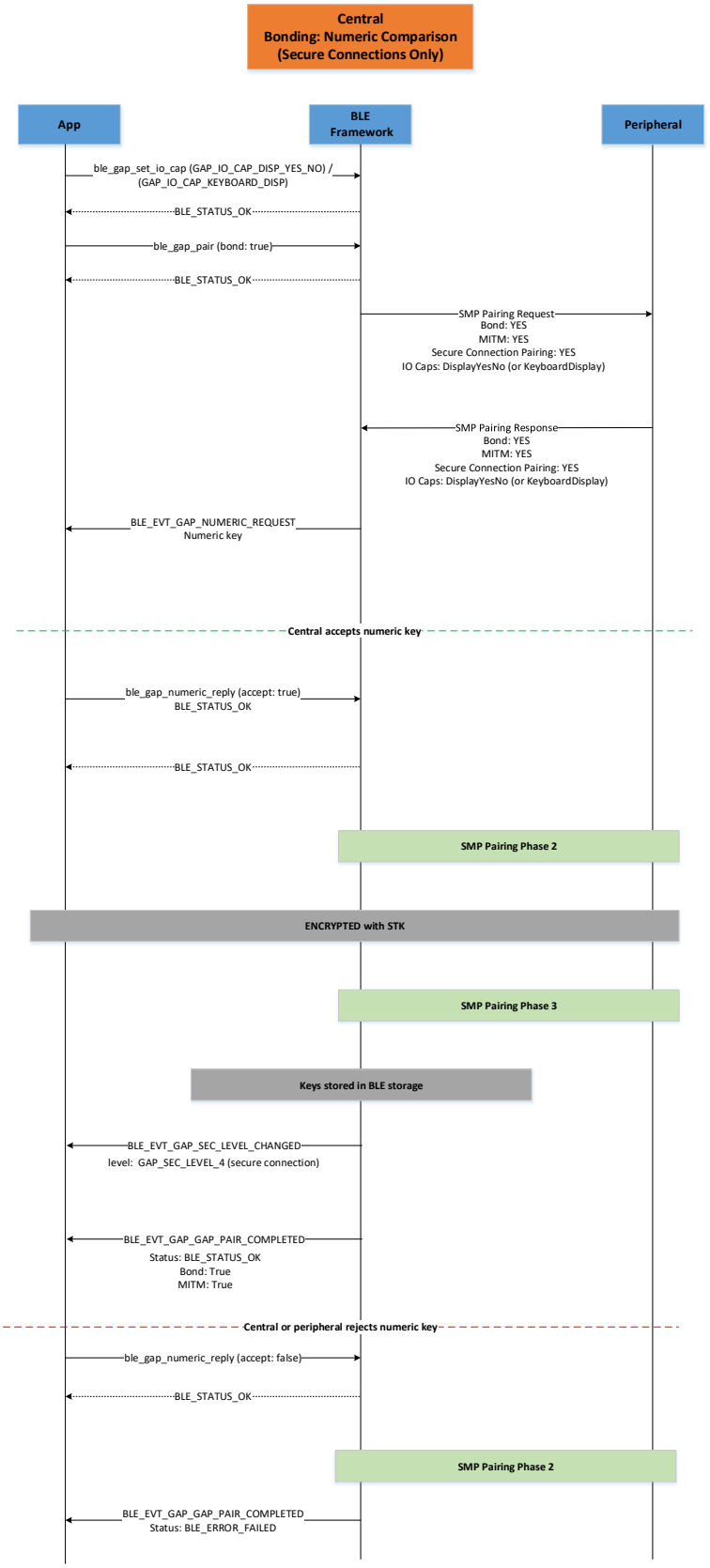

Figure 14: Bonding Numeric Comparison (Secure Connections Only)

7.3.4.2 Peripheral

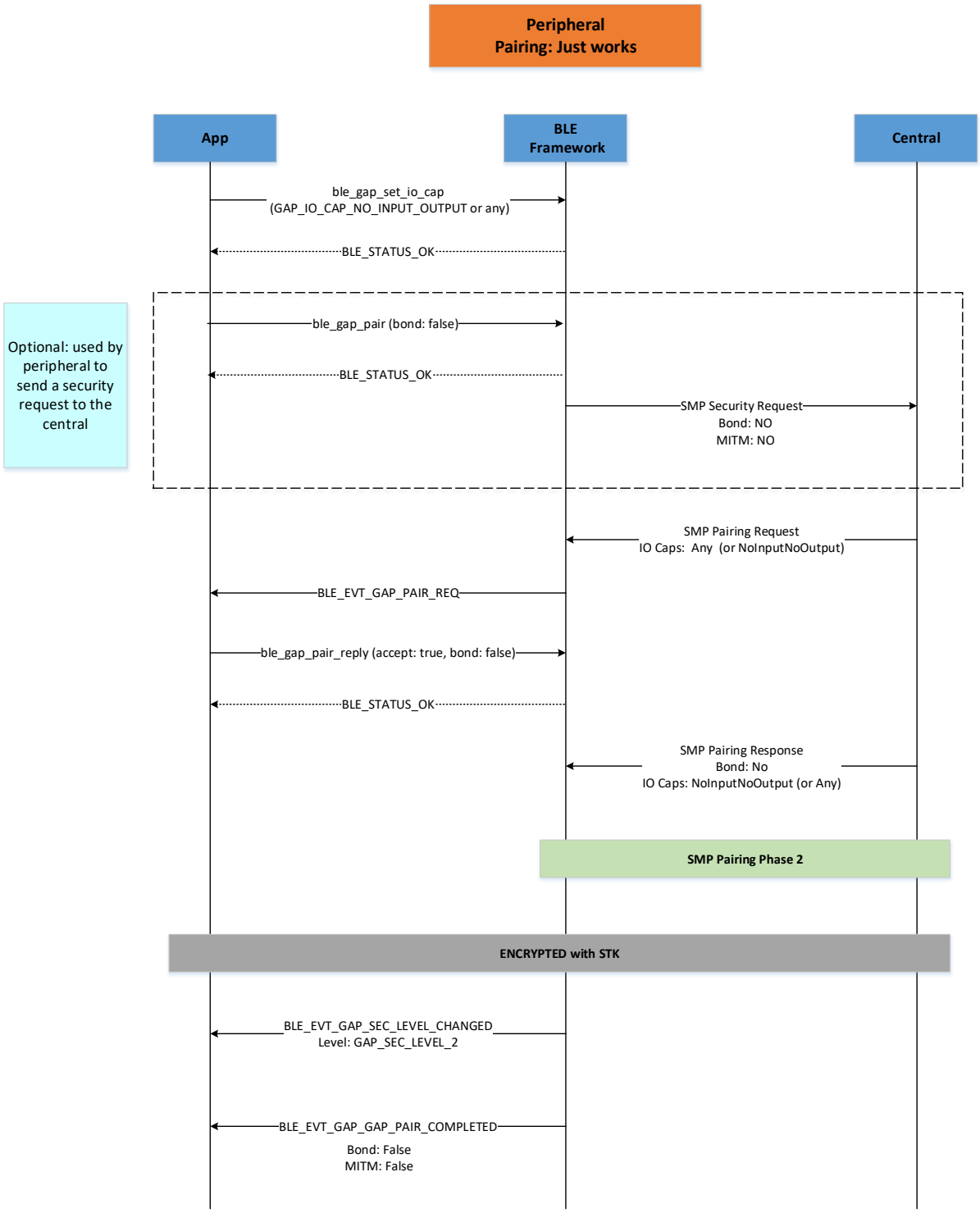

Figure 15: Pairing Just Works

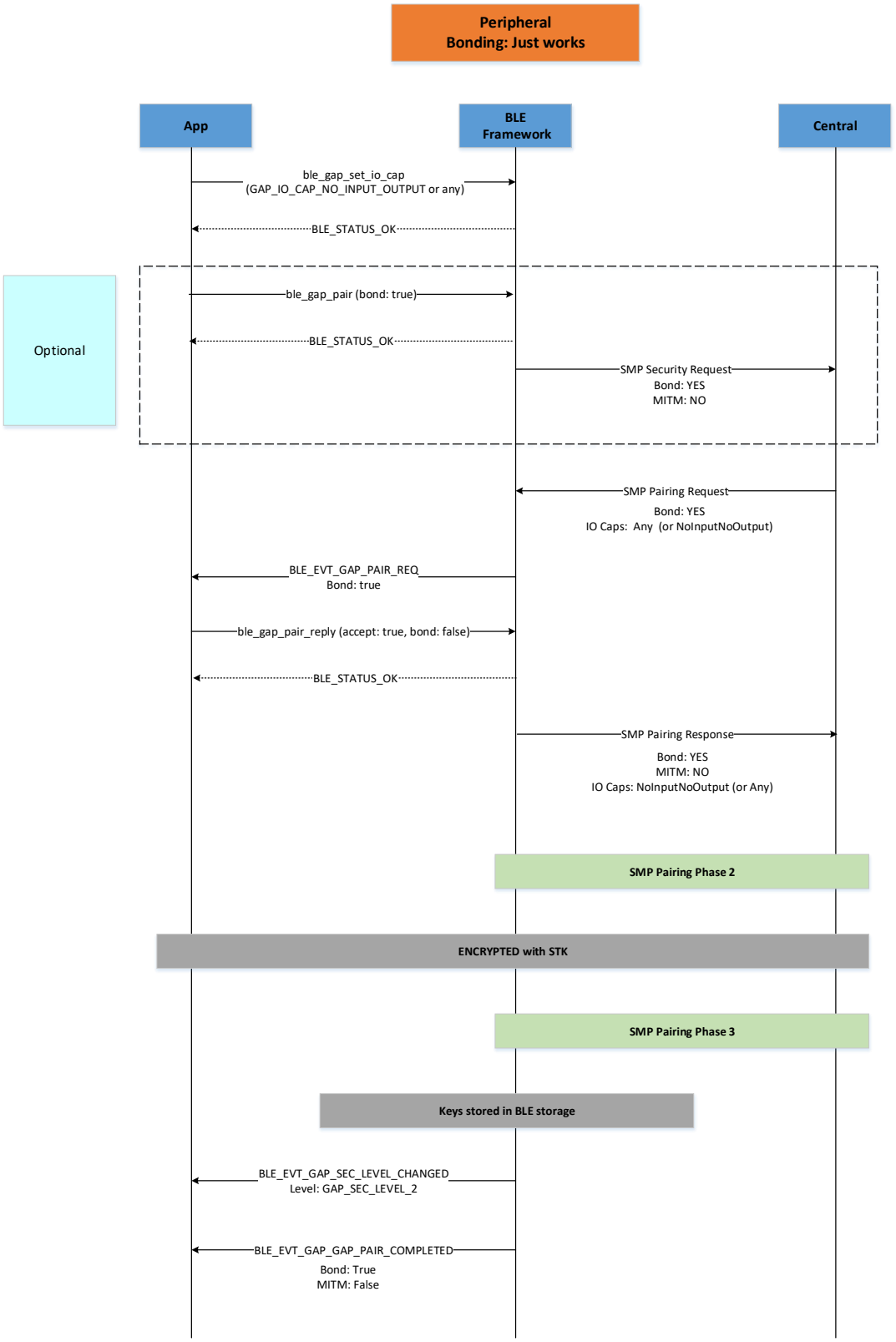

Figure 16: Bonding Just Works

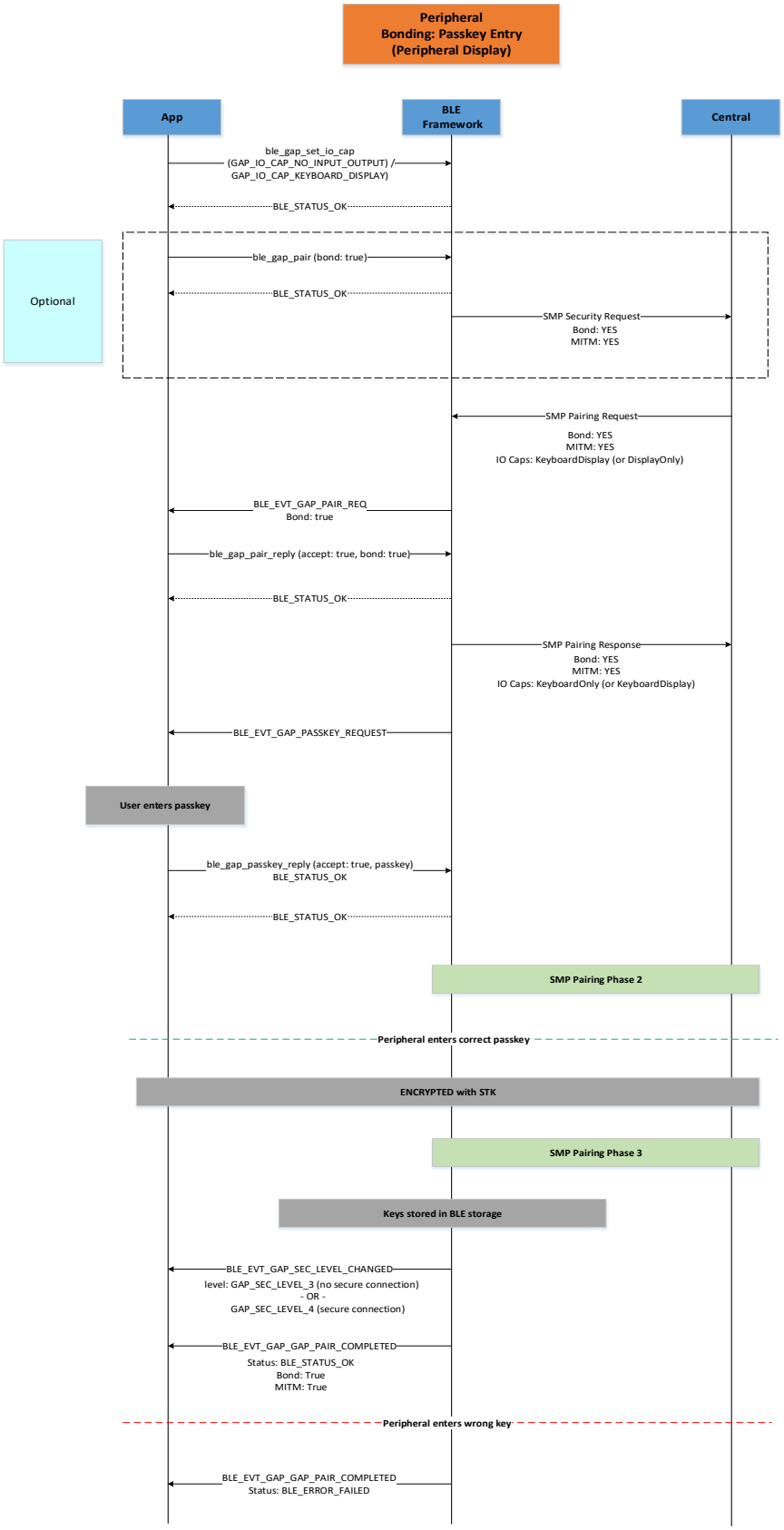

Figure 17: Bonding Passkey Entry (Peripheral Display)

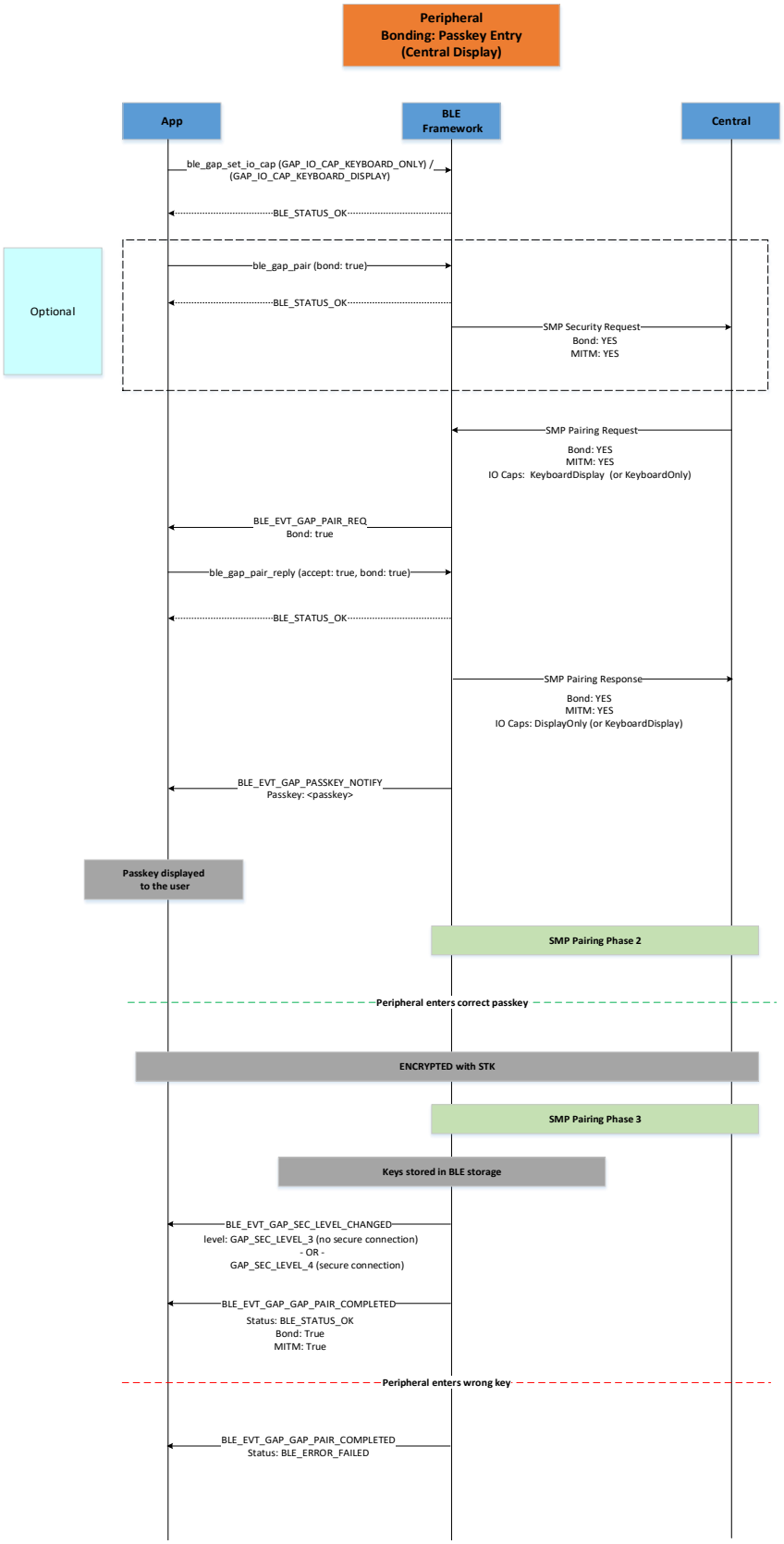

Figure 18: Bonding Passkey Entry (Central Display)

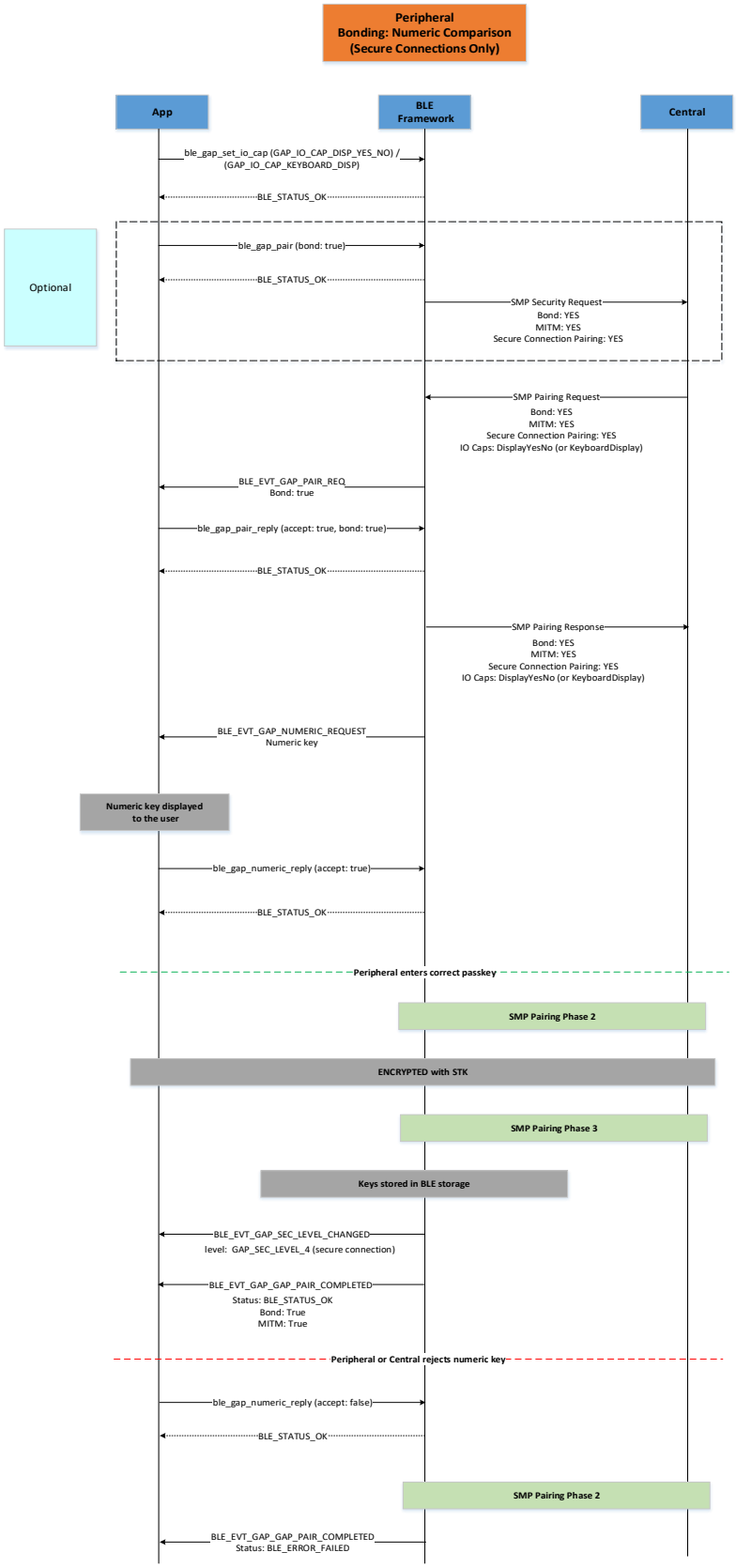

Figure 19: Bonding Numeric Comparison (Secure Connections Only)

7.3.5 BLE Storage

BLE Storage is the BLE Framework module that implements storage functionality for information related to connected and bonded peers, like security keys, CCC descriptors configuration and application-defined values. BLE Storage can manage the list of connected and bonded devices both in RAM and in persistent storage (for example, in the flash). By default, devices are managed in RAM only and persistent storage must be explicitly enabled in the application's configuration using macro `CONFIG_BLE_STORAGE`. Device data is then stored using Non-Volatile Memory Storage (NVMS) on the generic partition (see Section 12.4 for details).

Two kinds of data are stored:

- Device pairing data (exchanged keys and related information).
- Application-defined data managed using the BLE storage API (only the values with the 'persistent' flag set are stored in flash, for example CCC descriptor values).

Persistent storage can be enabled by the application by adding the following entries in the application's custom configuration file:

```
// enable BLE persistent storage
#define CONFIG_BLE_STORAGE

// enable Flash and NVMS adapters with VES (required by BLE persistent storage)
#define dg_configFLASH_ADAPTER                  1
#define dg_configNVMS_ADAPTER                   1
#define dg_configNVMS_VES                             1
```

The maximum number of bonded devices can be set using the `defaultBLE_MAX_BONDED` macro (defined to 8 by default). If the application attempts to bond more devices than its allowed, an error will be returned. This error should be handled by the application. It can then either unpair one of the currently bonded devices (using `ble_gap_unpair()` API function) or perform pairing without bonding.

Technical details on the BLE Storage implementation can be found in the following readme file:

`<sdk_root_directory>/sdk/interfaces/ble/readme.md`

7.3.6 LE Secure Connections

LE Secure Connections pairing is supported and enabled by default by the SDK using the API described in section 7.3.1. LE Secure Connections pairing will be used if the connected peer supports the feature without the need for the application to specifically request it. If the combination of the devices' capabilities result in a numeric comparison pairing algorithm (introduced for and used for the LE Secure Connections pairing), the application will be notified of a numeric comparison request during pairing by the reception of a `BLE_EVT_GAP_NUMERIC_REQUEST` event and should respond using `ble_gap_numeric_reply()` function.

If the application needs to use only LE Legacy Pairing and disable LE Secure Connections support in the SDK, it should define `dg_configBLE_SECURE_CONNECTIONS` macro to 0 in the application config file.

7.4 Logical Link Control and Adaptation Layer Protocol

The Logical Link Control and Adaptation Layer Protocol, referred to as L2CAP provides connection-oriented and connectionless data services to upper layer protocols with protocol multiplexing capability and segmentation and reassembly operation. As referred in [9], L2CAP permits higher level protocols and applications to transmit and receive upper layer data packets (L2CAP Service Data Units, SDU) up to 64 kilobytes in length. L2CAP also permits per-channel flow control and retransmission.

The L2CAP layer provides logical channels, named L2CAP channels, which are multiplexed over one or more logical links. Each one of the endpoints of an L2CAP channel is referred to by a *channel identifier (CID).*

L2CAP channels may operate in one of five different modes as selected for each L2CAP channel. The modes are:

- Basic L2CAP Mode (equivalent to L2CAP specification in Bluetooth v1.1)
- Flow Control Mode
- Retransmission Mode
- Enhanced Retransmission Mode
- Streaming Mode
- LE Credit Based Flow Control Mode

The null CID `(0x0000)` is never used as destination endpoint. Identifiers from `0x0001` to `0x003F` are reserved for specific L2CAP functions. These channels are referred to as Fixed Channels.

CID `0x0004` is used by the ATT, CID `0x0006` is used by the SMP while CID is used by the signaling channel.

As referred above, the connection-oriented data channels represent a connection between two devices, where a CID, combined with the logical link, identifies each endpoint of the channel.

Figure 20 illustrates the format of the L2CAP PDU in basic mode.

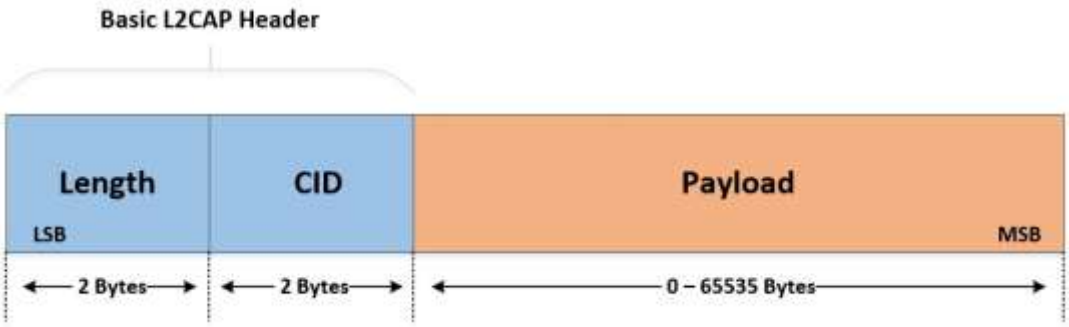

Figure 20: L2CAP PDU format in Basic L2CAP mode on COC

Summarizing:

- L2CAP implementations transfer data between upper layer protocols and the lower layer protocol.
- L2CAP maps channels to Controller logical links, which in turn run over Controller physical links. All logical links going between a local Controller and remote Controller run over a single physical link.
- L2CAP is packet-based but follows a communication model based on channels. A channel represents a data flow between L2CAP entities in remote devices. Channels may be connection-oriented or connectionless. Fixed channels other than the L2CAP connectionless channel (CID 0x0002) and the two L2CAP signaling channels (CIDs 0x0001 and 0x0005) are considered connection-oriented. All channels with dynamically assigned CIDs are connection-oriented.

7.4.1 Credit-Based Flow Control

The Credit-Based Flow Control is an L2CAP mode of operation that when used, allows both devices involved in the LE connection to have complete control over how many packets the peer device is allowed to send. This is achieved by the use of credits that represent the absolute maximum number

of LE frames that the device is willing to accept at a particular moment. The sending entity may send only as many LE-frames as it has credits. If the credit count reaches zero the transmission must stop. If more frames are sent the connection will be closed.

7.4.2 Functions

To establish a LE-Credit Based L2CAP connection, the initiator should send a LE Credit-Based connection request, specifying parameters like the Protocol Service Multiplexer (PSM), Maximum Transmission Unit (MTU), Maximum Payload Size (MPS), and the initial number of credits that the remote peer has to send data. The responding device should respond with a LE Credit-Based Connection Response specifying its own MTU, MPS and initial credits value. PSM is 2-byte odd number that can be used to support multiple implementations of a protocol. The valid range for PSM is between 0x80 - 0xFF. The fixed SIG assigned PSM values are between 0x00 and 0x7F. MTU represents the maximum size of data that the service above L2CAP can send to the remote peer (Maximum size of an SDU – Service Data Unit). MPS is the maximum payload size that an L2CAP entity can receive from the lower layer, and it is equivalent to the maximum PDU payload size. Each MPS corresponds to one credit. One SDU (of size MTU) can be fragmented to one or more PDUs (of size MPS). If the SDU length field value exceeds the receiver's MTU, the receiver shall disconnect the channel. If the payload length of any LE-frame exceeds the receiver's MPS, the receiver shall disconnect the channel. If the sum of the payload lengths for the LE-frames exceeds the specified SDU length, the receiver shall disconnect the channel. After the LE Credit-Based connection request and response frames are received or exchanged, the two entities agree to use the minimum values of MTU and MPS.

As an example, consider two devices that use the following values during the Credit-Based connection request/response procedure:

Table 18: Example of L2CAP COC

	Device A **Connection Request**	**Device B** **Connection Response**
PSM	0x80	- (Field not available on Connection Response)
MTU	100	250
MPS	50	23
Initial Credits	10	20

In this scenario, device B can receive PDUs of size at most 23, and SDUs of size at most 250. On the other hand, device A can receive PDUs of size at most 100, and SDUs of size at most 50. Device A can send 20 PDUs to device B, and should stop until device B updates the available credits. This update could be performed any time during the connection. If device A was to transmit an SDU of size 100 bytes (MTU), it would be fragmented to 5 PDUs of data sizes 21, 23, 23, 23 and 10 respectively. This transmission would consume 5 credits (one credit for every PDU that could be received from device B). Note that the usable payload size of the first PDU is 2 bytes less than the value of MPS as the first LE-frame contains a 2-byte field specifying the total length of the SDU. This is true only for the first frame. In the same manner, the transmission of a 23-byte SDU would require two credits and two PDUs of size 21 and 2 bytes respectively whereas the transmission of a 21-byte SDU would require just one PDU. Maximum SDU length (MTU) can be specified using the `ble_gap_mtu_size_set()` API call.

Table 19: L2CAP COC API- ble_l2cap.h

API call	**Description**
`ble_l2cap_connect()`	Create a l2cap connection oriented channel with remote peer. Connection establishment will be signaled using

API call	Description
	BLE_EVT_L2CAP_CONNECTED event.
ble_l2cap_listen()	Create a connection oriented channel listening for incoming connections. Incoming connection will be signaled using BLE_EVT_L2CAP_CONNECTED event.
ble_l2cap_disconnect()	Disconnect an established L2CAP channel.
ble_l2cap_send()	Send data on channel, response code is signaled using the BLE_EVT_L2CAP_SENT event.
ble_l2cap_add_credits()	Provide additional credits to remote peer. BLE_EVT_L2CAP_REMOTE_CREDITS_CHANGED event will be signaled on the remote peer.

7.4.3 Events

Table 20: L2CAP COC Events – received through ble_get_event() - ble_l2cap.h

Event	Argument	Description
BLE_EVT_L2CAP_CONNECTED	ble_evt_l2cap_connected_t	Channel connected. Members <local_credits> and <remote_credits> specify the initial credits for both sides of connections, whereas <mtu> indicates the negotiated MTU value (Maximum SDU length).
BLE_EVT_L2CAP_CONNECTION_FAILED	ble_evt_l2cap_connection_failed_t	Channel connection failed. Member <status> indicates the reason that connection failed.
BLE_EVT_L2CAP_DISCONNECTED	ble_evt_l2cap_disconnected_t	Channel disconnected. Member <reason> indicates the reason of disconnection.
BLE_EVT_L2CAP_SENT	ble_evt_l2cap_sent_t	Data sent on channel. <remote_credits> member specifies the remaining number of credits that are available for transmission.
BLE_EVT_L2CAP_REMOTE_CREDITS_CHANGED	ble_evt_l2cap_credit_changed_t	Available remote credits changed on channel. <remote_credits> member specifies the remaining number of credits that are available for transmission.
BLE_EVT_L2CAP_DATA_IND	ble_evt_l2cap_data_ind_t	Data received on channel. <local_credits_consumed> member specifies the local credits consumed for the received data.

7.5 LE Data Packet Length Extension

For Bluetooth Core versions 4.0 and 4.1 the maximum Packet Data Unit (PDU) size was 27 octets. Bluetooth Core version 4.2 introduced an important enhancement, namely LE Data Packet Length Extension, which allows for the PDU size to be anywhere between 27 and 251 octets. This means that, for example, the L2CAP layer can now fill up to 245 octets of higher layer data packets in every L2CAP PDU compared to 21 octets with previous Bluetooth Core versions. This significant increase

(more than 10 times) in the number of octets of user data sent per packet allows devices to transfer data up to 2.5 times faster than with previous versions. This will be of great benefit to applications that might require transferring large amounts of data such as Over-the-Air (OTA) firmware updates or downloading large data logs from sensors.

For the default PDU size to be extended on an established connection, the Data Length Update procedure must be performed. According to this control procedure, the `LL_LENGTH_REQ` and `LL_LENGTH_RSP` PDUs must be exchanged by the connected devices so that each is notified of its peer device's capabilities. Each device uses these PDUs to report its maximum receive data channel and maximum transmit data channel PDU payload length and PDU time. After this update procedure, the PDU size for each direction of the connection's data path is set by both controllers to the minimum of the values exchanged.

The DA1468x supports the LE Data Length Extension feature, so the values for the Receive and Transmit Data Length are set by default to the maximum allowed, which is 251 octets. The DA1468x controller when configured as a Bluetooth low energy central device will initiate a Data Length Update upon a new connection if the peer device's controller supports this feature. The BLE Manager will use the values defined by `dg_configBLE_DATA_LENGTH_RX_MAX` and `dg_configBLE_DATA_LENGTH_TX_MAX` macros for this initial Data Length Update negotiation.

7.5.1 Functions

Table 21: LE Data Length Functions – ble_gap.h

Function	Description
`ble_gap_data_length_set()`	Set the maximum Transmit data length and time for an existing connection or the preferred Transmit data length for future connections (that is, the Transmit data length to be used in future data length update negotiations). Connection data length change will be signaled using BLE_EVT_GAP_DATA_LENGTH_CHANGED event.

7.5.2 Macros

Table 22: LE Data Length Definitions

Macro	Default	Description
`dg_configBLE_DATA_LENGTH_RX_MAX`	251	Set the maximum Receive Data Channel PDU Payload Length. Unless ble_gap_data_length_set() is used by the application, this will define the Receive data length present in the LE Data Length Update negotiations done by the device.
`dg_configBLE_DATA_LENGTH_TX_MAX`	251	Set the maximum Transmit Data Channel PDU Payload Length. Unless ble_gap_data_length_set() is used by the application, this will define the Transmit data length present in the LE Data Length Update negotiations done by the device.

7.5.3 Events

Table 23: LE Data Length Events – fetched using ble_get_event() - ble_gap.h

Event	Argument	Description
`BLE_EVT_GAP_DATA_LENGTH_CHANGED`	`ble_evt_gap_data_length_changed_t`	Data Length changed for specified connection. Members <rx_length>,

Event	Argument	Description
		<rx_time>, <tx_length> and <tx_time> specify the values obtained after an LE Data Length Update negotiation (each direction's data length is typically set to the minimum of the values reported by the connected devices).
BLE_EVT_GAP_DATA_LENGTH_SET_FAILED	ble_evt_gap_data_length_set_failed_t	Data Length Set operation failed. Member <status> indicates the reason the set operation failed.

7.6 NVPARAM fields

Table 24 shows the Non-Volatile memory parameters which can be found in <sdk_root_directory>/sdk/adapters/include/platform_nvparam.h

Table 24: NVPARAM fields

Tag	Offset	Length
TAG_BLE_PLATFORM_BD_ADDRESS	0x0000	7
TAG_BLE_PLATFORM_LPCLK_DRIFT	0x0007	3
TAG_BLE_PLATFORM_EXT_WAKEUP_TIME	0x000A	3
TAG_BLE_PLATFORM_OSC_WAKEUP_TIME	0x000D	3
TAG_BLE_PLATFORM_RM_WAKEUP_TIME	0x0010	3
TAG_BLE_PLATFORM_SLEEP_ENABLE	0x0013	2
TAG_BLE_PLATFORM_EXT_WAKEUP_ENABLE	0x0015	2
TAG_BLE_PLATFORM_BLE_CA_TIMER_DUR	0x0017	3
TAG_BLE_PLATFORM_BLE_CRA_TIMER_DUR	0x001A	2
TAG_BLE_PLATFORM_BLE_CA_MIN_RSSI	0x001C	2
TAG_BLE_PLATFORM_BLE_CA_NB_PKT	0x001E	3
TAG_BLE_PLATFORM_BLE_CA_NB_BAD_PKT	0x0021	3
TAG_BLE_PLATFORM_IRK	0x0024	17

7.7 BLE Interrupt Generation

The BLE Core generates interrupts that are used to synchronize with the BLE Software. These interrupts are:

- ble_cscnt_irq: 625µs (slot) base time reference clock interrupt. When sleep mode is used, this interrupt will also be used to program the next advertising, scanning or connection event if it fires before the finetgtim interrupt.
- ble_rx_irq: Reception interrupt at the end of either each CS-RXTHR number of received packets or each received packet. CS-RXTHR can be configured at compile time using

position 44 of `rom_cfg_table_var[]` (`sdk\interfaces\ble\src\stack\plf\black_orca\src\arch\main\ble\jump_table.c`).

- `ble_slp_irq`: End of sleep mode events.
- `ble_event_irq`: End of Advertising / Scanning / Connection event. Used to cleanup/re-initialize state and defer TX/RX handling operations.
- `ble_error_irq`: Error interrupt generated on internal error.
- `ble_finetgtim_irq`: Fine target interrupt. Used to program the next advertising, scanning or connection event. When sleep mode is not used, programming of events will be done by this interrupt's service routine and not by the cscnt interrupt service routine.
- `ble_grossgtim_irq`: Gross target timer interrupt. Used by BLE stack SW timers, e.g. supervision timeout, link layer timeout, etc.

Depending on the context, these interrupts are generated or not. The grayed `ble_cscnt_irq` interrupt pulses in the following figures can be masked or unmasked. The figures that follow assume the following:

- Either extended or deep sleep mode is used.
- CS-RXTHR is 1.

When a sleep mode is used, the event will be programmed in the `cscnt_isr` only if it fires before the `finetgtim_irq` after waking up. The original figures assumed a sleep mode was used, so the points where a `finetgtim` interrupt will fire instead of a `cscnt` interrupt are noted with red color in the same row.

Figure 21 shows an example of interrupt generation for an advertiser device during an advertising event. The first advertising event shows advertising packets only. The second advertising event shows a scanner that tried to exchange data with the advertiser device. The definitions of values such as `T_advEvent` and `T_IFS` can be found in Bluetooth Specification.

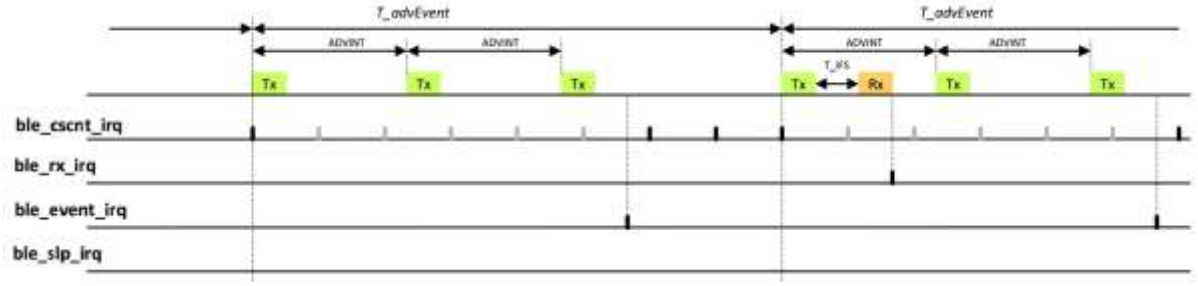

Figure 21: Advertiser Device Interrupts Generation

Figure 22 shows an example of interrupt generation for a scanner device during a scanning event. The first scanning window shows a passive scan event onto channel 39. The second scanning window shows an active scan event with no scan response onto channel 37.

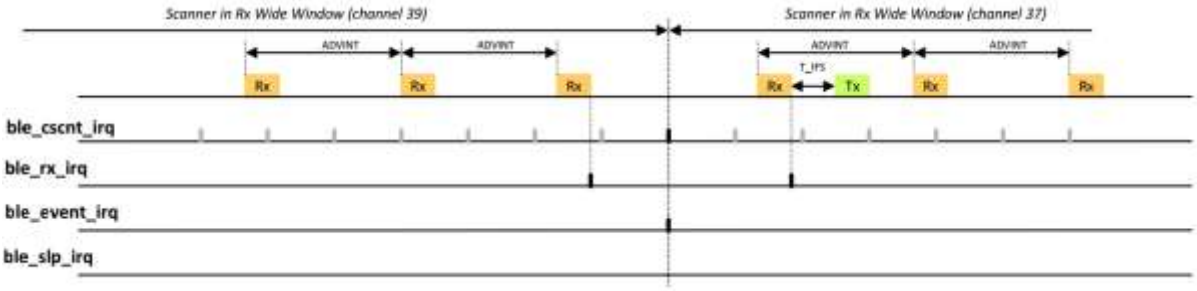

Figure 22: Scanner Device Interrupts Generation

Figure 23 shows an example of interrupt generation for a master device during a Link Layer connection event without Deep Sleep in between anchor points. The first and third connection event show a two packet exchange, while the second connection event shows a four packet exchange.

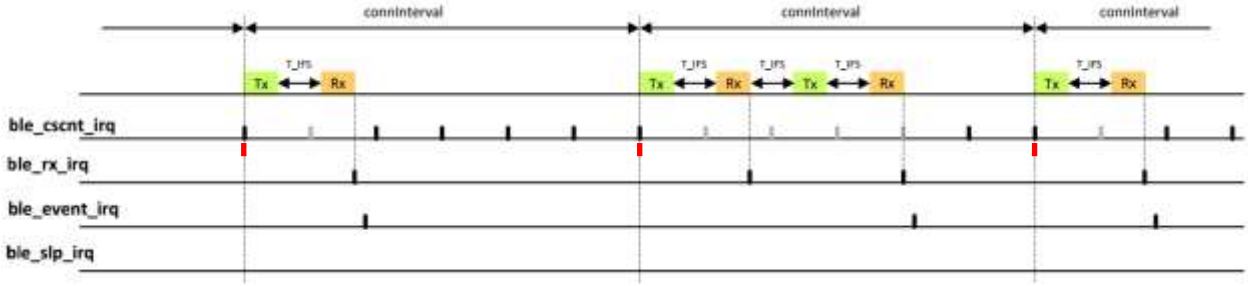

Figure 23: Master Device Interrupts Generation / Link Layer Connection Event without Deep Sleep

Figure 24 shows an example of interrupt generation for a master device during a Link Layer connection event with Deep Sleep in between anchor points.

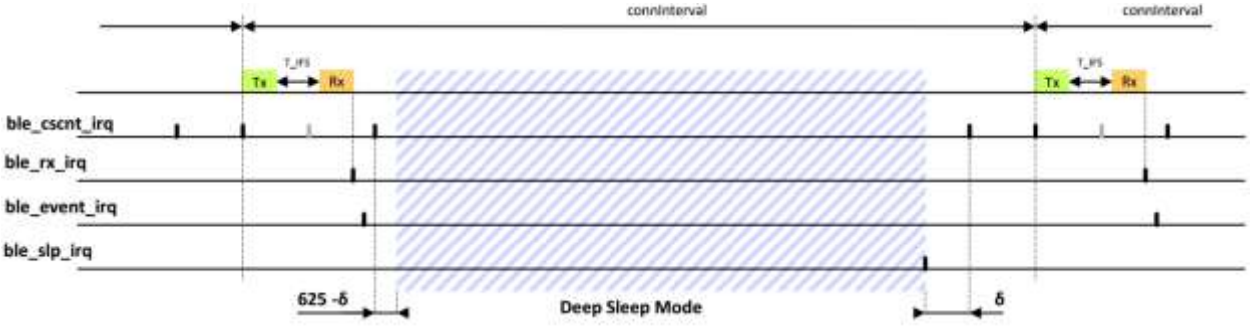

Figure 24: Master Device Interrupts Generation / Link Layer Connection Event with Deep Sleep

Figure 25 shows an example of interrupt generation for a slave device during a Link Layer connection event without Deep Sleep in between anchor points.

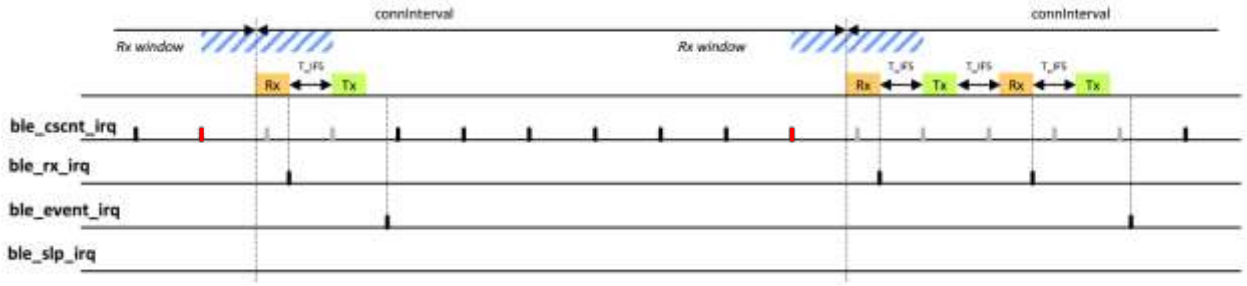

Figure 25: Slave Device Interrupts Generation / Link Layer Connection Event without Deep Sleep

Figure 26 shows an example of interrupt generation for a slave device during a Link Layer connection event with Deep Sleep in between anchor points.

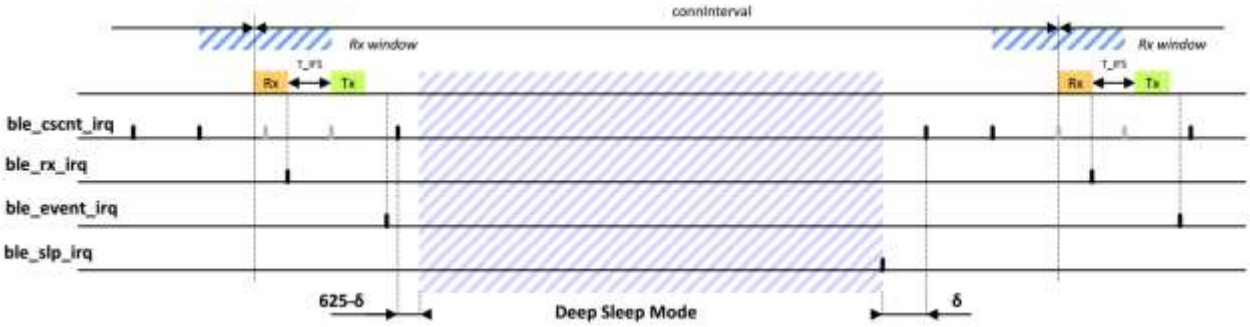

Figure 26: Slave Device Interrupts Generation / Link Layer Connection Event with Deep Sleep

The CSCNT/FINE interrupt which each coming event is programmed in should be serviced with a maximum latency of 300us. If the CSCNT/FINE Interrupt Servicing Routine (ISR) that will program the event is not run 300us+ after the interrupt is asserted, this may result in losing the next connection event (i.e. no data transmission or reception in this connection event).

7.8 Considerations on BLE Task Priorities

The BLE Software in the SDK consists of three modules as shown previously in Figure 6:

1. BLE manager: Provides the interface to the Bluetooth low energy functionality of the chip. The application task uses the BLE API to interface with the BLE manager. The BLE manager is a task that stands between the application and the BLE adapter. It uses the BLE adapter to interface with the BLE stack.
2. BLE adapter: The system task that provides the interface to the BLE stack, hence the BLE IP module. It runs the BLE stack internal scheduler, handles the BLE interrupts, receives the commands or the replies to events from the BLE manager, and passes BLE events to the BLE manager. BLE core functionality is implemented by the BLE adapter task.
3. BLE stack: The software stack that interfaces with the BLE IP and implements the Link Layer and the host stack, specifically the Logical Link Control and Adaptation Protocol (L2CAP), the Security Manager Protocol (SMP), the Attribute Protocol (ATT), the Generic Attribute Profile (GATT) and the Generic Access Profile (GAP). The BLE stack software is stored in the system's ROM and its API header files can be found in `<sdk_root_directory>/sdk/interfaces/ble/src/stack`. The BLE stack default configuration can be modified by editing `ble_stack_config.h` located in `<sdk_root_directory>/sdk/interfaces/ble/src/stack/config`. However, for an application specific change it is better to add the new configuration to the applications `config/custom_config_qspi.h` file which will override the stack defaults.

   The BLE stack software is run under the BLE adapter's task context, which instantiates and initializes the stack.

The two BLE system tasks have by default a higher priority than application tasks in the SDK.

Application developers should always make sure BLE adapter and BLE manager tasks always have a higher priority than application tasks, and that BLE adapter is higher priority than BLE manager.

The BLE adapter instantiates the BLE stack scheduler, which dispatches all the messages between the BLE stack's different layers and calls the appropriate handlers. For example, when an application uses API `ble_gatts_send_event()` to send a GATT notification, this will result in a propagation of messages between the BLE manager, the BLE adapter and several BLE stack's internal layers until it reaches a transmission buffer and, eventually, the air. BLE stack's reception handlers are also run in the context of the BLE adapter's task, so it is crucial that the BLE adapter is always run after a BLE interrupt to handle received data, check if the data programmed for transmission were transmitted and/or acknowledged by the peer device, etc.

7.9 BLE tasks timing requirements

When the application is not making any BLE API calls, the BLE adapter will typically run for a short period of time following every BLE interrupt. For example, in the scenario where a GATT server application sends a notification at every connection event, the BLE adapter will only need to run for about 18us following the `ble_cscnt/finetgtim_irq` interrupt that programs the connection event and about 68us following the `ble_event_irq` (rough average times when 96MHz clock is used and no control packet is received or sent during that connection event but only the notification data). In this scenario, the BLE adapter only needs to run between two consecutive events when the application uses `ble_gatts_send_event()` to send a new notification. In this case it needs roughly 190us for the data to be put in a TX buffer and to be programmed for the next connection event. Figure 27 shows two connection events and the period between them.

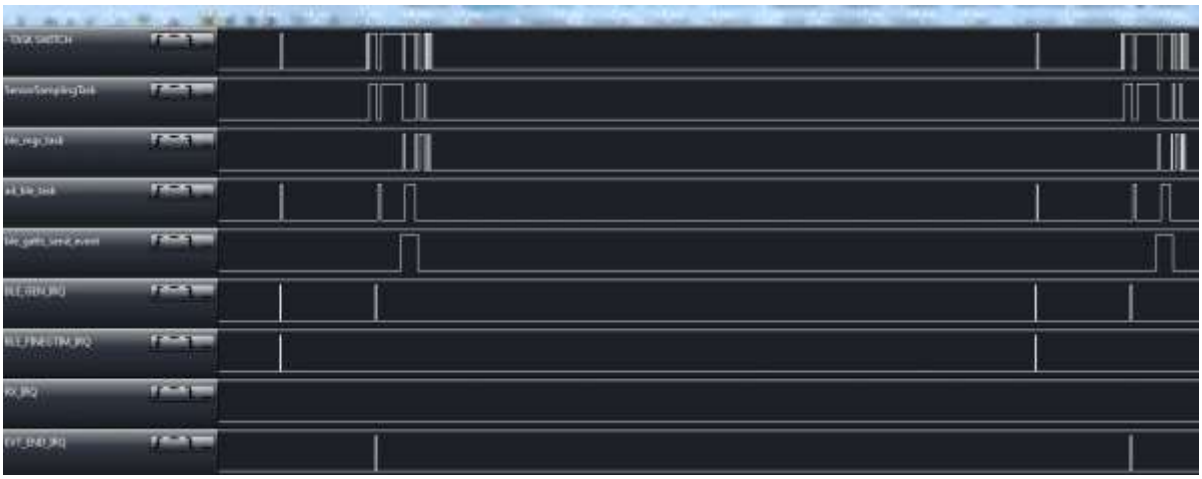

Figure 27: Two connection events

In some scenarios the BLE manager and the BLE adapter will communicate with messages without notifying the application. As an example, upon connection with a peer that uses a resolvable private address the BLE manager will attempt to resolve using known devices IRKs. In this case the BLE manager and BLE adapter will have more running slots. These periods will also be placed right after the `ble_event_irq`.

There are also other cases when the BLE framework will require a reply from the application when, for example, a pair request or a write request is received from the peer. Again, in these cases the BLE adapter and BLE manager will have to run more times in a period between two connection events.

7.10 Attribute operations

As the Attribute protocol is strict (section 4.8.3) when an attribute request such as a read or a write request is received, the BLE stack's GATT layer will switch to a busy state for as long as the request is not completed/handled. In the case of a write request or a read request of an attribute whose value is to be provided by the application, then the application will have to confirm these operations using `ble_gatts_read_cfm()` or `ble_gatts_write_cfm()` respectively (after receiving `BLE_EVT_GATTS_READ_REQ` or `BLE_EVT_GATTS_WRITE_REQ`). In this case, other GATT operations, such as notification sending, will be queued until this request is confirmed. See an example of this in Figure 28.

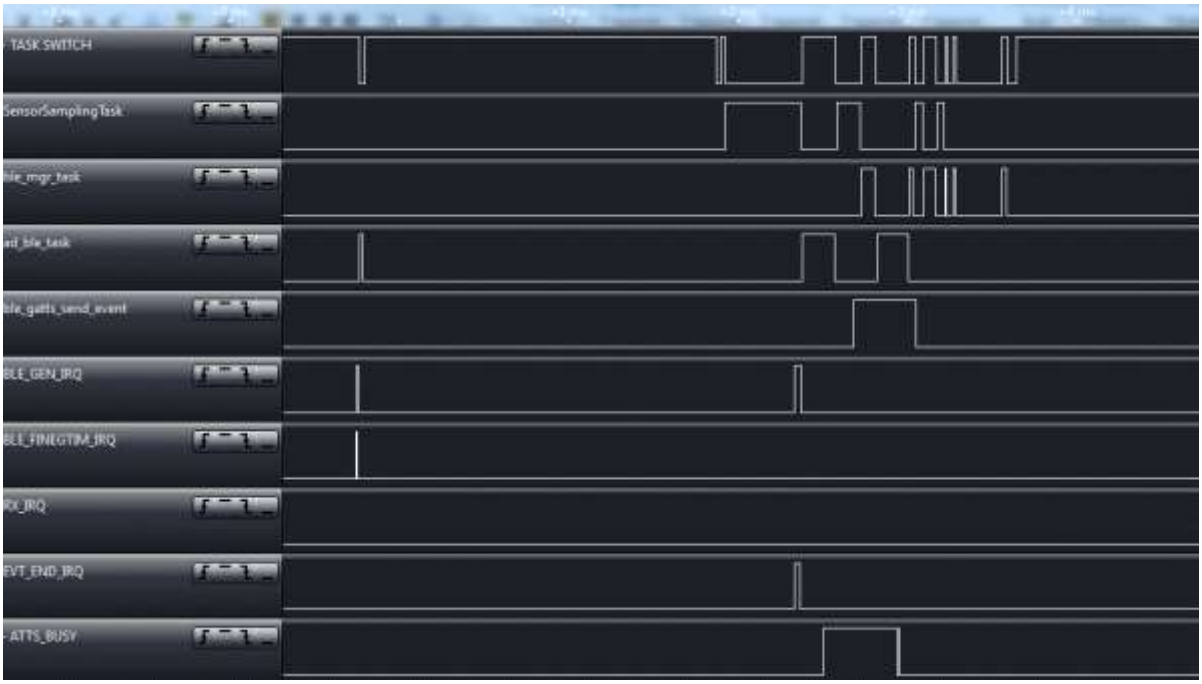

Figure 28: Attribute operations example

This plot shows the period after a connection event during which a write request was received from the peer. In this case this request is confirmed by a task different than the one making the `ble_gatts_send_event()` call. In this case the BLE adapter runs for an additional slot of about 180us. BLE manager also needs to be run in between since it implements the BLE framework functionality on top of the BLE adapter/stack. BLE manager will need in general smaller time slots to run, unless it reads/writes data from/to the flash.

7.11 Bluetooth low energy Application Examples

7.11.1 Advertising Application

The simplest Bluetooth low energy project in the SmartSnippets™ DA1468x SDK is `ble_adv_demo` which is found in the folder `<sdk_root_directory>/projects/dk_apps/demos/ble_adv`. The application starts the device as a peripheral, sets the device name and advertising data and starts advertising. Code 7 is an extract from `main.c`.

```
// Start BLE module as a peripheral device
ble_peripheral_start();

// Set device name
ble_gap_device_name_set("Dialog ADV Demo", ATT_PERM_READ);

// Set advertising data
ble_gap_adv_data_set(sizeof(adv_data), adv_data, 0, NULL);

// Start advertising
        ble_gap_adv_start(GAP_CONN_MODE_UNDIRECTED);
```

Code 10: Set BLE device

No BLE service is added, and the ones exposed are just GAP and GATT services. The infinite loop that implements the lifetime behavior of the application uses just `ble_get_event(true)` to block indefinitely on the BLE manager's event queue. As soon as an event is posted there, the task unblocks and handles it using a `switch` case. Code 11 shows main loop from `main.c`.

```
for (;;) {
        ble_evt_hdr_t *hdr;
        /* notify watchdog on each loop */
        sys_watchdog_notify(wdog_id);
        /* suspend watchdog while blocking on ble_get_event() */
        sys_watchdog_suspend(wdog_id);
        /*
         * Wait for a BLE event - this task will block
         * indefinitely until something is received.
         */
        hdr = ble_get_event(true);
        /* resume watchdog */
        sys_watchdog_notify_and_resume(wdog_id);
        if (!hdr) {
                continue;
        }
        switch (hdr->evt_code) {
        case BLE_EVT_GAP_CONNECTED:
                handle_evt_gap_connected((ble_evt_gap_connected_t *) hdr);
                break;
        case BLE_EVT_GAP_DISCONNECTED:
                handle_evt_gap_disconnected((ble_evt_gap_disconnected_t *) hdr);
                break;
        case BLE_EVT_GAP_PAIR_REQ:
        {
                ble_evt_gap_pair_req_t *evt = (ble_evt_gap_pair_req_t *) hdr;
                ble_gap_pair_reply(evt->conn_idx, true, evt->bond);
                break;
        }
        default:
                ble_handle_event_default(hdr);
                break;
       }
        // Free event buffer
        OS_FREE(hdr);
}
```

Code 11: Example of event handle

Since the BLE service framework is not used, the only events handled by the application are the three events handled by the `switch` case: connection, disconnection and pair request. This makes sense for this application as its only purpose is to start a connectable advertising, restart it in case of a disconnection and respond to pair requests from devices that require pairing/bonding upon connection.

Running this project will result in an advertising Bluetooth low energy peripheral device exposing GAP and GATT services. GAP service attributes can be read using any standard Bluetooth low energy central device. An example is described in Section 4.1.5 of the Software Developers Guide [3].

7.11.2 Peripheral Application

The `ble_peripheral` project is a good starting point for developing Bluetooth low energy peripheral applications. It is found in folder `<sdk_root_directory>/projects/dk_apps/demos/ble_peripheral`. Unlike other example projects, it does not implement a specific profile, but instead exposes several BLE services via a GATT server.

The application's initialization is similar to other projects that implement Bluetooth low energy peripheral applications. It uses the BLE service framework to instantiate several Bluetooth low energy services:

- Battery Service (multiple instances)
- Current Time Service
- Device Information Service
- Scan Parameter Service
- Dialog Debug Service
- Custom User Service

In addition to Bluetooth SIG-adopted services, `ble_peripheral` project instantiates two more services, Dialog Debug Service and a custom user service.

The Dialog Debug Service can be used to interact with the services that the application exposes using a Control Point characteristic to write commands and receive notifications from. A detailed description of the ways to interact with the Dialog Debug Service is included in the *readme.md* file inside the project's folder.

The custom user service does not define any specific functionality other than using 128-bit UUIDs for services, characteristics and descriptors. This custom service, referred to as `myservice` in the project source code, is an example of implementing a custom service using BLE API calls to create its attribute database. No specific functionality is defined when one of these attributes is read or written. More details on how to create and use custom services will be given in section 7.14.

After the attribute database is created, the device will end-up advertising and it will wait for a connection event.

The `ble_peripheral` project uses the BLE service framework to handle service events, the application also defines handlers for connection, advertising completion and pair request events. The `ble_peripheral` project stands in terms of its completeness somewhere between the `ble_adv_demo` and a full profile like the `pxp_reporter`.

The services the project will expose can be configured using the file `config/ble_peripheral_config.h`.

7.11.3 Central Application

The `ble_central` project is found in folder `<sdk_root_directory>/projects/dk_apps/ble_central`. It is the recommended starting point for creating a Bluetooth low energy central application. The application initialization is similar to the previously described projects. The difference is that the device is configured as a Bluetooth low energy central device and no attribute database is created as the device implements a GATT client. Code 12 is extracted from `ble_central_task.c`.


```
ble_central_start();
ble_register_app();
```

Code 12: Configure device as a BLE central

After configuring the Bluetooth low energy parameters, the device attempts to connect to another device. To work with this demo the other device must expose the specific Bluetooth Device (BD) address defined in the `addr` structure:

```
ble_gap_connect(&addr, &cp);
```

Code 13: Connection to another device

The project is configured to connect by default to another DA1468x device, preferably one that exposes a couple of services such as `ble_peripheral`. A good experiment would be to run the `ble_central` project after having programmed another DA1468x board with `ble_peripheral`. Of course, `addr` must be modified to force the `ble_central` device to connect to a device with a different BD address. The default BD address of a project can be changed via the project's `config/custom_config_qspi.h` by adding the following override.

```
#define defaultBLE_STATIC_ADDRESS      {0x02,0x00,0x80,0xCA,0xEA,0x80}
```

Upon connection, the `handle_evt_gap_connected()` handler uses BLE API call `ble_gattc_browse()` (or `ble_gattc_discover_svc()`, `ble_gattc_discover_char()` and `ble_gattc_discover_desc()`, if `CFG_USE_BROWSE_API` macro is set to 0) to discover all services, characteristics and descriptors of the peer device. The project uses the serial interface to print information of the discovered attribute database.

The options of the project can be configured using the file `config/ble_central_config.h`.

7.11.4 Multi-Link Application

The `ble_multi_link` demo located in `<sdk_root_directory>/projects/dk_apps/demos` folder is a project designed to demonstrate the Bluetooth Core version 4.1 feature LE Topology. The device is initialized to have both Bluetooth low energy central and peripheral roles using `ble_enable()` and `ble_gap_set_role(GAP_PERIPHERAL_ROLE | GAP_CENTRAL_ROLE)` API calls. The project instantiates the custom Dialog Multi-Link Service, which exposes a single characteristic, named Peripheral Address.

The demo flow is illustrated in Figure 29. The Multi-link device with BD_ADDR2 starts advertising and waiting for a connection originating from a central Bluetooth low energy device (BD_ADDR3). After the central device is connected to the Multi-Link demo device (step 1), it can write the BD address of another BLE peripheral device (BD_ADDR3) to the Peripheral Address characteristic exposed by the Dialog Multi-Link Service (step 2). This will result in the Multi-Link demo device trying to connect, as a Bluetooth low energy central device to the peripheral device with BD_ADDR3 (step 3). After this connection is established, the Multi-Link demo device will have two concurrent connections one as master and one as slave.

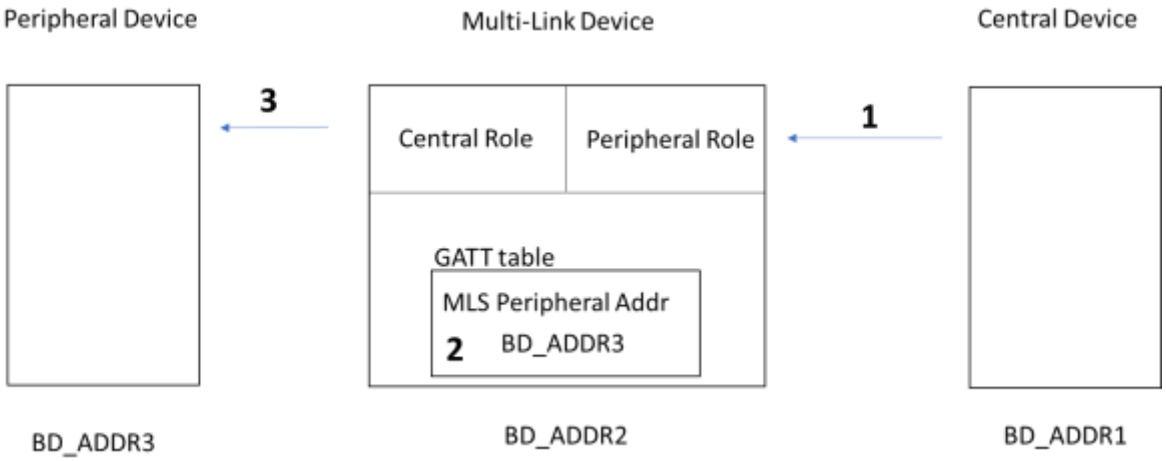

Figure 29: Architecture of Multi-Link Demo

The Multi-Link demo project uses the serial interface to output information about the connected devices.

7.11.5 External Host Application

- The Bluetooth low energy external host project is found in folder `<sdk_root_directory>/projects/dk_apps/demos/ble_external_host`. It is the only project that does not use the BLE framework and the Dialog BLE API. Instead, its application task is a custom adapter created to send and receive HCI and BLE stack proprietary messages over the serial interface, allowing the development of the host on a separate processor.
- After building the project, the user should make sure that the RTS/CTS pins are connected on the Development Kit. Then, a host can send HCI messages over the serial interface and receive the controller's responses.

7.12 BLE profile projects

- In addition to the projects described in the previous sections, there are several application projects that implement Bluetooth low energy profiles. These projects are more complex and provide a full implementation of Bluetooth low energy applications. As such they provide a good reference on how to combine the Bluetooth low energy functionality with several OS mechanisms, GPIO handling and interfacing with external sensors.
- The profiles implemented are the following:
  - HID over GATT Profile (HOGP) – Device role (`hogp_device`) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`
  - HID over GATT Profile (HOGP) – Host role (`hogp_host`) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`
  - Heart Rate Profile – Sensor role (`hrp_sensor`) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`
  - Proximity Profile – Reporter role (`pxp_reporter`) located under `<sdk_root_directory>/projects/dk_apps/demos`
  - Weight Scale Profile – Weight Scale role (`wsp_weightscale`) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`

- Apple Notification Center Service (ANCS) - Notification Consumer (NC) role (ancs) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`
- Blood Pressure Profile (BLP) – Blood Pressure Sensor role (`blp_sensor`) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`
- Bond Management Service (BMS) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`
- Cycling Speed And Cadence collector (CSCP) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`
- Health Thermometer Profile – Thermometer role (`htp_thermometer`) located under `<sdk_root_directory>/projects/dk_apps/ble_profiles`

7.13 Using adopted Bluetooth low energy services

Table 25 summarizes the API header files of the Bluetooth low energy services implemented by the SmartSnippets™ DA1468x SDK. These files can be found under `<sdk_root_directory>/sdk/interfaces/ble_services/include`. The developer can use these APIs to add these services to another project.

Table 25: BLE service API header files

File name	Description
ble_service.h	BLE service framework API: • Add service to framework • Handle event using BLE service framework • Elevate permission • Get number of attributes in a service • Add included services
bas.h	Battery Service – BAS
bcs.h	Body Composition Service – BCS
bms.h	Bond Management Service – BMS
cts.h	Current Time Service – CTS
dis.h	Device Information Service – DIS
dlg_debug.h	Dialog Debug Service
dlg_suota.h	Dialog SUOTA Service
hids.h	Human Interface Device Service – HID
hrs.h	Heart Rate Service – HRS
ias.h	Immediate Alert Service – IAS
lls.h	Link Loss Service – LLS
scps.h	Scan Parameters Service – ScPS
sps.h	Serial Port Service – SPS
tps.h	Tx Power Service – TPS
uds.h	User Data Service – UDS
wss.h	Weight Scale Service – WSS

7.14 Adding a custom service

The following code segments provide an overview of the initialization required to create a new custom service called XXX. It requires the files xxx.c and xxx.h to be created. A good example to base these on is the dlj_mls service in the Multi-Link demo. This provides a single write only characteristic in the service.

Each service needs a structure containing both the generic ble_service_t structure and any callbacks and characteristic handles required by the service. In the example below for service XXX there is one callback and one characteristic defined.

```
typedef struct {
        ble_service_t svc;              // Core BLE service structure
        xxx_cb_t cb;                    // Callback provided by app to xxx
                                        // service to process an event
        uint16_t xxx_char1_val_h;       // Declare handle for each characteristic
                                        // that can be read or written
} xxx_service_t;
```

Code 14: Structure definition for XXX service

The requirements of the initialization function xxx_init() are illustrated below. They key information here is the comments which are explaining what each line is doing.

```
xxx_service_t* xxx_init(callback1){
// Allocate and initialise xxx_service_t structure
// Define any callback functions required by the service, write only in this case
xxx->svc.write_req = <this services write request handler>
// Create primary service UUID with either 16 or 128 bit value
uuid=ble_uuid_from_string() or uuid=ble_uuid_create16()
// add PRIMARY service with X attributes
num_attrs=X
ble_gatts_add_service(&uuid, GATT_SERVICE_PRIMARY, num_attrs)
//Create characteristic 1 for this service and allocate handle for it in GATT
table
ble_gatts_add_characteristic(&uuid, GATT property, ATT permissions, size,0, NULL,
&xxx->xxx_char1_h)
//  Set start_h and pass in null terminated variable length list of all
characteristic handles in the service
ble_gatts_register_service(&xxx->svc.start_h, &xxx->xxx_char1_h,0);
// Calculate end handle for service based on number of attributes in service
xxx->svc.end_h= xxx->svc.start_h + num_attrs;
// add the passed in callback function to service structure
xxx->xxx_cb1=callback1;
// add newly created service to ble framework
ble_service_add(&xxx->svc);
// and return handle for the service to the application
return &xxx->svc
}
```

Code 15: Initialisation function for XXX service

7.15 Extending Bluetooth low energy functionality

The Dialog BLE API can be used to create any Bluetooth low energy application. These API header files are in folder <sdk_root_directory>/sdk/interfaces/ble/include. They come with additional Doxygen documentation and are summarized in Table 26.

Table 26 : Dialog BLE API header files

File name	Description
ble_att.h	Attribute Protocol API: Mostly definitions.
ble_attribdb	Helper to manage complex attributes database.
ble_bufops	Helpers to put and get data from BLE buffers.
ble_common.h	Common API: Functions used for operations not specific to a certain BLE host software component
ble_gap.h	GAP API: • Device parameters configuration: device role, MTU size, device name exposed in the GAP service attribute, etc. • Air operations: Advertise, scan, connect, respond to connection requests, initiate or respond to connection parameters update, etc. • Security operations: Initiate and respond to a pairing or bonding procedure, set the security level, unpair, etc.
ble_gatt	Common definitions for GATT API
ble_gattc.h	GATT client API: • Discover services, characteristics, etc. of a peer device • Read or write a peer device's attributes • Initiate MTU exchanges • Confirm the reception of indications • …
ble_gatts.h	GATT server API: • Set up the attribute database • Set attribute values • Notify/indicate characteristic values • Initiate MTU exchanges • Respond to write and read requests • …
ble_l2cap	BLE L2CAP API.
ble_storage.h	BLE persistent storage API.
ble_uuid.h	BLE UUID declarations and helper functions.

8 The Security Framework

The Security Framework provides a collection of high-level cryptographic algorithms. It is intended to allow the application to access the cryptographic algorithms. It follows a layered software architecture approach that consists of the LLDs of the hardware cryptographic engines, the system services and adapters that provide a higher level API for using the engines and the algorithm implementations. The layered architecture is depicted in Figure 30.

Security framework architecture:

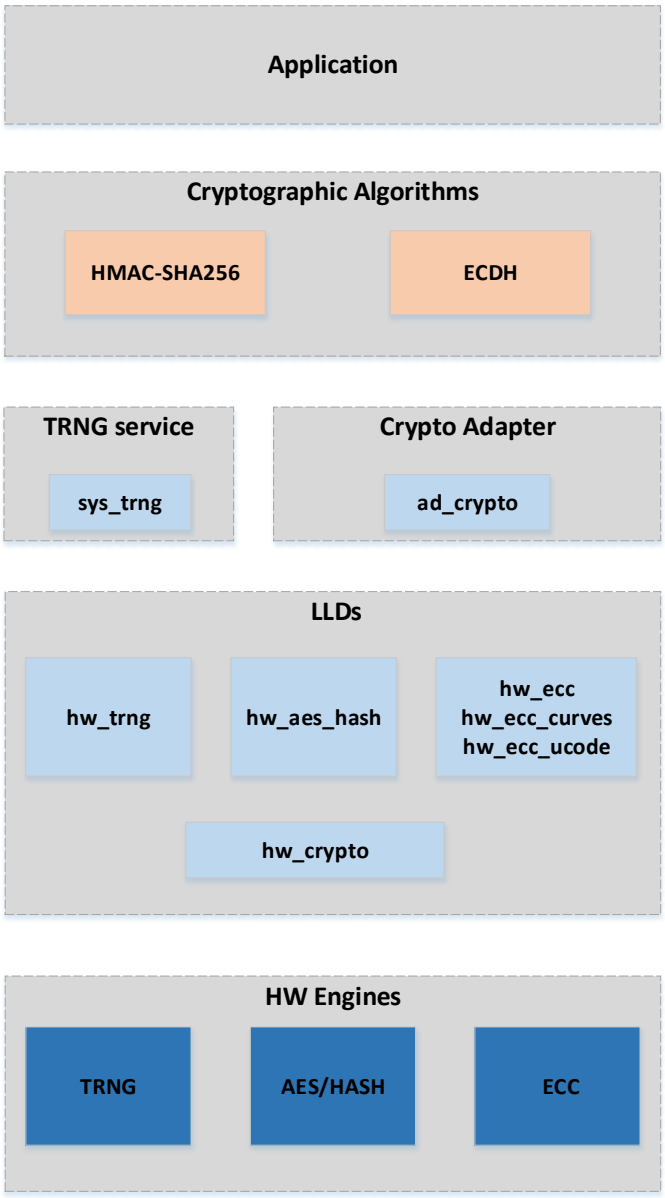

Figure 30: Security framework architecture

8.1 LLDs of the security framework

8.1.1 TRNG Engine LLD

The True Random Number Generator (TRNG) engine is a non-deterministic 32-bit random number generator. The TRNG LLD provides the API for reading generated random numbers, controlling clock enable signals and handling TRNG interrupts. Please refer to the Doxygen documentation of the LLD for details of the API.

8.1.2 AES/HASH Engine LLD

The AES/HASH engine is a hardware accelerator for AES (Advanced Encryption Standard) encryption and HASH functions. It supports ECB, CBC and CTR AES for 128, 192 or 256 bit keys. The supported hash functions are MD5, SHA-1 and SHA-2. The AES/HASH LLD provides the API for controlling clock enable signals, configuring the engine, and handling its interrupts. Please refer to the Doxygen documentation of the LLD for details of the API.

8.1.3 ECC Engine LLD

The ECC engine is a hardware accelerator for Elliptic Curve Cryptography (ECC) operations. It supports arbitrary operand sizes up to 256 bits and uses a part of the system RAM for exchanging input and output data. The ECC LLD provides an API for controlling clock enable signals, configuring the engine, handling its interrupts, writing and reading input/output data. Please refer to the Doxygen documentation of the LLD for details of the API.

8.1.4 Crypto engines LLD

The AES/HASH and ECC engines share a common interrupt source towards the ARM Cortex M0 (`crypto_irq`). The Crypto Engines LLD provides an API for managing this common interrupt for each of the two engines. Please refer to the Doxygen documentation of the LLD for details of the API.

8.2 TRNG service

The TRNG service is a system service for providing random numbers while minimizing the power consumption due to the TRNG engine. It provides an API for reading one or multiple 32-bit numbers or bytes. Please refer to the Doxygen documentation of the service for details of the API.

8.3 Crypto adapter

The crypto adapter is a module that guarantees exclusive access to each of the cryptographic engines (AES/HASH and ECC). It also prevents the system from going into sleep mode while a task has acquired one of the engines for performing cryptographic operations. Finally, it provides a mechanism for event notification related to the engines' operation, thus allowing a task to block until an operation is completed.

The crypto adapter is also the module that allocates and configures the system RAM block that is needed by the ECC engine and loads the ECC microcode whenever it is necessary.

The use of the crypto adapter by a task follows the sequence shown in Figure 70. When task has acquired the resource via the adapter it can then directly call the relevant LLD APIs. When it is finished it must release the resource through the adapter so that any other tasks with pending acquisition requests can use it.

It is possible to reduce the code size of the crypto adapter as an optimization in the use case of only one task needing to access the resource or when the application uses only one of the two cryptographic resources. This is achieved through specific configuration macros that can be found in the adapter's header file located in the `<sdk_root_directory>/bsp/adapters/include/` folder. Please note that the BLE framework is one of the users of the ECC engine.

Please refer to the Doxygen documentation of the crypto adapters for details of the API and application examples.

8.4 Cryptographic algorithms

A set of high-level security-related cryptographic algorithm implementations are provided in folder `<sdk_root_directory>/sdk/interfaces/crypto/include`. These implementations hide the details of the lower layers of the security framework architecture and provide an easy to use API.

8.4.1 Hash-based Message Authentication Code (HMAC)

The hash-based message authentication code (HMAC) is a specific type of message authentication code algorithm that involves the use of a specific cryptographic hash function. The process of generating an HMAC and then validating it at the receiver is shown in Figure 31 and further details on the algorithm can be found in [5] and [6]. The SmartSnippets™ DA1468x SDK provides an implementation of a SHA256 based HMAC, the API of which can be found in `<sdk_root_directory>/sdk/interfaces/crypto/include/crypto_hmac.h`. The API is fully documented in Doxygen and provides examples of use.

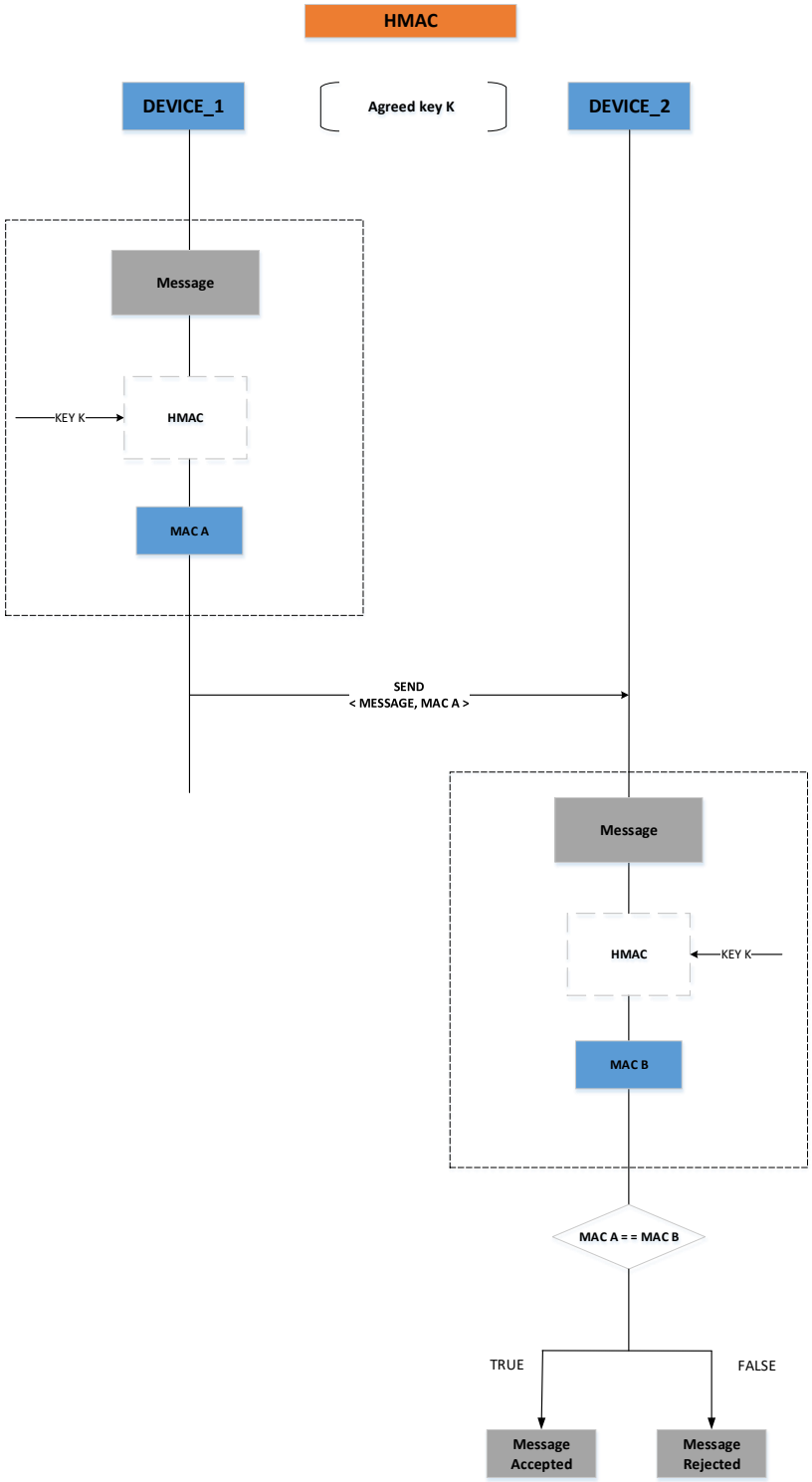

Figure 31: HMAC algorithm

8.4.2 Elliptic Curve Diffie-Hellman (ECDH)

Elliptic Curve Diffie-Hellman is an elliptic curve variant of the Diffie-Hellman algorithm. It allows two parties to generate a common shared secret by exchanging public keys over an insecure channel, after agreeing to use a common set of domain (curve) parameters, as shown in Figure 32. More information about ECDH with ECC can be found in [7]. `Curve25519` is a specific Diffie-Hellman function that allows fast and secure implementations of ECDH. More details about Curve25519 can

be found in [8].The SmartSnippets™ DA1468x SDK provides an implementation of ECDH in `<sdk_root_directory>/sdk/interfaces/crypto/include/crypto_ecdh.h`. The curves supported currently are secp192r1, secp224r1, secp256r1 and Curve25519. It is possible to reduce the code size of the ECDH by configuring specific macros regarding the use of `Curve25519`. For more details you can refer to the Doxygen documentation that includes also examples of usage.

Note 7 That the shared secret obtained with the ECDH key agreement protocol should not be used directly. Instead it should be passed through some form of key derivation function (KDF). The SmartSnippets™ DA1468x SDK security framework does not currently provide KDF implementations.

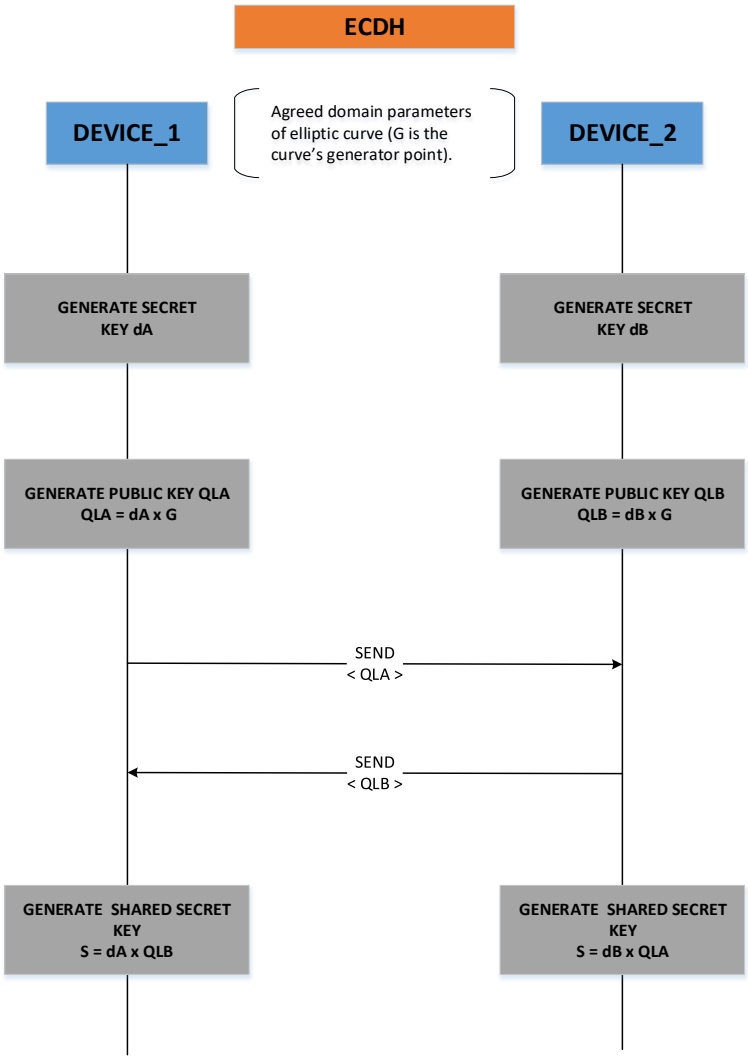

Figure 32: ECDH algorithm

9 System Management

The DA1468x has a number of independent power domains which are shown in Figure 33 that can be switched off to minimize power consumption.

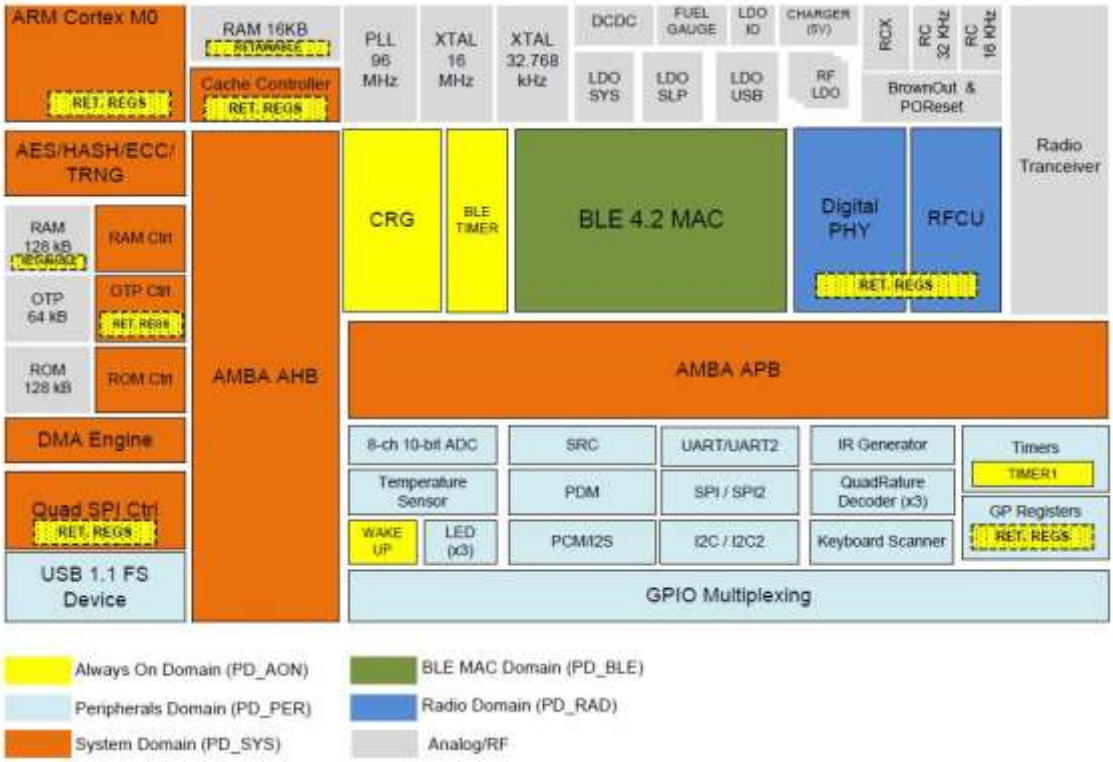

Figure 33: DA1468x Power Domains

The Clock and Power Manager (CPM) is responsible for managing these power domains, putting the system into sleep, handling the wake-up and providing the application tasks with a seamless way to control all system clocks. These are described in detail in the following sub-sections.

9.1 Power Modes

DA1468x supports the following power states (modes):

1. Active,
2. Idle,
3. Extended sleep,
4. Hibernation.

In **Active mode**, both System (`PD_SYS`) and Peripheral (`PD_PER`) power domains remain active. The BLE MAC (`PD_BLE`) and the Radio (`PD_RAD`) power domains may or may not remain active while the ARM Cortex M0 processor is able to execute code.

Idle mode is identical to Active mode, except that the ARM Cortex M0 core is executing a `WFI()` instruction which is just waiting for an interrupt to restart code execution. This interrupt may be an external one or simply the internal tick interrupt originating from the system's clock. Idle mode offers significantly lower power consumption than the Active one.

When in **Extended Sleep** mode all power domains are powered down except for the Always-ON (`PD_AON`) domain. The `PD_AON` domain remains active to supply power to the blocks that can wake the system up, such as the BLE timer, the wake-up controller etc. The `XTAL16M` is stopped with only the

low power clock remaining operational and the configured retained RAM blocks (Section 13.3) which retain data or in certain cases, code during extended sleep.

Hibernation mode bears two significant differences to Extended Sleep mode:

1. Retained RAM is also powered down.
2. The next wakeup event will internally generate a reset signal.

The first two modes, Active and Idle are categorized as power-up modes, during which the system is fully functional. The other two modes, Extended Sleep and Hibernation are categorized as power-down modes, during which the system is sleeping and thus consuming significantly less power.

9.2 Wake-up Process

9.2.1 Wake-up modes

Two wake-up modes are supported:

1. Resume the OS without waiting for the XTAL16M to settle. This is the default setting.
2. Resume the OS only after the XTAL16M has settled. This mode ensures that the application will run using the high-precision clock.

If any task that requires an accurate clock can be blocked until the XTAL16M clock becomes available, using the default setting is always recommended. Resuming the OS without waiting for the XTAL16M to settle ensures the lowest possible power consumption in typical applications. This is possible since there are tasks which only require RC16 to run and so they can finish earlier. This allows the system to return to Sleep mode with minimal delay.

9.2.2 Wake-up events

When in power-down mode, DA1468x can wake-up in two ways:

1. Synchronously, from the Timer1 or the BLE timer in Extended Sleep mode only.
2. Asynchronously, from the Wake-up timer or the VBUS interrupt.

Certain applications require the system to exit Sleep mode to serve an OS Timer or a BLE event. If these are time-based events the low-power clock must be available during Sleep mode. Synchronous wake-up however is only supported in the Extended Sleep mode.

9.3 Sleep architecture

The Sleep architecture design is built around the following principles:

1. The system must be able to wake up synchronously to serve OS and BLE events.
2. The requirements of OS and BLE events may be different. Specifically, BLE events require the XTAL16M to have settled and be set as the system clock. On the other hand, an OS event may or may not have such requirement. For instance, an OS task that needs to read a value from a sensor via the I2C interface does not require the XTAL16M clock. However, if the OS task uses the UART to read the sensor then the XTAL16M is mandatory.
3. The system must be able to wake-up asynchronously.
4. The process of switching between Sleep and Wake-up modes should be both simple and deterministic.
5. The Clock and Power Manager (CPM) must control only the System and the Peripherals power domains. The BLE and the Radio power domains must be controlled independently by the BLE driver.
6. No error will be introduced by the software architecture to the low power clock other than that caused by the inherent physical characteristics of the external crystal (in the case of XTAL32K) or the RCX.

There are two sources that control the synchronous system wake-up, the OS and the BLE. The implementation should use a single timer to wake-up the system, regardless of which source it serves. This is to ensure that statements 4 and 6 of the previous list are covered. Since the BLE timer is not accessible when the BLE is powered-down, Timer1 becomes the only available option.

The implementation of Timer1 facilitates a system wake-up process with enough time for both BLE initiation and XTAL16M settling. When an XTAL 16RDY interrupt arrives, XTAL16M is set as the system clock and BLE wake-up takes place. Provided that the only task of BLE_LP_ISR is to power-up the BLE core, the necessary BLE_WAKEUP_LP interrupt is programmed to occur in a time window in which the BLE core should be already functional.

The CPM prevents the system entering Sleep mode when the BLE power domain is still active. Only after the BLE completes event handling and the existing BLE driver puts the BLE power domain into power down, will the CPM take over and put the whole system in Sleep mode. To ensure that the system will exit Sleep mode, the CPM calculates the next wake-up time. A certain time-frame which is defined as the maximum sleep time, may be applied prior to servicing the next synchronous event. To correctly program the sleep time, the CPM takes into account which OS or BLE event should be considered as the next wake-up source. XTAL16M settling time is not included in the calculation when an OS event wakeup is expected, only in the case of a BLE wakeup event. Possible overlaps of OS and BLE events are also taken into consideration.

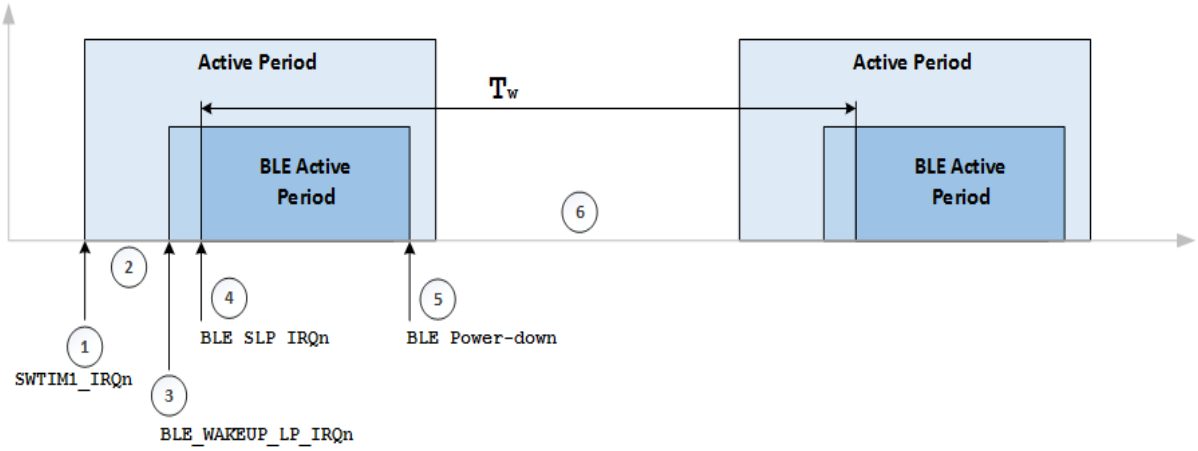

Figure 34: Synchronous BLE event

Figure 34 presents a typical scenario, where the system wakes-up for servicing a BLE event. The interrupts along with the system state each one triggers are clearly depicted:

1. After SWTIM1_IRQn interrupt, the system wakes-up and the clock is RC16.
2. XTAL16M settling time window. During this time, the system clock switches back to XTAL16M (or PLL) from RC16.
3. Using a BLE_WAKEUP_LP_IRQn interrupt, the BLE notifies the system that it has woken-up and is now available.
4. The BLE core is active when a BLE_SLP_IRQn interrupt occurs.
5. BLE goes to power-down after informing the CPM of the next wake-up time (Tw).
6. Inactive Period.

The inactive period is calculated by the CPM based on the time that BLE is scheduled to wake-up. By the time the next BLE_WAKEUP_LP_IRQn arrives, the XTAL16M must have already settled and the clock switching from RC16 to XTAL16M must have been completed.

This scenario can be extended to include a wake-up from an asynchronous request, where the system is triggered by a request originated from the wake-up controller. In such case, the system is solely based on the RC16 clock, therefore doesn't involve the XTAL16M. After the interrupt has been

handled, the CPM puts the system back into Sleep mode by repeating the same process used in the synchronous case.

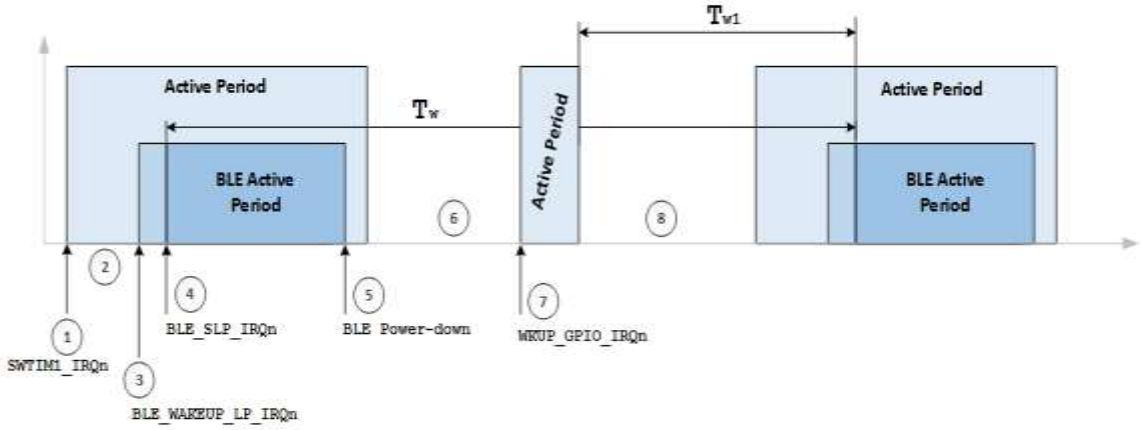

Figure 35: Asynchronous BLE event

Figure 35 presents the wake-up and sleeping process during an asynchronous event.

1. After SWTIM1_IRQn interrupt, the system wakes-up and the clock is RC16.
2. XTAL16M settling time window. During this time, the system clock switches back to XTAL16M (or PLL) from RC16.
3. Using a BLE_WAKEUP_LP_IRQn interrupt, the BLE notifies the system that has woken-up and is now available.
4. The BLE core is active when a BLE_SLP_IRQn interrupt occurs.
5. BLE goes to power-down after informing the CPM of the next wake-up time (Tw).
6. Inactive Period.
7. A WKUP_GPIO_IRQn interrupt is received by the system. During inactive period, the clock is set to RC16.
8. After serving the asynchronous request the CPM puts the system back to Sleep mode. However, the time to wake up is not Tw any more. The CPM must recalculate the time window based on the next scheduled event when BLE needs to be active. The new value is depicted as Tw1 in Figure 35.

To ensure correct operation it is necessary for the CPM to manage all the requirements of the various subsystems of DA1468x. The driver Adapters are designed to simplify the operation of the CPM in managing the sleep requirements of all the drivers in the system.

The Adapter for each driver implements a layer between the application and the low level hardware drivers. They control the tasks' access to the resource so that multiple tasks can safely use the resource. The sharing of a resource between multiple tasks is illustrated Figure 70.

The Adapter is also responsible for handling the power management for the driver. It will initialize the hardware resource during wake-up and deal with the special case where the resource cannot be fully operational until the XTAL16M is available. In this case the Adapter can handle separately the partial initialization via the callback ad_wake_up_ind and the full initialization of the resource via the callback ad_xtal16m_ready_ind.

The Adapter(s) register to the CPM so that the CPM can inform them about the progress of wake-up or sleep entry. The API that is provided to the Adapters for this purpose is:

Table 27 : API for the adapters

API for the adapters	Description
`pm_id_t pm_register_adapter(const adapter_call_backs_t *cb)`	Registers an Adapter to the CPM.
`void pm_unregister_adapter(pm_id_t id)`	Unregisters an Adapter from the CPM.

At registration, the Adapter provides its APIs for communication with the CPM. This API is listed in Table 28 and used by Adapters that are implemented as part of the SmartSnippets™ DA1468x SDK.

Table 28: API for the communication with the CPM

API for the communication with the CPM	Description
`bool ad_prepare_for_sleep(void)`	The CPM inquires the Adapter about whether the system can go to sleep.
`void ad_sleep_canceled(void)`	If an Adapter rejects sleep, the CPM calls this function to resume any Adapters that have previously accepted it.
`void ad_wake_up_ind(bool)`	The CPM informs the Adapter that the pad latches are to be removed so that it re-initializes the GPIOs that are used by the hardware resource it controls and, depending on the resource type, the resource itself.
`void ad_xtal16m_ready_ind(void)`	The CPM informs the Adapter that the XTAL16M is ready and is or may be set as the system clock.

The Adapter also informs the CPM about the time it needs to prepare the hardware resource for power-down. The CPM uses this information when it executes the sleep entry procedure. This time will be zero in general but there are exceptions (e.g. the UART interface).

When the CPM is invoked to put the system to sleep (the OS is idle) and the BLE is powered-down, the CPM executes the following steps:

1. Calculates the time when the OS needs to wake-up to serve its next scheduled event.
2. Calculates the time when the BLE needs to wake-up to serve its next scheduled event.
3. Determines the maximum sleep time that is allowed based on the results from Steps 1 and 2.
4. If the calculated sleep time is larger than a pre-set minimum sleep time below which sleeping is not power efficient (`dg_configMIN_SLEEP_TIME`), then the CPM informs all the registered Adapters about its intention to put the system in power-down by calling each Adapter's `ad_prepare_for_sleep()` function. The return value of this function allows the Adapters to tell the CPM that it cannot enter sleep as it still has work to do.
5. If all Adapters confirm the power-down entry, the CPM continues and puts the system into sleep.
6. If at least one Adapter rejects the power-down entry, the CPM informs the Adapters that have already accepted it, to resume normal operation and puts the system into Idle mode.

During wake-up, CPM and Adapters interact twice:

First, when it prepares the system for power-up the CPM activates the Peripherals power-domain where it calls each Adapter's `ad_wake_up_ind()` to inform them of the imminent removal of the pad latches.

Second, when the `XTAL16RDY_IRQn` ISR is called after the XTAL16M has settled it calls each Adapter's `ad_xtal16m_ready_ind()` to inform them that the clock is ready.

Any Adapters such as the UART which need the high precision clock will perform the initialization of the resource at this point.

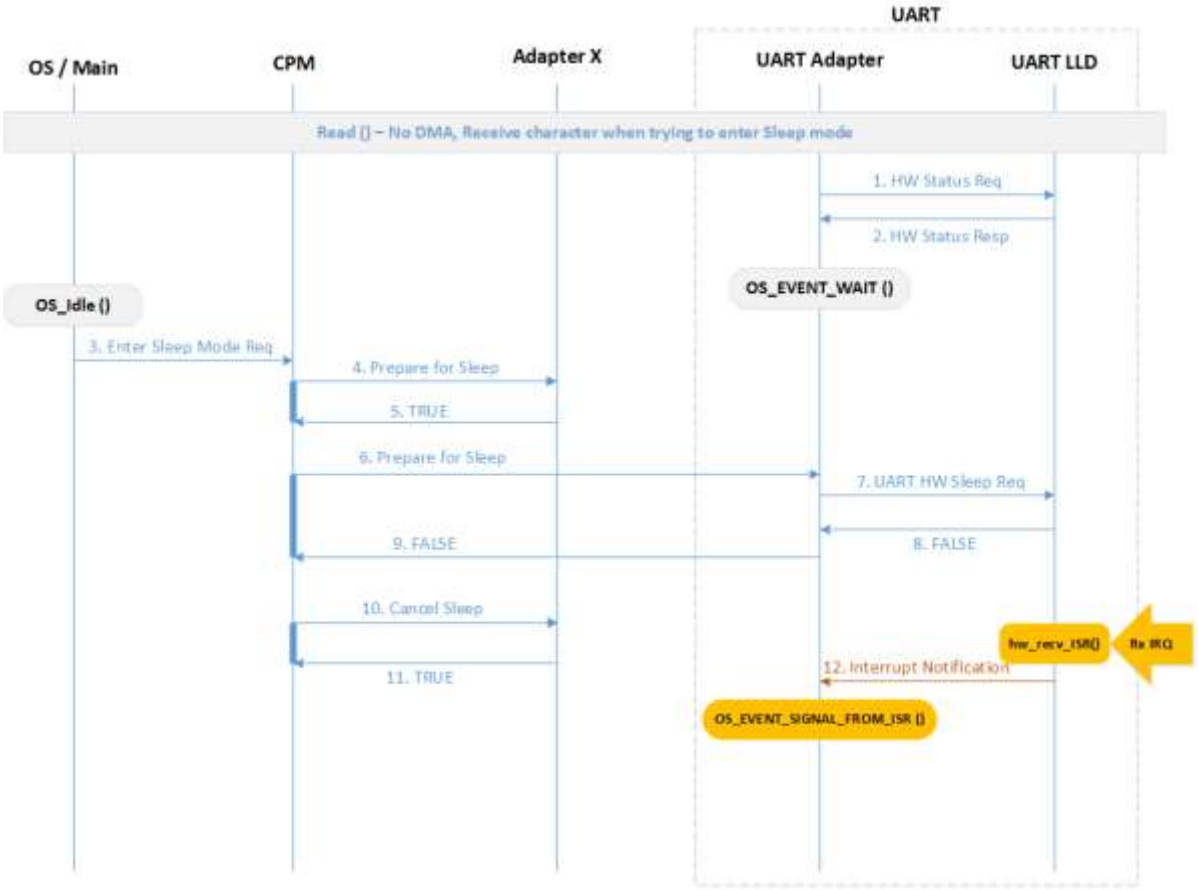

Figure 36: CPM and Adapter Interaction - an Adapter aborts sleep

The interaction between the CPM and the Adapters is shown in Figure 36. According to the scenario presented there, a system that consists of the OS, the CPM and two Adapters (Adapter X and UART Adapter) is about to switch mode, from Active to Sleep. The UART module includes two submodules, UART Adapter and UART LLD, which exchange hardware status messages [e.g. `hw_uart_receive()`] while waiting for an OS-originated event.

To enter power-down `OS_Idle()` issues a “Enter Sleep Mode Req” [`pm_sleep_enter()`] to the CPM. The CPM will first calculate sleep time and if sleep is possible will contact the first Adapter in the system, Adapter X, and send it a “Prepare for Sleep Req” [`*ad_prepare_for_sleep()`]. Adapter X accepts the power-down, starts preparing for it and responds by a simple “TRUE” response. Then the CPM calls the next Adapter in sequence, the UART. This adapter tries to deactivate the UART block unit, by notifying UART LLD via a “UART HW Sleep Req” [`hw_uart_rx_off()`]. However, during this process a character is received in the UART LLD, the Flow OFF check fails and the UART Adapter receives a “FALSE” notification which is forwarded to the CPM. After receiving the “FALSE” notification, CPM contacts Adapter X to inform that the system will remain in power-up state by sending a “Cancel Sleep” message [`*ad_sleep_canceled()`]. Finally, the system is placed in Idle state.

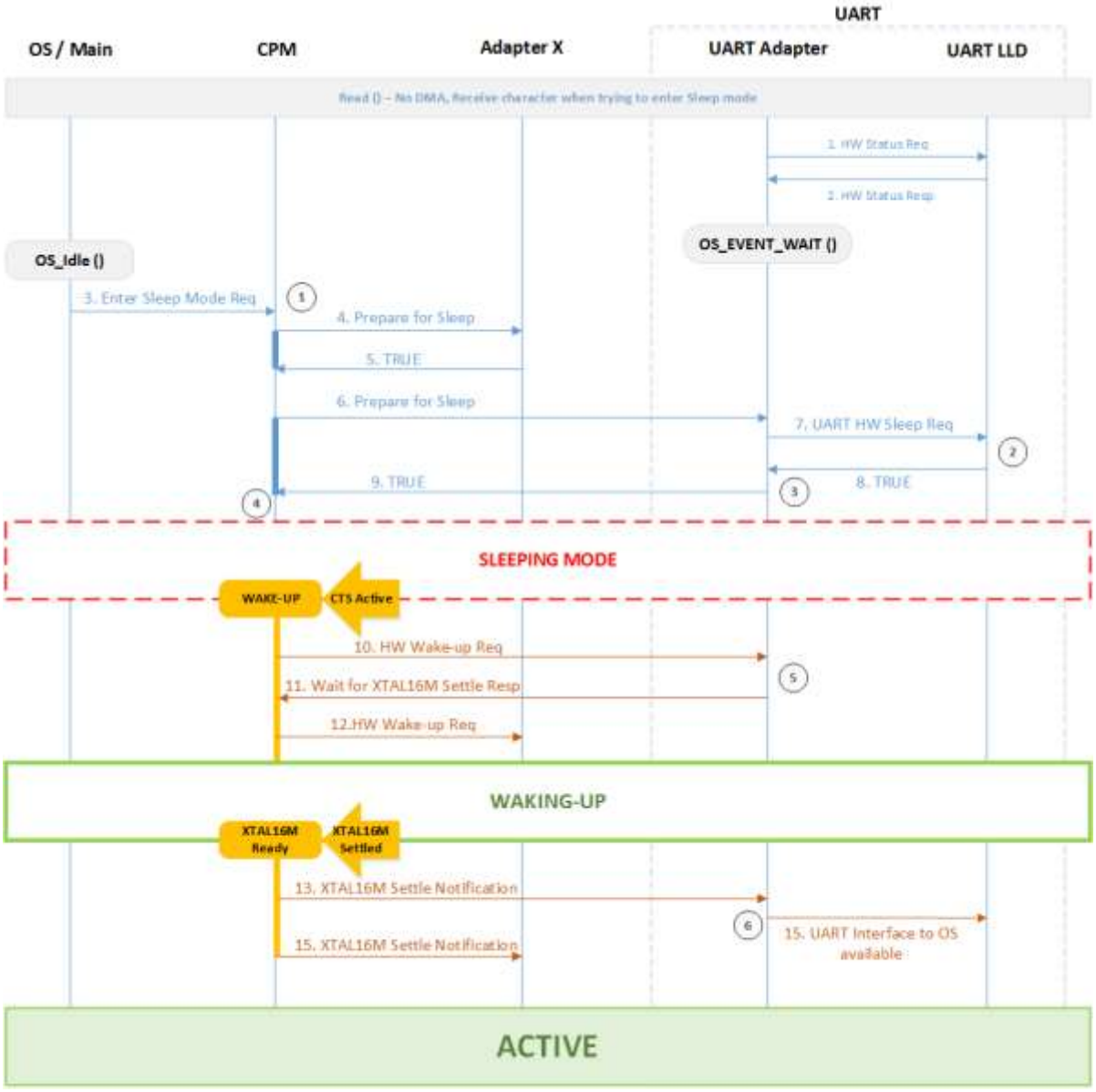

Figure 37: CPM and Adapter Interaction during Sleep/Active mode switch

Figure 37 illustrates the interaction of the CPM with the Adapters of the previous example (Figure 36), in a successful sleep entry and a subsequent wake-up from an external event (the CTS pin changes state). In this example, both Adapter X and the UART Adapter accept the power-down so the CPM puts the system into the proper power-down state. While sleeping, a transmit request from the remote host is issued and the system wakes-up. Before removing the pad latches, the CPM calls the `ad_wakeup_ind()` of both Adapters to perform partial or full initialization. The UART Adapter detects the remote host's request and instructs the CPM not to put the system back to sleep until the XTAL16M has settled and the UART block is turned-on by calling the `pm_defer_sleep_for()`. This is the simpler approach as the CPM will immediately put the system to Idle state if there is nothing else to be done. If the call to `pm_defer_sleep_for()` is omitted then the system will be put into Idle state but following a more complex procedure. This procedure involves the CPM contacting the Adapters, the UART Adapter will deny the sleep request and the CPM will call the Idle `WFI()` after cancelling sleep for Adapter X. When the XTAL16M settles the CPM calls the `ad_xtal16m_ready_ind()` of both Adapters. The UART Adapter will check whether the system clock is the high precision clock and, if not, request it be changed. This is completed without delay. Then it can proceed with the initialization of the UART hardware.

9.3.1 BLE Wake-up

Special code is included in the SmartSnippets™ DA1468x SDK to ensure the proper wake-up of the BLE core. If a problem occurs, then an assertion is triggered. Two macros can be used to allow a percentage of such errors to occur without triggering an assertion: BLE_MAX_MISSES_ALLOWED and BLE_MAX_DELAYS_ALLOWED. Before explaining how these macros operate, a discussion about the BLE sleep and wake-up procedures is necessary.

During the BLE sleep preparation step the software calculates the time (time=tS) that the BLE core can be in sleep mode (or, in other words, the time when the BLE core must wake-up). The software then programs this time to the BLE core and instructs it to enter power-down state. Then the BLE core switches from the 16MHz clock to the low power clock and enters power-down state. While in this mode, the BLE power domain can be turned off to save power.

The wake-up sequence of the BLE core consist of the following steps (please refer also to the datasheet):

- BLE_WAKEUP_LP_IRQ occurs at a predefined number of clocks (time = tW) before the time (time = tS) when the BLE core must wake-up. The term "wake-up" refers to the BLE core being powered up and running at 16MHz. So, the service routine of this interrupt is responsible for powering-up the BLE power domain and starting the fast clock before the time tS. It then waits for the BLE_SLP_IRQ to occur to pass the execution to its handler.
- BLE_SLP_IRQ occurs at time tS or later. Ideally, if no delays have occurred then BLE_SLP_IRQ will occur exactly at tS. If, for any reason, the wake-up of the BLE core was delayed then BLE_SLP_IRQ will occur later than tS. When BLE_SLP_IRQ occurs the BLE core automatically switches from the low power clock to the 16MHz clock and enters power-up state. The ISR is responsible for performing clock compensation. This is readjusting the time counters of the BLE core (slot and uses timer) to take into account the sleep period (tS or a time period larger than tS).

The time tW is calculated to take account of potential normal delays in the system (i.e. a critical section that runs with the interrupts disabled) and make sure they do not affect the wake-up of the BLE core. So, by default, tW is larger than the "optimal" setting. Therefore, the service routine of the BLE_WAKEUP_LP_IRQ assumes that the wake-up of the BLE core will have been completed well before the BLE_SLP_IRQ is triggered. Based on this assumption, if the BLE_SLP_IRQ hits right after the wake-up of the BLE core, there is a high chance that the servicing of the BLE_WAKEUP_LP_IRQ has been delayed. This case is checked in the ble_lp_isr() when the code is built in DEVELOPMENT_MODE. Normally, an assertion would hit to indicate this "error". However, the macro BLE_MAX_MISSES_ALLOWED can be used to allow for a small percentage of such delays. This can be used in applications where the maximum time that the system runs with the interrupts disabled, blocking the servicing of the BLE_WAKEUP_LP_IRQ, is larger than usual. The percentage is calculated as:

```
misses_allowed = (BLE_MAX_MISSES_ALLOWED / BLE_WAKEUP_MONITOR_PERIOD) * 100%
```

After the BLE_WAKEUP_LP_IRQ ISR, the BLE_SLP_IRQ ISR executes. slp_isr() checks whether the BLE core was woken up in time or the wake-up was delayed. This is done by checking the programmed sleep time tS with the actual sleep time tA. If tA > tS then the wake-up was delayed by tA-tS low power clock cycles. This is as error as in most cases, it results in missing the BLE event that the system woke-up for. This case can happen if the interrupts were disabled for quite a long time. In that case, ble_wakeup_lp_isr interrupt would have been latched but its servicing would have been deferred until the interrupts were enabled. During this period, the BLE core would be sleeping. So, after powering the BLE core up, the BLE_SLP_IRQ would hit but the actual sleep time would have been larger than the programmed time. The macro BLE_MAX_DELAYS_ALLOWED can be used to allow for a small percentage of such delays in the same way as the macro BLE_MAX_MISSES_ALLOWED is used. The percentage is calculated as:

```
delays_allowed = (BLE_MAX_DELAYS_ALLOWED / BLE_WAKEUP_MONITOR_PERIOD) * 100%
```


BLE_WAKEUP_MONITOR_PERIOD is set to 1024 by default.

9.4 Power configuration

The term "*power configuration*" is used to denote the configuration of the system during sleep. In power-up modes, the system uses the DCDC converter as the main power supply (unless otherwise configured) as this setup offers lower power consumption.

Although the power setup is quite straightforward for the power-up modes, this is not the case for the power-down modes as there are various options for providing power to the power rails (see Figure 38). More specifically,

- The 3.3V output can be driven by the LDO_ret (low power, low driving strength) or the LDO_VBAT_RET (high driving strength but resampling is required periodically to keep the correct voltage level, which increases power consumption).
- The 1.8V output for the Flash can be provided by the LDO_IO_RET or the DCDC. Note that the LDO_IO_RET is powered from the 3.3V rail. Note that the DCDC requires periodic recharging to keep the outputs constant, which increases power consumption.
- The 1.8V output for the external peripherals can be provided by the LDO_IO_RET2 or the DCDC. Note that the LDO_IO_RET2 is also powered from the 3.3V rail. Note that the DCDC requires periodic recharging to keep the outputs constant, which increases power consumption.
- The 1.4V output is not provided during sleep.

The low power clock is required to be active during sleep to be able to wake-up the part of the system that does the resampling of the bandgap or restoring the energy of the inductor of the DCDC.

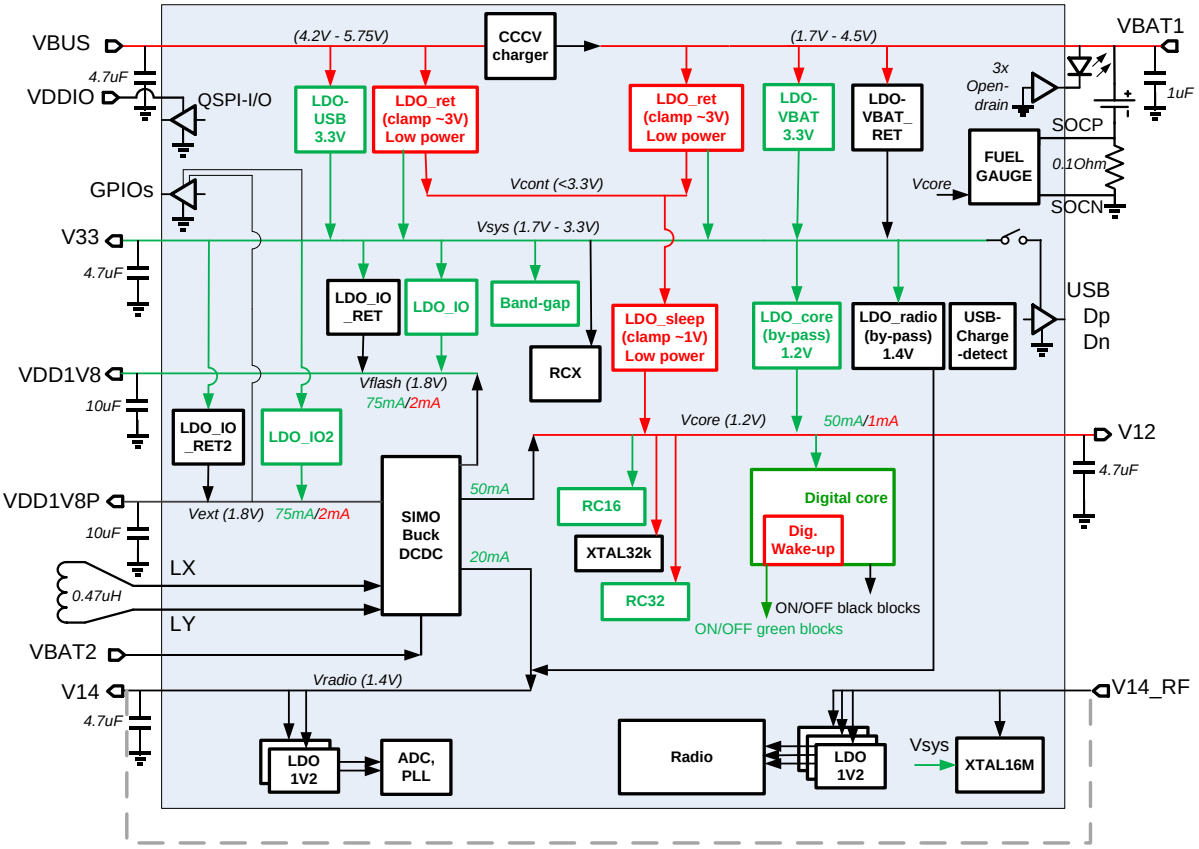

Figure 38: Power Management Unit

In Figure 38, the red color is used for the Sleep state and the green for the Active state.

9.4.1 Recommended Power-Down Power Configuration

1. In the recommended power-down Power Configuration the 3.3V rail is powered via the LDO_VBAT_RET. In this case it is required to bring-up part of the system periodically to resample the bandgap. The 1.8V for the Flash and for the external peripherals are provided via the LDO_IO_RET and LDO_IO_RET2 respectively.

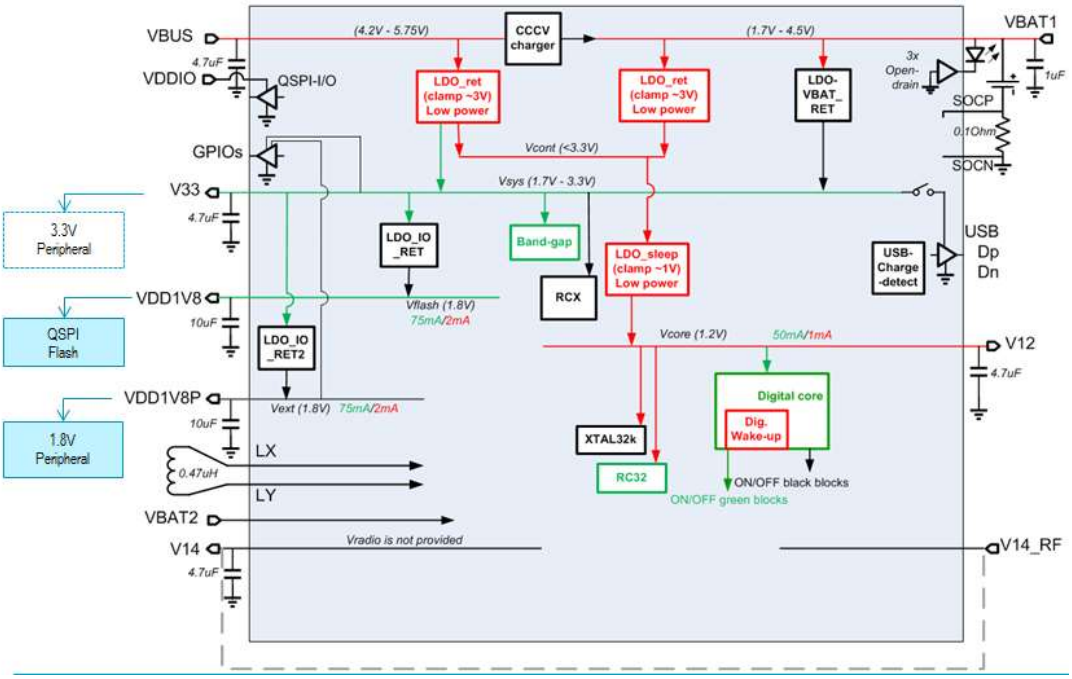

Figure 39: Recommended Power configuration

The SmartSnippets™ DA1468x SDK provides the following configuration settings:

Table 29: Configuration settings

Available configuration settings	Description
dg_configSET_RECHARGE_PERIOD	This is the period of the Sleep Timer that is used to bring-up part of the system periodically to resample the bandgap voltage or to restore the energy of the inductor of the DCDC.
dg_configPOWER_FLASH	If set to '1' then the 1.8V for the external QSPI Flash is supplied.
dg_configFLASH_POWER_DOWN	If set to '1' then the QSPI Flash is put to "Power Down" for the duration of the sleep period.
dg_configFLASH_POWER_OFF	If set to '1' then the QSPI Flash is powered-off during sleep (the 1.8V is turned-off during sleep). Note that dg_configFLASH_POWER_DOWN has priority over dg_configFLASH_POWER_OFF. If both are set, then the QSPI Flash will be put to "Power Down" mode while sleeping.
dg_configPOWER_EXT_1V8_PERIPHERALS	If set to '1' then the 1.8V for the external peripherals is supplied.

9.4.2 System Clock

The clock tree diagram of the DA1468x is depicted in Figure 40.

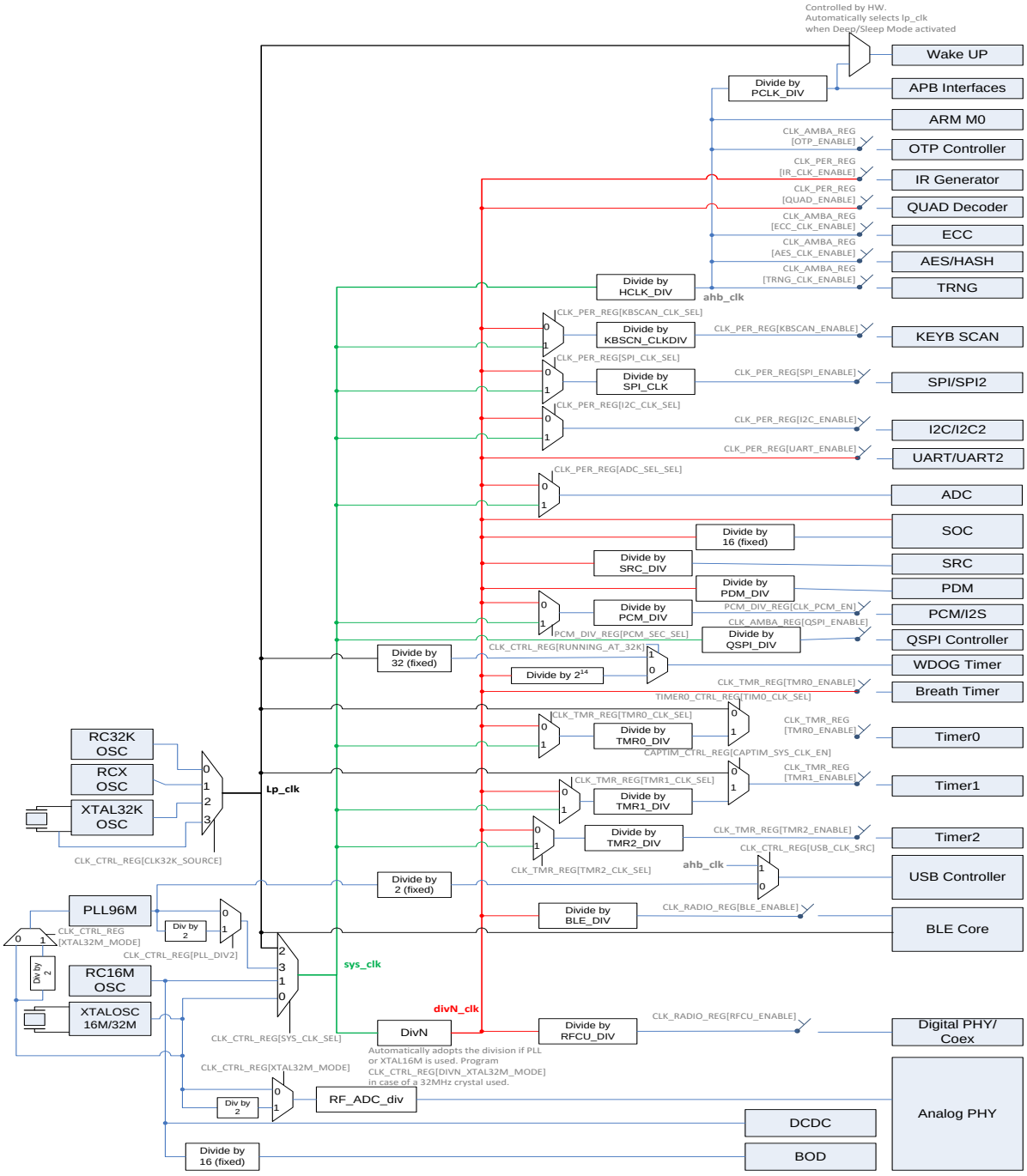

Figure 40: Clock tree diagram

The clock manager, which is part of the CPM, has the following characteristics:

- Performs the clock initialization of the system after boot.
- Controls only the “system level” clocks. The driver of each hardware resource (or the Adapter, if one exists) is responsible for controlling the clock setting for the resource. In this context, the CPM controls the “system clock”, which is the green line in the figure above, the AHB and APB clocks, which are the blue lines at the top of the figure, and the low power clock, which is the black line.

- After wake-up, the system runs by default using the RC16 clock. The clock manager switches back to the last configuration set by the application (system clock type and divider, and AHB and APB dividers) either immediately if possible, or after the XTAL16M has settled. This procedure is transparent to the application tasks. The CPM unblocks any task that has blocked waiting for the high precision clock.
- Handles requests to switch to another clock configuration. A request may be denied if this switch affects a hardware resource that is active (i.e. a hardware timer).
- Conditionally lowers the clocks (using the 1:N dividers for the system clock and the AHB and APB clocks) automatically when the system enters Idle mode. This procedure is not performed if the lower clock frequencies affect a hardware resource that is active.
- Hides the complexity of the clock tree from the applications by offering a unified API for clock control.
- Offers the application tasks the ability to switch to another clock configuration during runtime, if possible. The low power clock cannot be changed during runtime.

The clock manager API consists of the following functions:

Table 30: Functions in Clock Manager API

Function	Description
`bool cm_sys_clk_set(sys_clk_t type)`	Set the system clock. The available options are: `RC16`, `XTAL16M` (or `XTAL32M`), `PLL48` and `PLL96`. The low power clock cannot be set as the system clock.
`bool cm_cpu_clk_set(cpu_clk_t clk)`	Set the system clock and the `AHB` divisor such that the requested clock frequency is achieved.
`void cm_apb_set_clock_divider(apb_div_t div)`	Set the clock divisor for the `APB` clock. The actual frequency depends on the system clock used.
`bool cm_ahb_set_clock_divider(ahb_div_t div)`	Set the clock divisor for the `AHB` clock. The actual frequency depends on the system clock used.
`_get_ and _fromISR`	variants of the above `_set_` functions are also available.
`void cm_lp_clk_init(void)`	Initialize the Low Power clock.
`bool cm_sys_clk_set(sys_clk_t type)`	Set system clock.
`bool cm_lp_clk_is_avail(void)`	Check if the Low Power clock is available.
`void cm_clk_init_low_level(void)`	Execute clock initialization after power-up.
`void cm_sys_clk_init(sys_clk_t type)`	Execute clock initialization after the OS has started.

9.4.2.1 XTAL32M support

Note 8 XTAL32M support is available only for DA14683 devices – it is not supported for DA14681 devices.

In order to set XTAL32M clock as the main system clock user must follow next steps:

1. Update clock configuration in the application's (e.g. pxp_reporter) *custom_config_qspi.h* header file by defining:

```
#define dg_configEXT_CRYSTAL_FREQ        EXT_CRYSTAL_IS_32M
```


2. Update clock initialization in the application's main function (in `main.c` file). The following functions must be altered as:

```
cm_sys_clk_init(sysclk_XTAL32M)

cm_sys_clk_set(sysclk_XTAL32M)
```

9.5 Charger configuration

Three different charging configurations are currently supported by the SmartSnippets™ SDK. More details on charging can be found in [13]. The available configuration settings for the charger are:

- No Charging
- Charging with default parameters
- Charging with custom parameters

The configuration settings for the integrated charger of Li-ion batteries can be found in `sdk/config/bsp_defaults.h`. These parameters should not be modified in-place but overridden in the project specific `config/custom_config_qspi.h` folder if needed. Table 31 shows the available settings for the charger.

Table 31: Configuration settings for integrated charger of Li-ion batteries

Configuration settings	Description
dg_configUSE_USB_CHARGER	It enables / disables the use of the Charger from the application.
dg_configUSE_USB_ENUMERATION	It controls whether enumeration with the USB Host will take place or not.
dg_configALLOW_CHARGING_NOT_ENUM	It controls whether the Charger will start charging using charge current up to 100mA until the enumeration completes.
dg_configUSE_NOT_ENUM_CHARGING_TIMEOUT	According to the USB Specification, there is a time limit that a device, which is connected to the USB bus but not enumerated, can draw power. This configuration setting controls whether the Charger will respect this time limit or not.
dg_configPRECHARGING_INITIAL_MEASURE_DELAY	This is the time to wait before doing the first voltage measurement after starting pre-charging. This is to ensure that an initial battery voltage overshoot will not trigger the Charger to stop pre-charging and move to normal charging.
dg_configPRECHARGING_THRESHOLD	The voltage threshold below which pre-charging starts.
dg_configCHARGING_THRESHOLD	The voltage threshold at which pre-charging stops and charging starts.
dg_configPRECHARGING_TIMEOUT	The maximum time that pre-charging will last. If the dg_configCHARGING_THRESHOLD is not met within this period then charging is stopped.
dg_configCHARGING_TIMEOUT	The maximum time that charging will last. This setting covers both the CC and CV phases of charging.
dg_configCHARGING_CC_TIMEOUT	The maximum time that the charging hardware will stay in the CC phase. If this period elapses and the charging phase is still CC then charging stops.

Configuration settings	Description
dg_configCHARGING_CV_TIMEOUT	The maximum time that the charging hardware will stay in the CV phase. If this period elapses and the charging phase is still CV then charging stops.
dg_configUSB_CHARGER_POLLING_INTERVAL	While being attached to a USB cable and the battery has been charged, this is the interval that the VBAT is polled to decide whether a new charge cycle will be started.
dg_configBATTERY_CHARGE_GAP	This is the safety limit used to check for battery overcharging.
dg_configBATTERY_REPLENISH_GAP	This is the threshold below the maximum voltage level of the battery where charging will be restarted to keep the battery fully charged.
dg_configBATTERY_TYPE	This is the battery type that is used in the system. Valid options are BATTERY_TYPE_LICOO2, BATTERY_TYPE_LIMN2O4, BATTERY_TYPE_LIFEPO4, BATTERY_TYPE_LINICOAIO2 (charging voltage for all the options is 4.2V), BATTERY_TYPE_CUSTOM (charging voltage dg_configBATTERY_TYPE_CUSTOM_ADC_VOLTAGE) and BATTERY_TYPE_NO_RECHARGE.
dg_configBATTERY_TYPE_CUSTOM_ADC_VOLTAGE	In case of a custom battery, this parameter must be defined to provide the charging voltage level of the battery (in ADC measurement units). It is used by the charger to check if the battery is charged before starting charging, possible over-charging etc.
dg_configBATTERY_CHARGE_VOLTAGE	This is the charging voltage setting for the charger hardware. See [1], CHARGER_CTRL1_REG:CHARGE_LEVEL description for more details.
dg_configBATTERY_CHARGE_CURRENT	This is the charging current setting for the charger hardware. See [1], CHARGER_CTRL1_REG:CHARGE_CUR description for more details.
dg_configBATTERY_PRECHARGE_CURRENT	This is the pre-charging current setting for the charger hardware. The correlation of settings between the configured value and the current is shown in the Table 33.
dg_configBATTERY_LOW_LEVEL	If not zero, this is the lowest allowed limit of the battery voltage. If VBAT drops below this limit, the system enters hibernation mode.
dg_configBATTERY_CHARGE_NTC	It controls whether the thermal protection will be enabled or not. Using this requires an external thermistor.

9.5.1 No Charging

To enable the “No Charging” configuration, the user has to set dg_configUSE_USB_CHARGER = 0 in the project’s config/custom_config_qspi.h file.

Note 9 It is important to use this configuration when no battery is attached to avoid any unwanted behavior.

9.5.2 Default Charging

If no custom parameters are defined in the project's `config/custom_config_qspi.h` file then the default ones will be used. The default configuration settings for the charger are shown in Table 32.

Table 32: Charging with default parameters

Configuration settings	Description
dg_configPRECHARGING_THRESHOLD	2462 (3.006V)
dg_configBATTERY_TYPE	BATTERY_TYPE_CUSTOM && dg_configBATTERY_CHARGE_VOLTAGE = 0xA (4.2V)
dg_configBATTERY_PRECHARGE_CURRENT	18
dg_configCHARGING_THRESHOLD	2498 (3.05V)
dg_configBATTERY_CHARGE_CURRENT	2 (30mA)
dg_configBATTERY_CHARGE_NTC	1 (disabled)
dg_configPRECHARGING_TIMEOUT	30 * 60 * 100 (30min)

9.5.3 Custom Charging parameters

The user may also define custom parameters for the charger. For example, the `pxp_reporter` demo application uses custom charging parameters, as shown in Code 16: Charging with custom parameters. These parameters are defined in the application's `config/custom_config_qspi.h` file.

```
#define dg_configBATTERY_TYPE                    (BATTERY_TYPE_CUSTOM)
#define dg_configBATTERY_CHARGE_VOLTAGE          0xA     // 4.2V
#define dg_configBATTERY_TYPE_CUSTOM_ADC_VOLTAGE         (3439)
#define dg_configPRECHARGING_THRESHOLD           (2462)  // 3.006V
#define dg_configCHARGING_THRESHOLD              (2498)  // 3.05V
#define dg_configBATTERY_CHARGE_CURRENT          4       // 60mA
#define dg_configBATTERY_PRECHARGE_CURRENT       20      // 2.1mA
#define dg_configBATTERY_CHARGE_NTC              1       // disabled
#define dg_configPRECHARGING_TIMEOUT             (30 * 60 * 100)  // N x 10msec

#define dg_configUSE_USB                         1
#define dg_configUSE_USB_CHARGER                 1
#define dg_configALLOW_CHARGING_NOT_ENUM         1
#define dg_configUSE_NOT_ENUM_CHARGING_TIMEOUT   0
```

Code 16: Charging with custom parameters

Table 33: Pre-charging current settings

dg_configBATTERY_PRECHARGE_CURRENT setting	Pre-charging current (mA)
16	Reserved
17	Reserved
18	1
19	1.5
20	2.1
21	3.2
22	4.3

dg_configBATTERY_PRECHARGE_CURRENT setting	Pre-charging current (mA)
23	Reserved
24	6.6
25	7.8
26	Reserved
27	11.3
28	13.3
29	15.3

9.5.4 Charger configuration process

Charger configuration (process) description can be divided into three distinct parts. The first is related to the USB configuration, the second to the charging algorithm and the third to the actual charging parameters.

The first part of the Charger configuration process depends on the application capabilities. In Table 34 below, two typical configurations are listed, one when enumeration is not supported and a second when it is supported, i.e. the application includes a USB driver.

Table 34: Charger - Configuration settings for the USB interface

Without enumeration	
dg_configUSE_USB_CHARGER	1
dg_configUSE_USB_ENUMERATION	0 (or left to the default value)
dg_configALLOW_CHARGING_NOT_ENUM	1 This will be the most common setting as it offers the option to charge from an SDP port.
dg_configUSE_NOT_ENUM_CHARGING_TIMEOUT	0 May be set to 1 if adhering to the USB specification is mandatory. Even if it is left as 0 and the SDP port shuts down the power after 45 minutes, the charging will simply stop.
With enumeration	
dg_configUSE_USB_CHARGER	1
dg_configUSE_USB_ENUMERATION	1
dg_configALLOW_CHARGING_NOT_ENUM	1
dg_configUSE_NOT_ENUM_CHARGING_TIMEOUT	0

The configuration of the charging algorithm is more complex as it requires the setting of various voltage levels in ADC measurement units[1]. In Table 35 below, a typical configuration for the charging algorithm is listed.

The mathematical formula used for converting Vbat to ADC is the following:

```
y[ADC units] = (4095 * Vbat[Volts]) / 5)
```

[1] For a Spreadsheet-based tool that helps calculate the voltage from the ADC value, please contact Dialog customer support.

Table 35: Charger - Configuration settings for the charging algorithm

Configuration settings	Values
dg_configPRECHARGING_INITIAL_MEASURE_DELAY	Undefined (the default setting is used)
dg_configPRECHARGING_THRESHOLD	2462 (3.006V)
dg_configCHARGING_THRESHOLD	2498 (3.05V)
dg_configPRECHARGING_TIMEOUT	30 * 60 * 100 (30 minutes, the default setting is 15 minutes)
dg_configCHARGING_CC_TIMEOUT	120 * 60 * 100 (2 hours, the default setting is 3 hours)
dg_configCHARGING_CV_TIMEOUT	180 * 60 * 100 (3 hours, the default setting is 6 hours)
dg_configUSB_CHARGER_POLLING_INTERVAL	1 * 60 * 100 (1 minute, the default setting is 1 second)
dg_configBATTERY_CHARGE_GAP	Undefined (the default setting of 0.1V is used)
dg_configBATTERY_REPLENISH_GAP	Undefined (the default setting of 0.2V is used)

The final part of the Charger configuration depends on the characteristics of the battery that is used in the system. Let's consider a battery that has the charging profile which is depicted in Figure 41 with a charging voltage of 4.35V. There are three phases to the charging process.

- Pre-Charge

  Below 3.0V pre-charging with a very low charging current of 2.1mA is required until the 3.0V level is reached.

- Constant Current (CC)

  The normal charging current of 30mA is applied in the Constant Current phase of the charging until the voltage reaches the charging voltage.

- Constant Voltage (CV)

  The charging current is gradually reduced to keep the normal charging voltage on the battery. Charging is considered completed when the charging current drops to 10% of the nominal value, i.e. 3mA.

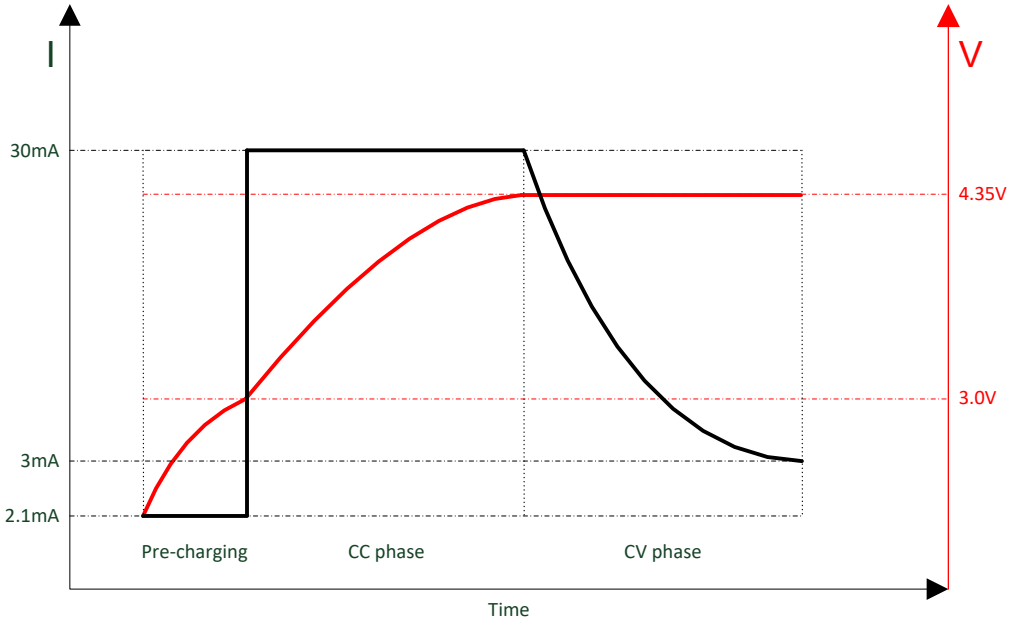

Figure 41: Battery charging profile

The configuration of the charger for this specific battery is listed in Table 36.

Table 36: Charger – configuration settings for a specific battery

Configuration settings	Values
dg_configBATTERY_TYPE	BATTERY_TYPE_CUSTOM (Since this Li-ion battery has a charging voltage level other than 4.2V, this is a custom battery.)
dg_configBATTERY_TYPE_CUSTOM_ADC_VOLTAGE	3562 (4.35V)
dg_configBATTERY_CHARGE_VOLTAGE	0xD (the hardware setting for 4.35V)
dg_configBATTERY_CHARGE_CURRENT	2 (the hardware setting for 30mA)
dg_configBATTERY_PRECHARGE_CURRENT	20 (the setting for 2.1mA)
dg_configBATTERY_LOW_LEVEL	2496 (3.05V)

9.5.5 Issues for non-rechargeable batteries

If the user uses a non-rechargeable battery, header files must be modified. In that case please contact Dialog Customer Support.

If the user uses a USB charger with an invalid battery type such as BATTERY_TYPE_NO_RECHARGE or BATTERY_TYPE_NO_BATTERY the compilation will be aborted with an error. When using a non-rechargeable battery, the Hibernation option is disabled because of low voltage detection.

9.5.6 Charger related callback functions

Table 37 below refers to charger related callback functions which can be found in the *sys_charger.c* file located in `<sdk_root_directory>\sdk\bsp\system\sys_man`. These functions are defined as weak (`__attribute__((weak))`).

Note 10 These call-backs may be implemented by the application code in order to catch events sent by the USB charger. Note that the USB charger task is started before the application task. Thus, these call-backs may be called before the application task is started. The application code should handle this case, if there is a need. For example, a possible implementation of the `usb_attach_cb()` call-back is shown in Code 17:

```
/*
 * PRIVILEGED_DATA bool app_task_is_initialized;
 *
 * enum RCV_USB_INDICATIONS {
 *         RCV_USB_NONE,
 *         RCV_USB_ATTACH,
 *         ...
 * } app_usb_indication;
 *
 * void usb_attach_cb(void)
 * {
 *         if (app_task_is_initialized) {
 *                 // Do something
 *         } else {
 *                 // Raise a flag for the app to check when started.
 *                 app_usb_indication = RCV_USB_ATTACH;
 *         }
 * }
 *
 * void app_task(void *pvParameters)
 * {
 *         ...
 *         switch (app_usb_indication) {
 *         case RCV_USB_NONE:
 *                 break;
 *         case RCV_USB_ATTACH:
 *                 ...
 *                 break;
 *         ...
 *         default:
 *                 break;
 *         }
 *
 *         app_usb_indication = RCV_USB_NONE;
 *         app_task_is_initialized = true;
 * }
 *
 */
```

Code 17: Callback function example to catch events sent by the USB-charger

Table 37: Charger related callback functions

Function	Description
`usb_attach_cb()`	Callback function used to notify the application task that the usb cable has been attached.
`usb_detach_cb()`	Callback function used to notify the application task that the usb cable has been detached.
`usb_start_enumeration_cb()`	Callback function used to notify the application task

Function	Description
	that the charger will start to enumerate if possible. This means that the dg_configUSE_USB_ENUMERATION flag has been set to 1. This also means that the dg_configUSE_USB is set to 1 and the USB interface will be used for data transfers.
usb_charging()	Callback function used to notify the application task that the charger has started the charging procedure according to the predefined settings.
usb_precharging()	Callback function used to notify the application task that pre-charging has started (the pre-charging current has been set). The charging state is set to USB_PRE_CHARGING_ON.
usb_precharging_aborted()	Callback function used to notify the application task that pre-charging has stopped. It means that the charging state has been set to USB_CHARGING_BLOCKED. This could happen if after a period, defined by the dg_configPRECHARGING_TIMEOUT parameter, the Vbat is not higher than 3.0 V.
usb_charging_stopped()	Callback function used to notify the application task that stop of battery charging has been detected. The charging state will be set to USB_CHARGING_OFF afterwards.
usb_charging_aborted()	Callback function used to notify the application task that the charging process has been aborted. The charger state has been set to USB_CHARGER_ATTACHED.
usb_charging_paused()	Callback function used to notify the application task that the charging process has been paused. It means that the charger state is set to USB_CHARGER_PAUSED.
usb_charged()	Callback function used to notify the application task that the charging of the battery (Li-ion) has ended. The charger state has been set to USB_CHARGER_ATTACHED.
usb_charger_battery_full()	Callback function used to notify the application task that the charging has stopped because vbat_level has reached the defined level. The charging state has been set to USB_CHARGING_OFF.
usb_charger_bad_battery()	Callback function used to notify the application task that there is a problem with the vbat_level of the battery. The charging state has been set to USB_CHARGING_BLOCKED.
usb_charger_temp_low()	Callback function used to notify the application task that the charging process has been aborted. This caused by a persisting error generated due to the fact that the battery temperature is too low.
usb_charger_temp_high()	Callback function used to notify the application task that the charging process has been aborted. This caused by a persisting error generated due to the fact that the battery temperature is too high.
usb_is_suspended()	Callback function used to notify the application task that the USB has been suspended.
usb_is_resumed()	Callback function used to notify the application task that the USB has been resumed. The system clock and the IRQs have been resumed while the charger state

Function	Description
	has been set to `USB_CHARGER_ATTACHED`.

9.6 Watchdog Service

9.6.1 Description

The system watchdog service (`sys_watchdog`) has been designed to monitor system tasks and avoid system freezes. The interaction of this service with other parts of the system is shown in Figure 42:

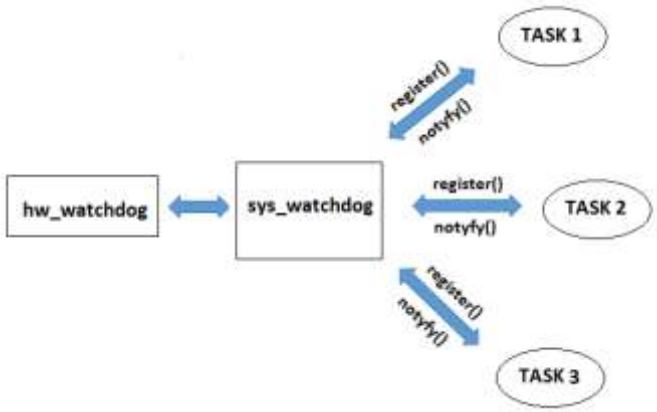

Figure 42: Watchdog overview

Effectively, `sys_watchdog` is a layer on top of the watchdog low-level driver that allows multiple tasks to share the underlying hardware watchdog timer. The watchdog service can be used to trigger a full system reset. This will allow system to recover from a catastrophic failure in one or more tasks.

9.6.2 Concept

A task that should be monitored has to first register itself with this service to receive a unique handle *(id)*. The task must then periodically notify `sys_watchdog` using this id, to signal that the task is working properly. In case of any error during registration, the invalid id -1 is returned.

The hardware watchdog is essentially a countdown timer, which will trigger a full system reset if it expires. To prevent this, the watchdog timer must be reset to its starting value before it expires. This starting value can be configured in the application custom configuration files via the numerical macro `dg_configWDOG_RESET_VALUE`. Its default value is the maximum `0xFF`, which corresponds to approximately 2.6 seconds (the time unit is 10.24 msec). The maximum number of tasks that can be monitored is defined by the configuration macro `dg_configWDOG_MAX_TASKS_CNT` (the absolute maximum is 32).

If during one watchdog period all monitored tasks notify `sys_watchdog`, the hardware watchdog will be updated via the `hw_watchdog_set_pos_val()` LLD API function; in this case, no platform reset will be triggered for this watchdog period. However, a platform reset will be triggered if at least one task does not notify `sys_watchdog` in time. There are two ways for a task to notify `sys_watchdog`.

Each task is responsible for periodically notifying `sys_watchdog` that it is still running using `sys_watchdog_notify()`. This must be done before the watchdog timer expires. Occasionally a registered task may want to temporarily exclude itself from being monitored if it expects to be blocked for a long time waiting for an event. This is done using the `sys_watchdog_suspend()` API function. This function suspends monitoring of specific tasks in `sys_watchdog`, as there is no need to monitor a task that is blocked waiting for an event that might take too long to occur (i.e. it would lead to the task

failing to notify the watchdog service, thus resulting in a system reset). When the task is unblocked, the `sys_watchdog_resume()` API function should be called to restore task monitoring by the watchdog service. From that moment on the task shall notify the watchdog service as usual.

Finally, the `sys_watchdog_set_latency()` API function is intended to be used in cases where a task would require a watchdog period greater than the configured watchdog timer reset value. Using this API allows a task to delay notification of `sys_watchdog` for a given number of watchdog periods, without triggering a system reset. The effect of calling the API function is one-off, thus it must be set every time increased latency is required.

9.6.3 Examples

To register the task with `sys_watchdog` use the following code snippet.

```
/* registration pxp task to be monitored by watchdog */
        wdog_id = sys_watchdog_register(false);
```

Code 18: Notify sys_watchdog of the task

To notify `sys_watchdog` use `sys_watchdog_notify()`. If the task is going to suspend for an event then temporarily exclude the current task from being monitored using `sys_watchdog_suspend()`. Once the task has received an event it can resume its watchdog operation with `sys_watchdog_resume()`. This flow is shown below:

```
/** notify watchdog on each loop since there's no other trigger for this - monitoring
* will be suspended while blocking on OS_TASK_NOTIFY_WAIT()
*/
    sys_watchdog_notify(wdog_id);
 /*
  * Wait on any of the event group bits, then clear them all
  */
sys_watchdog_suspend(wdog_id);
ret = OS_TASK_NOTIFY_WAIT(0, OS_TASK_NOTIFY_ALL_BITS, &notif, OS_TASK_NOTIFY_FOREVER);
                /* Blocks forever waiting for the task notification.
               Therefore, the return value must
                  * always be OS_OK
                  */
                 OS_ASSERT(ret == OS_OK);
                 sys_watchdog_resume(wdog_id);
```

Code 19: Using sys_watchdog while suspending task for an event

9.6.4 API

Table 38: Configuration functions for sys_watchdog

Function	Description
`void sys_watchdog_init(void)`	Initialize sys_watchdog service. This should be called before using the sys_watchdog

Function	Description
	service, preferably at application startup.
`int8_t sys_watchdog_register(bool notify_trigger)`	Register current task with the `sys_watchdog` service. Returned identifier shall be used in all other `sys_watchdog` API calls from current task. Once registered, the task shall notify `sys_watchdog` periodically using `sys_watchdog_notify()` to prevent watchdog expiration. It is up to each task how this is done, but a task can request that it will be triggered periodically using the task notification capability, to notify `sys_watchdog` back as a response.
`void sys_watchdog_unregister(int8_t id)`	Unregister task from the `sys_watchdog` service.
`void sys_watchdog_suspend(int8_t id)`	Suspend task being monitoring by the `sys_watchdog` service. A monitor-suspended task is not unregistered entirely, but it is ignored by the watchdog service until its monitoring is resumed. It is faster than unregistering and registering the task again.
`void sys_watchdog_resume(int8_t id)`	Resume monitoring of a task by the `sys_watchdog` service. It should be called as soon as the reason that `sys_watchdog_suspend()` was called is removed.
`void sys_watchdog_notify(int8_t id)`	Notify `sys_watchdog` module about task. A registered task shall call this API function periodically to notify `sys_watchdog` that it is alive. This should be done frequently enough to fit into the watchdog timer interval set by `dg_configWDOG_RESET_VALUE`.
`void sys_watchdog_set_latency(uint8_t id, uint8_t latency)`	Set watchdog latency for a task. This allows a task to miss a given number of notification periods to `sys_watchdog` without triggering a system reset. Once set, the task is allowed to not notify `sys_watchdog` for “latency” consecutive watchdog timer intervals as defined by `dg_configWDOG_RESET_VALUE`. This option can be used to facilitate the operation of code that is expected to remain blocked for long periods of time (i.e. computation). This value is set once and does not reload automatically, thus it shall be set every time increased latency is required.

10 System Memory

10.1 Random Access Memory

10.1.1 Code Location

The complete BLE stack (both the Controller and Host) is in the system's Read Only Memory (ROM) and it is executed from there as well.

Application code on the other hand may reside either in on-chip OTP or in external QSPI Flash. It can be executed either in place (cached mode) or be copied into RAM and then executed from there (mirrored mode).

10.1.1.1 Execution Modes

There are two execution modes for the application code:

(a) Cached Mode, where the application code is in the OTP or Flash and is executed in place using the 16KB cache RAM.

(b) Mirrored mode, where the entire application code is copied into the RAM from where it is executed.

Depending on the execution mode, the available RAM size for Application Data varies.

- In Cached Mode, code is executed in place from either OTP or Flash memory. There is 128KB of RAM available for data and any code that is moved there.
- In Mirrored Mode, both application code and data are eventually located in the platform's 144KB RAM. In this mode the 16KB cache RAM is added to the 128KB of normal RAM

Note 11 The SDK has been developed to support only cached mode from flash.

10.1.2 Data Heaps

All data variables are allocated from one of two memory heaps

10.1.2.1 Application Heap

The Application Heap is used to allocate memory for every RTOS task including the OS itself. Its size is dependent on the actual application and must be configured together with the RTOS configuration.

The retained variables that need to be maintained when the system is sleeping should be clearly declared using the `PRIVILEGED_DATA` or `INITIALISED_PRIVILEGED_DATA` attributes, which inform the linker that these elements must be placed in the data sections that will be retained when the system goes to sleep.

10.1.2.2 BLE Stack Heap

The BLE Stack has its own dedicated Data Heap. Its size and retention scheme is pre-configured in the SmartSnippets™ DA1468x SDK and should not be changed by the application developer.

10.1.3 Optimal Memory Size

The Application heap and retained memory configuration is application-dependent and so must be optimized by the developer.

Optimizing Heap size is done empirically by configuring the system with a big heap and then measuring Heap ratio usage while executing final application. FreeRTOS provides some advanced methods of monitoring RTOS Heap usage during the development phase. Refer to section 13.2 for more details about the optimization of the Heap size.

10.2 Non-Volatile Memory Storage

The SmartSnippets™ DA1468x SDK defines a set of storage classification rules that enable the good design of memory storage requirements and memory budget estimation. It is essential for the developer to have a clear understanding of their requirements for the following elements that could use non-volatile storage:

- System Parameters that need to be stored in NVM (e.g. device address)
- Firmware upgrade (dual images)
- App-specific binaries (e.g. pre-recorded audio messages)
- Logging (e.g. event logs)
- Application data (e.g. sensor values, statistics, authentication keys)

For each storage type a corresponding, dedicated region is mapped in the Flash partition. Each region is identified by a partition ID. When the NVMS Adapter makes read/write accesses to storage it uses the partition ID and an offset. Additional details can be found in the NVMS Adapter section, 12.4.

The SmartSnippets™ DA1468x SDK defines the following memory partitions (in a non-SUOTA build) to manage the storage:

- (FW) Firmware Image Region
- (PARAMS) Parameters Region
- (BIN) Binaries Region
- (LOG) Logging of events or values
- (DATA) Generic data Region, Statistics etc

The exact Memory mapping depends on the actual Flash device (i.e. size, sector size) used on the board. It needs to be specified during compilation in `<sdk_root_directory>/config/1M/partition_table.h`. A default partition table is provided with the SmartSnippets™ DA1468x SDK which fits in DK QSPI Flash (1Mbytes with sectors of 4KB), the actual definition of which is shown in Code 20:

```
PARTITION2( 0x000000,0x07F000,NVMS_FIRMWARE_PART  ,0 )
PARTITION2( 0x07F000,0x001000,NVMS_PARTITION_TABLE,PARTITION_FLAG_READ_ONLY )
PARTITION2( 0x080000,0x010000,NVMS_PARAM_PART     ,0 )
PARTITION2( 0x090000,0x030000,NVMS_BIN_PART       ,0 )
PARTITION2( 0x0C0000,0x020000,NVMS_LOG_PART       ,0 )
PARTITION2( 0x0E0000,0x020000,NVMS_GENERIC_PART   ,PARTITION_FLAG_VES )
```

Code 20: Memory mapping

Other partition tables for different sized Flash (on a custom board) could be selected by adding one of these defines to `config/custom_config_qspi.h`

```
#define USE_PARTITION_TABLE_2MB
#define USE_PARTITION_TABLE_2MB_WITH_SUOTA
#define USE_PARTITION_TABLE_512K
#define USE_PARTITION_TABLE_512K_WITH_SUOTA
#define USE_PARTITION_TABLE_1MB_WITH_SUOTA
```


10.2.1 QSPI Flash Support

This section describes the QSPI Flash support in SmartSnippets™ DA1468x SDK and the steps required to add support for a new Flash memory.

The SDK supports by default three different flash types:

- Winbond: W25Q80EW, 8Mbit
- Gigadevice: GD25LQ80B, 8Mbit
- Macronix: MX25U51245G, 512 Mbit

These three devices have been tested with the SDK release using the modes of operation listed below. The default device is the Winbond W25Q80EW because this is the flash type that is mounted on the Pro and Basic DK boards. Another device can be selected by changing the macros shown in section 10.2.1.4.

Section 10.2.1.7 explains how to add other flash devices that have the same boot sequence as the three supported devices. The user will need to check carefully the Flash command set and verify correct read/write/erase operation at the desired clock speed.

Flash types with a different boot sequence can be used on the DA14681-SDK by modifying the QSPI Flash Initialization Section (QFIS) in the OTP of the DA1468x. This method is explained in [10].

10.2.1.1 Modes of operation and configuration

The SmartSnippets™ DA1468x SDK supports two modes of operation: Autodetect mode and Manual mode. The Autodetect mode can detect the flash type at runtime, while the Manual mode involves explicitly declaring the flash used in the project at compile time.

Note 12 The Manual Mode is the default and recommended mode. The Autodetect Mode will greatly increase code size and Retained RAM usage, and may prevent the project fitting in RAM.

10.2.1.2 Autodetect Mode

The Autodetect mode detects the flash that is used in runtime, and selects the proper flash driver to use. The Autodetect mode can only detect among the flash devices officially supported by the SmartSnippets™ DA1468x SDK. If no match is found, a default driver will be used (which may or may not work).

Since the Autodetect mode needs to select the driver to use in runtime, it has the code for all the drivers in the binary. It also keeps the selected driver's configuration parameters in Retained RAM. Therefore, the Autodetect mode is **NOT** recommended for production builds.

10.2.1.3 Manual Mode

The Manual mode simply consists of a hardcoded declaration of the flash driver to use. Therefore, only the code of the selected driver needs to be compiled in the binary, and there is no need to retain the driver parameters in Retained RAM, since the compiler optimizes them out. This mode is suitable for Production builds.

10.2.1.4 Flash Configuration

The Flash subsystem is configured using the macros shown in Table 39, which must be defined in the `config/custom_config_qspi.h` file of the project:

Table 39: Macros for the configuration of the Flash subsystem

qspi_flash_config_t field	Description
dg_configFLASH_AUTODETECT	Default: 0. This macro, if set, enables the Autodetect Mode. Please note, that the use of this macro is NOT recommended.

qspi_flash_config_t field	Description
dg_configFLASH_HEADER_FILE	○ This macro must be defined as a string named after the header file to use for the specific flash driver. E.g. qspi_w25q80ew.h, qspi_gd25lq80b.h, qspi_mx25u51245.h.This header file must be either one of the qspi_<part_nr>.h header files found in <sdk_root_directory>\sdk\bsp\memory\include, or a header file under the project's folder, as long as this path is in the compiler's include path (see the document section 10.2.1.7 about adding support for new flash devices).
dg_configFLASH_MANUFACTURER_ID	This macro must be defined to the Flash manufacturer ID, as defined in the respective driver header file (e.g. WINBOND_ID, GIGADEVICE_ID, MACRONIX_ID).
dg_configFLASH_DEVICE_TYPE	This macro must be defined to the corresponding device type macro, as defined in the driver header file (e.g. W25Q80EW, GD25LQ_SERIES, MX25U_MX66U_SERIES).
dg_configFLASH_DENSITY	This macro must be defined to the corresponding device density macro, as defined in the driver header file (e.g. W25Q_8Mb_SIZE, GD25LQ80B_SIZE, MX25U51245_SIZE).

When the system is in Manual Mode (dg_configFLASH_AUTODETECT == 0), which is the default, all the three macros above are defined in sdk/config/bsp_defaults.h to enable the default flash used, which is the Winbond W25Q80EW.

10.2.1.5 Code Structure

The QSPI Flash access functionality is implemented in qspi_automode.c and qspi_automode.h file. Common command definitions and functions needed for all devices are declared in qspi_common.h. Device specific code is defined in header files named as qspi_<flash device name>.h.

The code in qspi_automode.c (and in some other parts of the SmartSnippets™ DA1468x SDK, as well), calls device-specific functions and uses device-specific values to properly initialize the flash device. Each driver header file provides an instance of the structure qspi_flash_config_t to the main driver, containing all the device-specific function pointers and variables.

10.2.1.6 The flash configuration structure qspi_flash_config_t

Each driver header file must provide its own instance of qspi_flash_config_t. Please note that this instance must be named with a unique name, like flash_<device name>_config, since all the device header files are included in the qspi_automode.c file. Therefore, there is a single global namespace. Also, please note that the struct instance must be declared as const so that the compiler can optimize references to it.

The qspi_flash_config_t structure, as shown in Table 40, has the following fields (see <sdk_root_directory>/sdk/memory/include qspi_common.h for more information):

Table 40: The qspi_flash_config_t structure

qspi_flash_config_t field	Description
Initialize	Pointer to the flash-specific initialization function.
is_suspended	Pointer to a flash-specific function that checks if flash is in erase/program suspend state.

qspi_flash_config_t field	Description
deactivate_command_entry_mode	Pointer to a flash-specific function that performs extra steps needed when command entry mode is deactivated.
sys_clk_cfg	Pointer to a flash-specific function that performs Flash configuration when system clock is changed (e.g. change dummy bytes or QSPIC clock divider).
get_dummy_bytes	Pointer to a flash-specific function that returns the number of dummy bytes currently needed (it may change when the clock changes).
manufacturer_id	The Flash JEDEC vendor ID (Cmd 0x9F, 1st byte). This and the device_type and device_density are needed for flash autodetection, when on Autodetect mode.
device_type	○ The Flash JEDEC device type (Cmd 0x9F, 2nd byte).
device_density	The Flash JEDEC device type (Cmd 0x9F, 3rd byte).
erase_opcode	The Flash erase opcode to use.
erase_suspend_opcode	The Flash erase suspend opcode to use.
erase_resume_opcode	The Flash erase resume opcode to use.
page_program_opcode	The Flash page program opcode to use.
quad_page_program_address	If true, the address will be transmitted in QUAD mode when writing a page. Otherwise, it will be transmitted in serial mode.
read_erase_progress_opcode	The opcode to use to check if erase is in progress (Usually the Read Status Reg opcode (0x5).
erase_in_progress_bit	The bit to check when reading the erase progress.
erase_in_progress_bit_high_level	The active state (true: high, false: low) of the bit above.
send_once	If set to 1, the "Performance mode" (or “burst”, or “continuous”; differs per vendor) will be used for read accesses. In this mode, the read opcode is only sent once, and subsequent accesses only transfer the address.
extra_byte	The extra byte to transmit, when in "Performance mode" (send once is 1), that tells the flash that it should stay in this continuous, performance mode.
address_size	Whether the flash works in 24- or 32-bit addressing mode.
break_seq_size	Whether the break sequence, that puts flash out of the continuous mode, is one or two bytes long (the break byte is 0xFF).
ucode_wakeup	The QSPIC microcode to use to setup the flash on wakeup. This is automatically used by the QSPI Controller after wakeup, and before CPU starts code

qspi_flash_config_t field	Description
	execution. This is different based on whether flash was active, in deep power down or completely off while the system was sleeping.
power_down_delay	This is the time, in usec, needed for the flash to go to power down, after the Power Down command is issued.
release_power_down_delay	This is the time, in usec, needed for the flash to exit the power down mode, after the Release Power Down command is issued.

In Autodetect mode, these structures reside in the .rodata section of the code. As soon as the flash subsystem is initialized, it reads the flash JEDEC ID (command 0x9F) to find out which is the actual flash device that is used. It then uses the JEDEC ID to select the corresponding flash_<flash device>_config structure, and copies it in the Retained RAM. It then uses it for all the flash operations that need it.

When in Manual mode, no JEDED ID is read and no copy is performed to the Retained RAM. Instead, the constant pointer flash_config is directly initialized (inside the flash-specific driver file) to the specific (and constant) flash_<device name>_config structure. The compiler then optimizes out the entire structure.

10.2.1.7 Adding support for a new flash device

The SmartSnippets™ DA1468x SDK driver subsystem currently supports a specific set of QSPI flash devices. It provides, however, the capability to add support for other flash devices as well.

Each device driver must have its own header file that should be named qspi_<device name>.h. The programmer can either use the qspi_XXX_template.h, or start from an existing driver file.

The new flash driver file should be placed inside the project's path, in a folder that is in the compiler's include path (an obvious choice is the config/ folder, but others can be used as well). This is recommended so that potential SDK upgrades will not interfere with the project-specific flash driver implementation.

Common code among flash families or vendors can be factored out in common header file per family/vendor. There are currently such common header files, like qspi_macronix.h and qspi_winbond.h. However, this is NOT necessary; moreover, it is the responsibility of the device driver header file to include the common header file, if needed.

Note 13 A custom flash driver can ONLY be used in Manual mode, which means that the macros described in Table 39 MUST be defined in config/custom_config_qspi.h.

The following steps are usually needed to create the new flash driver:

1. Copy and rename the template header file, or an existing driver file.
2. Rename all the functions and variables appropriately. It is important to remember that all the drivers reside in the same namespace and so all function and variable names must be unique.
3. Define the proper JEDEC ID values for the Manufacturer code, the device type and the device density
4. Verify that the suspend, resume, exit power-down, enter power-down, fast read, write enable, read status register are valid for the new device type.
5. Guard the header file using an #if preprocessor macro that checks for the specific driver selection.
6. Define any other driver-specific macros that are needed (like timings etc).
7. Define the constant wakeup microcode arrays that will be needed, per configuration mode that will be supported (dg_configFLASH_POWER_OFF, dg_configFLASH_POWER_DOWN or none of them). The microcode will be copied during the driver initialization in a special memory in the QSPI

controller, and will be used after system wakeup to initialize the QSPI (since the CPU isn't yet running code at this time). Please see [1] for the uCode format.

8. Declare the constant struct instance of type `qspi_flash_config_t`, named `flash_<device name>_config`, and initialize it with proper values. Please note that this must be declared as const.
9. Extend the function flash `<device name>_initialize()` if needed, e.g. to write some special QSPI configuration registers or the QUAD enable bit. Otherwise, leave empty.
10. Extend the function `flash_<device name>_sys_clock_cfg()` if needed. This can include modifying the dummy bytes when the system (and hence the QSPI) clock changes, or changing the QSPI clock divider (if, for example, the flash device cannot cope with 96MHz). Otherwise, leave empty.
11. The function `is_suspended()` should read the flash Status Register and return true if Erase or Write is suspended on the device.
12. If Continuous Read Mode (sometimes referred to as Performance or Burst Mode) is used, make sure to set `send_once` to 1, and set `extra_byte` to a proper value for the flash to keep working in this mode. This is flash-specific.
13. If the flash supports 32-bit addressing (e.g. the Macronix MX25U51245G Flash), make sure to use the proper uCode for wakeup. Also set `page_program_opcode`, `erase_opcode`, `break_seq_size` (this should also take into consideration whether the device will be working in Continuous Read mode as well) and `address_size`.
14. If the address, during write, will be provided in QUAD mode, set `quad_page_program_address` to true.

Note 14 The SmartSnippets™ DA1468x SDK supports reading in QUAD I/O mode (where the address and data are read in QUAD mode, and only the command is transferred in serial mode), both in Continuous Read and normal mode.

To test the flash driver, use the PXP Reporter demo application, and configure the new flash driver in its `custom_config_qspi.h` and `custom_config_qspi_suota.h` files. Do the following tests:

1. Verify that the application boots by using SmartSnippets™ Studio Power Profiler and a cell phone to connect to the device
2. Verify that the application continues working after the system starts going to sleep (after ~8 seconds), that the cell phone can connect to the device and that it can maintain the connection for a while.
3. Repeat steps 1 and 2 by changing the application clock to 96 MHz (change sysclk_XTAL16M in main.c to sysclk_PLL96)
4. Repeat steps 1, 2 and 3 and change `dg_configPOWER_1V8_SLEEP` to 0 (in flashes where this makes sense) and (separately), `dg_configFLASH_POWER_DOWN` to 1, to test the supported wake up sequence driver modes.
5. Repeat steps 1 through 4 using the SUOTA Configuration of the PXP Reporter application. This will test the write/erase functionality of the driver.

10.2.1.8 Working with a new flash device

Read/Erase/Program of new QSPI flash devices should be done using SmartSnippets™ Studio standard procedure (check "General Installation and Debugging Procedure" in SDK's Doxygen documentation).

Before working with new QSPI flash devices the following steps are required:

- Support for the new flash is added to the SDK as described in "Adding support for a new flash device" paragraph.
- SDK `uartboot` project (the secondary bootloader used by SDK flash programming tools) is built with support for the new QSPI flash as described in "Flash configuration" paragraph.

- SDK `cli_programmer` project (the tool used in the SDK for accessing flash) is re-built in `Release_static` (if working in Linux) or `Release_static_win` (if working in Windows) configuration, as described in SDK's Doxygen documentation "CLI programmer application" paragraph.

Note 15 SmartSnippets™ Toolbox only supports read/erase/programming of the default supported QSPI flash devices. Therefore, it is not recommended to be used with new flash devices.

11 Operation modes and startup procedure

The SmartSnippets™ DA1468x SDK supports two operational modes that correspond to the configurations described in section 10.1.1.1. They are:

Table 41: Operation modes

Operation mode	Code location	Cache enabled	Description
RAM	RAM	No	Program is loaded directly to RAM.
Flash cached	Flash (quad SPI mode)	Yes	Program runs in-place from QSPI flash. The first 0x100 – flash image header size bytes are copied to RAM by the boot loader.

11.1 Generated ELF file

After the project code is compiled, the linker generates an Extensible Linking Format (ELF) file based on the supplied linker scripts of each project (`sections.ld` and `mem.ld`). The following tables show the section structure (as obtained by the `readelf` utility with the `-s` option) for the three combinations of RAM mode with no Bluetooth low energy, flash cached mode with no Bluetooth low energy and flash cached mode with Bluetooth low energy.

The key difference with Bluetooth low energy support is the `jump_table_mem_area` section that holds pointers to callback functions used by the BLE ROM code.

`.copy.table` is used by the startup procedure in order to copy (load) code and data sections onto RAM. The table contains one or more section entries. The first 4 bytes in each entry are the section's source address, the next 4 bytes are the section's destination address in RAM and the last 4 bytes are the section's size. Usually there are two sections that are loaded onto RAM: the `.data` section and the `RETENTION_ROM0` section. `.zero.table` works in a similar manner, but it only writes zero's to memory areas with zero-initialized data. Its entries contain a destination address (4 bytes) and size (4 bytes). The sections that are initialized to zero are `.bss` and the zero-initialized part of `RETENTION_RAM0`.

Table 42: Example program sections in RAM operation mode

Nr	Name	Type	Addr	Off	Size	ES	Flag	Link	Info	Align
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	07fc0000	008000	0055c4	00	AX	0	0	16
[2]	.ARM.exidx	ARM_EXIDX	07fc55c4	00d5c4	000008	00	AL	1	0	4
[3]	.copy.table	PROGBITS	07fc55cc	00d5cc	000018	00	WA	0	0	1
[4]	.zero.table	PROGBITS	07fc55e4	00d5e4	000018	00	WA	0	0	1
[5]	.data	PROGBITS	07fd8000	010000	000064	00	WA	0	0	4
[6]	.bss	NOBITS	07fd8064	018064	00020c	00	WA	0	0	4
[7]	.heap	PROGBITS	07fd8270	018008	001c00	00		0	0	8
[8]	.stack_dummy	PROGBITS	07fd8270	019c08	000200	00		0	0	8
[9]	RETENTION_ROM0	PROGBITS	07fd0000	018000	000008	00	WA	0	0	4
[10]	RETENTION_RAM0	NOBITS	07fd0008	018008	001f58	00	WA	0	0	4
[11]	RETENTION_RAM1	PROGBITS	00000000	019e08	000000	00	W	0	0	1
[12]	.ARM.attributes	ARM_ATTRIBUTES	00000000	019e08	000028	00		0	0	1
[13]	.comment	PROGBITS	00000000	019e30	000070	01	MS	0	0	1
[14]	.shstrtab	STRTAB	00000000	019ea0	0000a8	00		0	0	1

Nr	Name	Type	Addr	Off	Size	ES	Flag	Link	Info	Align
[15]	symtab	SYMTAB	00000000	01a1f0	003b40	10		16	642	4
[16]	strtab	STRTAB	00000000	01dd30	001e73	00		0	0	1

Key to Flags:

```
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```

Table 43: Example program sections for flash cached operation mode

[Nr]	Name	Type	Addr	Off	Size	ES	Flag	Link	Info	Align
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	08000000	008000	004fb0	00	AX	0	0	16
[2]	ARM.exidx	ARM_EXIDX	08004fb0	00cfb0	000008	00	AL	1	0	4
[3]	.copy.table	PROGBITS	08004fb8	00cfb8	000018	00	WA	0	0	1
[4]	.zero.table	PROGBITS	08004fd0	00cfd0	000018	00	WA	0	0	1
[5]	.data	PROGBITS	07fc8000	010000	000064	00	WA	0	0	4
[6]	.bss	NOBITS	07fc8064	018064	000210	00	WA	0	0	4
[7]	.heap	PROGBITS	07fc8278	010e48	001c00	00		0	0	8
[8]	.stack_dummy	PROGBITS	07fc8278	012a48	000200	00		0	0	8
[9]	RETENTION_ROM0	PROGBITS	07fc0100	010100	000d48	00	WAX	0	0	4
[10]	RETENTION_RAM0	NOBITS	07fc0e48	010e48	001f54	00	WA	0	0	4
[11]	RETENTION_RAM1	PROGBITS	00000000	012c48	000000	00	W	0	0	1
[12]	.ARM.attributes	ARM_ATTRIBUTES	00000000	012c48	000028	00		0	0	1
[13]	.comment	PROGBITS	00000000	012c70	000070	01	MS	0	0	1
[14]	.shstrtab	STRTAB	00000000	012ce0	0000a8	00		0	0	1
[15]	.symtab	SYMTAB	00000000	013030	004130	01		16	717	4
[16]	. strtab	STRTAB	00000000	017160	002187	00		0	0	1

Key to Flags:

```
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```

Table 44: Example program sections for flash cached mode with BLE support

[Nr]	Name	Type	Addr	Off	Size	ES	Flag	Link	Info	Align
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	08000000	008000	00c7f4	00	AX	0	0	16
[2]	.ARM.exidx	ARM_EXIDX	0800c7f4	0147f4	000008	00	AL	1	0	4
[3]	jump_table_mem_ar	PROGBITS	0800c7fc	0147fc	000240	00	A	0	0	4
[4]	.copy.table	PROGBITS	0800ca3c	014a3c	000018	00	WA	0	0	1
[5]	.zero.table	PROGBITS	0800ca54	014a54	000018	00	WA	0	0	1
[6]	.data	PROGBITS	07fc0100	018100	000074	00	WA	0	0	4
[7]	.bss	NOBITS	07fc0174	018174	001e28	00	WA	0	0	4
[8]	.heap	PROGBITS	07fc1fa0	01ffd0	000800	00		0	0	8
[9]	.stack_dummy	PROGBITS	07fc1fa0	0207d0	000800	00		0	0	8
[10]	RETENTION_ROM0	PROGBITS	07fd6000	01e000	001fcc	00	WAX	0	0	4
[11]	RETENTION_RAM0	NOBITS	07fd7fcc	01ffcc	005df8	00	WA	0	0	4
[12]	RETENTION_RAM1	PROGBITS	00000000	020fd0	000000	00	W	0	0	1
[13]	RETENTION_BLE	NOBITS	07fdec00	01ffcc	001400	00	WA	0	0	1
[14]	.ARM.attributes	ARM_ATTRIBUTES	00000000	020fd0	000028	00		0	0	1
[15]	.comment	PROGBITS	00000000	020ff8	000070	01	MS	0	0	1

[Nr]	Name	Type	Addr	Off	Size	ES	Flag	Link	Info	Align
[16]	.shstrtab	STRTAB	00000000	021068	0000ca	00		0	0	1
[17]	.symtab	SYMTAB	00000000	02142c	00c510	10		18	2314	4
[18]	. strtab	STRTAB	00000000	02d93c	0086f5	00		0	0	1

Key to Flags:

```
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```

11.2 Program loading

11.2.1 RAM mode

A program built for RAM operation can be loaded to RAM either directly using the ELF file and J-Link debugger or be first converted to a raw binary and then written to RAM with the CLI programmer tool. RAM operation is used only for debugging purposes as it avoids the step of programming the QSPI flash. After loading the program, `SYS_CTRL_REG` must be configured so that RAM is mapped to address `0x00000000`. After a soft reset is issued, the written program starts execution. RAM operation does not rely on the boot loader.

11.2.2 Flash cached mode

A program built for flash cached mode is written into QSPI flash memory with the CLI programmer tool. After a hard reset, the boot loader detects the valid program in QSPI flash and prepares to run it. When executing from flash, the first `0x100` virtual addresses are mapped to the beginning of RAM, i.e. memory area `0x00000000 – 0x000000ff` is mapped to memory area `0x07fc0000 – 0x07fc00ff` (`SYS_CTRL_REG[REMAP_INTVECT]` must always be set to 1). This ensures that ARM Interrupt Vector Table (IVT) is always in RAM memory for quick access.

The bootloader checks if the flash has a valid program by looking for the presence of a special header which is added to the image before the actual program. This header is added prior to writing the image into flash by the `bin2image` utility. The following tables describe the structure of the flash header.

Table 45: Flash image header for DA14680/1-01

Address (byte)	Value	Description
0:1	‘p’, ‘P’ or ‘q’, ‘Q’	ASCII header to identify the functional mode of the device. “pP”: Mirrored mode (SPI) “qQ”: Cached mode (QSPI)
2:3	0, 0	Reserved
4:7	Any	Image length (big endian)

Simply prepending the header to the binary image would shift the entire image in flash and corrupt the code as all the function addresses would be wrong. Instead, `bin2image` only modifies the first `0x100` bytes.

As shown in Figure 43, the reserved area that follows the IVT is reduced by the size of the flash header (H). This is harmless, since the contents of the reserved area are written after the program starts execution. After the boot loader detects the valid image in flash, it copies memory area `[0x80000000 + H, – 0x800000ff]` to RAM `[0x07fc0000, 0x07fc00ff – H]`, thus skipping the header and placing the `IVT` in the beginning of RAM, restoring the reserved area to its original size.

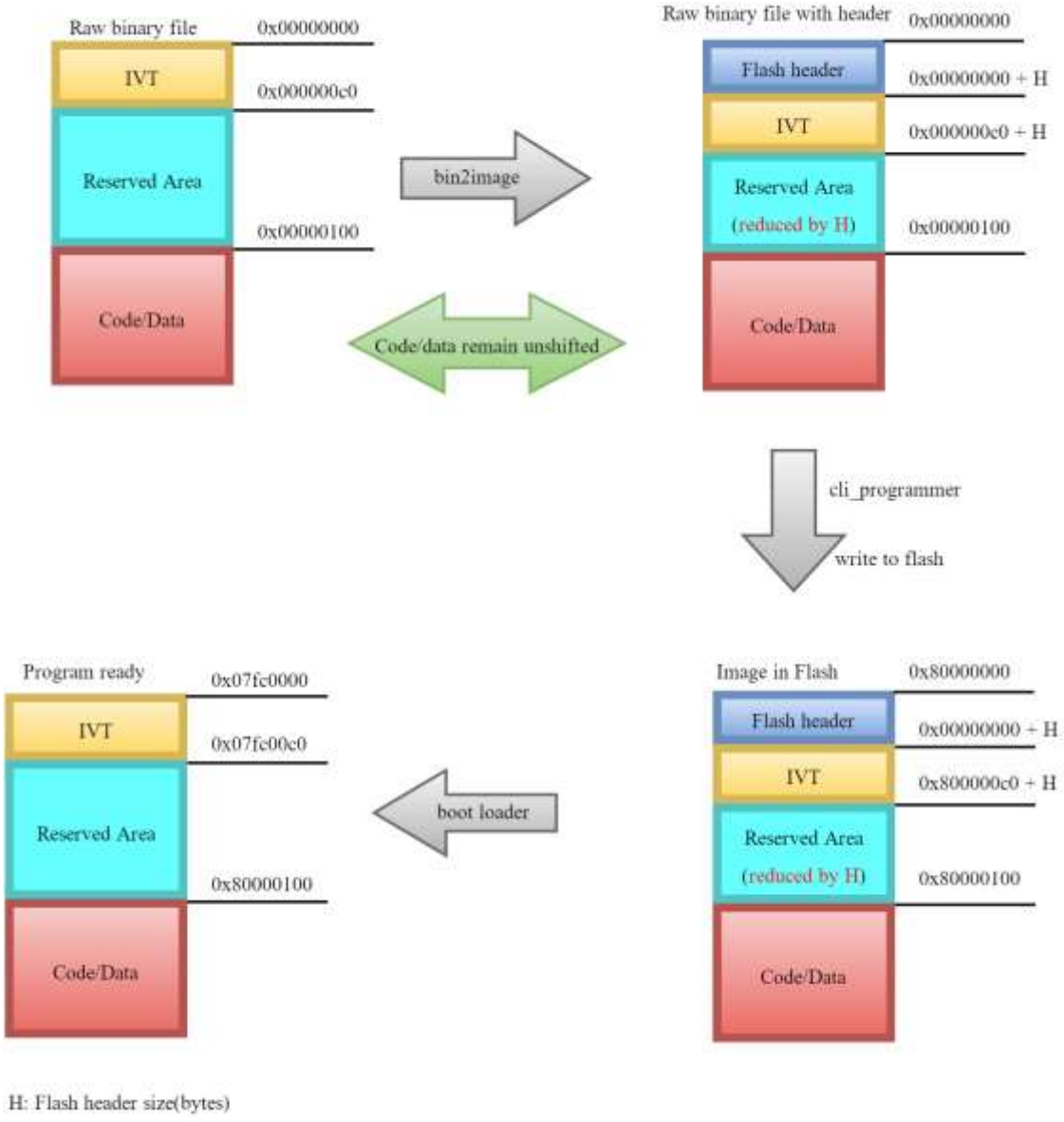

Figure 43: Flash cached pre-execution stages

11.3 BLE ROM patches

DA1468x employs special hardware to support BLE ROM patching. The patch controller is a memory address translator with 28 entries and it has additional registers for validating and invalidating entries. 20 of those entries are for patching BLE ROM functions, and the remaining 8 are for patching data. The patch controller intercepts memory read accesses from the processor. If the memory read address matches any of the addresses of the valid patch controller entries, a patch action is triggered.

For data patches, the patch controller simply redirects the read access to the address written in the entry's data address register.

For function patches, the patch controller redirects execution to the corresponding function address at virtual memory location `0x000000C0 + N`, where `N (0 <= N < 20)` is the index of the entry of the

patch controller that has been triggered. This implies that virtual memory addresses `0x000000C0` to `0x0000010c` must contain patch function addresses (4 byte aligned), if the corresponding entry of the patch controller is marked as valid. This is the reserved memory area described in the previous section after the IVT. The patch memory area includes 8 * 4 additional reserved bytes for a total size of 28 * 4 = 112 bytes. Its first 64 bytes reside in the Reserved Area in RAM as discussed previously in this document, and the remaining bytes are in flash memory (Bluetooth low energy applications do not support RAM operation mode), beginning at address 0x80000100.

11.4 Startup procedure

The startup procedure is the part of the program that runs after reset and before entering `main()`. It consists briefly of the following steps (please consult startup code within the SmartSnippets™ DA1468x SDK for details).

`Reset_Handler` in `sdk\bsp\startup\startup_ARMCM0.S`

- Deactivate cache and include it to available RAM (RAM operation only)
- Copy first 0x100 –H bytes from Flash to RAM (Flash cached operation and flash offset = 0)

`SystemInitPre()` in `sdk\bsp\startup\system_ARMCM0.c`

- Enable debugger (if corresponding option is enabled)
- Enable Fast clocks
- Check alignment of copy and zero tables

`SystemInit()` in `sdk\bsp\startup\system_ARMCM0.c`

- Check IC version compatibility with SW
- Initialize TCS (see Appendix H for details on TCS contents)
- Activate BOD protection
- Configure interrupt priorities
- If executing from RAM ensure PMU is in a known good state
- RC16 Clock setup
- Disable XTAL16M
- Set QSPI to highest speed

`SystemInitPost()` in `sdk\bsp\startup\system_ARMCM0.c`

- Start LDOs
- Set Radio voltage to 1.4V
- Initialize the QSPI flash (flash cached mode only)
- Read Trim values from OTP
- Apply Trim values from OTP
- Enable the QSPI Flash (flash cached mode only)
- Apply the System values from TCS
- Configure cache (flash cached mode only)

`Reset_Handler` in `sdk\bsp\startup\startup_ARMCM0.S`

Note 16 In RAM configuration these steps take place before `SystemInitXXX()` calls

- Copy code and data to RAM according to .copy.table section

- Initialize certain memory areas to zero according to .zero.table section

11.5 Secure Boot

Note 17 XTAL32M support is available only for DA14683 devices – it is not supported for DA14681 devices.

Secure Boot is an alternative bootloader which could be used as a second stage bootloader during Software Update over the Air (SUOTA) procedure which supports:

- FW (firmware) Authentication: **securely** ensures validity and authenticity of entire Application firmware during booting.
- Rollback Prevention: prevents execution of out-of-date vulnerable code.
- Public Keys administration: Root keys being used for Integrity protection can be revoked.

Secure Boot depends on cryptographic Engines Low-level drivers only. Being the only thread running at boot time, it does not require thread-safe APIs from Security framework.

11.5.1 Features

Secure Boot is implemented in *main_secure.c* file located in: `<sdk_root_directory>\sdk\bsp\system\loaders\ble_suota_loader`. Figure 44 below presents Secure Boot's main functionality.

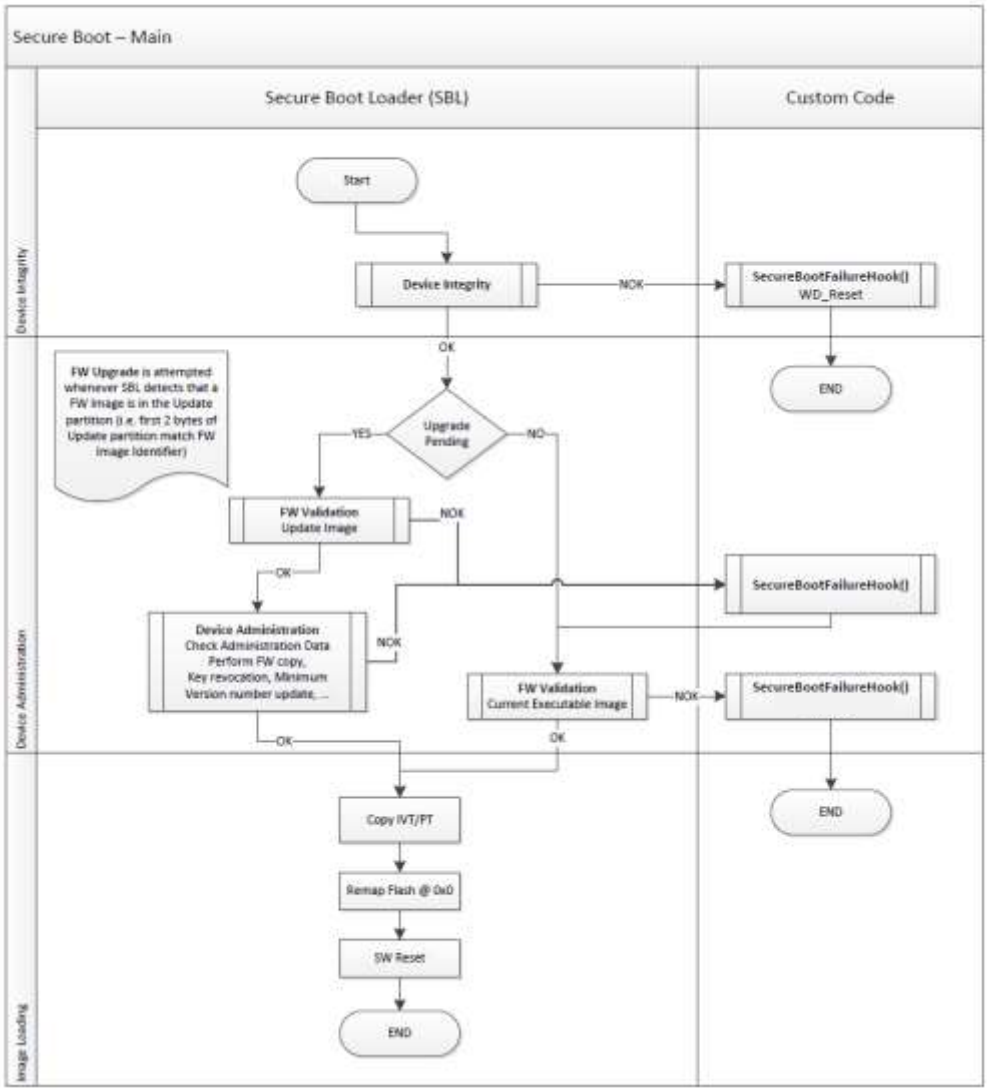

Figure 44: Secure Boot - Main

Next list presents Secure Boot's features:

- "Device Integrity", Figure 45, is a feature of Secure Boot which:

1. Compares bootloader's CRC placed in OTP header with the calculated CRC
2. Checks "Secure Device" field in OTP header (some functionalities of Secure Boot are available only for secured devices)
3. Validates the symmetric keys that stored in OTP and used in encryption/decryption
4. Validates the root/public keys that stored in OTP and used in image signature validation
5. Checks minimum FW version image stored in the OTP ()

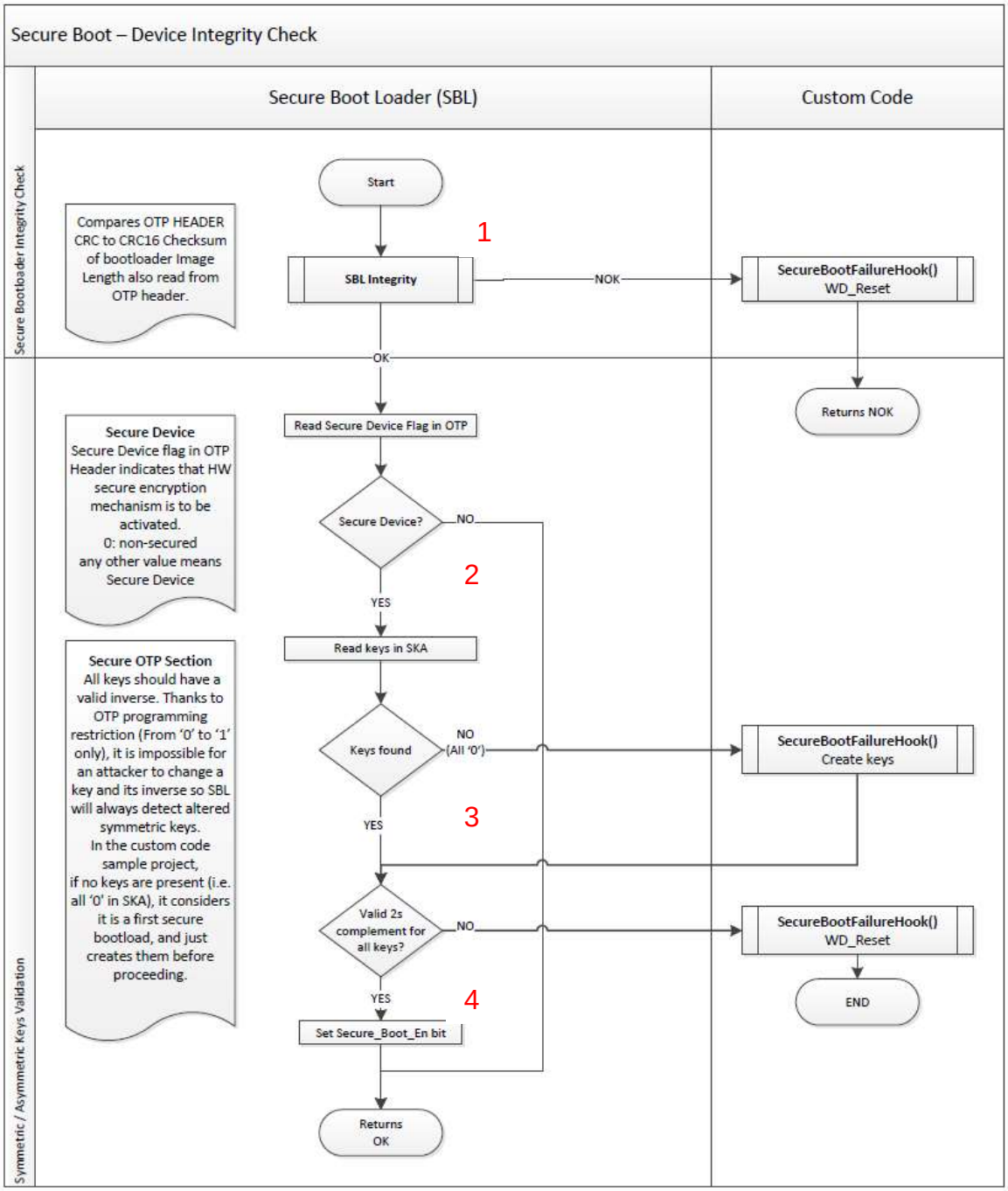

Figure 45: Secure Boot – Device Integrity Check

- "Firmware Validation" is a feature of Secure Boot which:

1. Checks SUOTA 1.1 header
2. Checks image's CRC
3. Validates the header of security extension content
4. Checks FW version number with the current minimum FW version

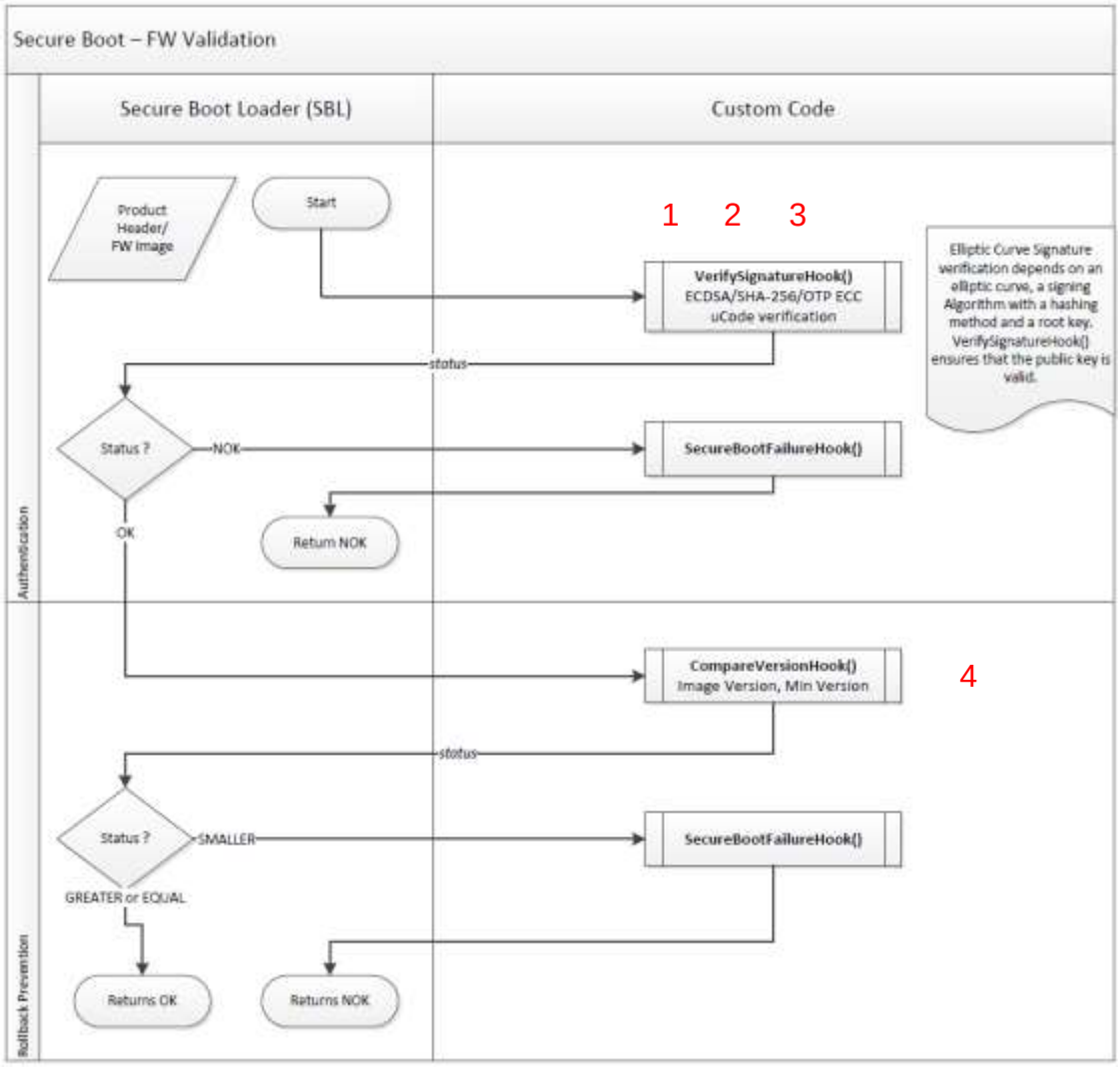

Figure 46: Secure Boot – FW validation

- Copying of the FW stored on the 'update' partition to the 'executable' partition (section 1, Figure 47)
- Upgrade of the minimum FW version array (section 2, Figure 47)
- Customizable code (hooks), (section 3, Figure 47)
- Root/public keys revocation possibility (section 4, Figure 47)

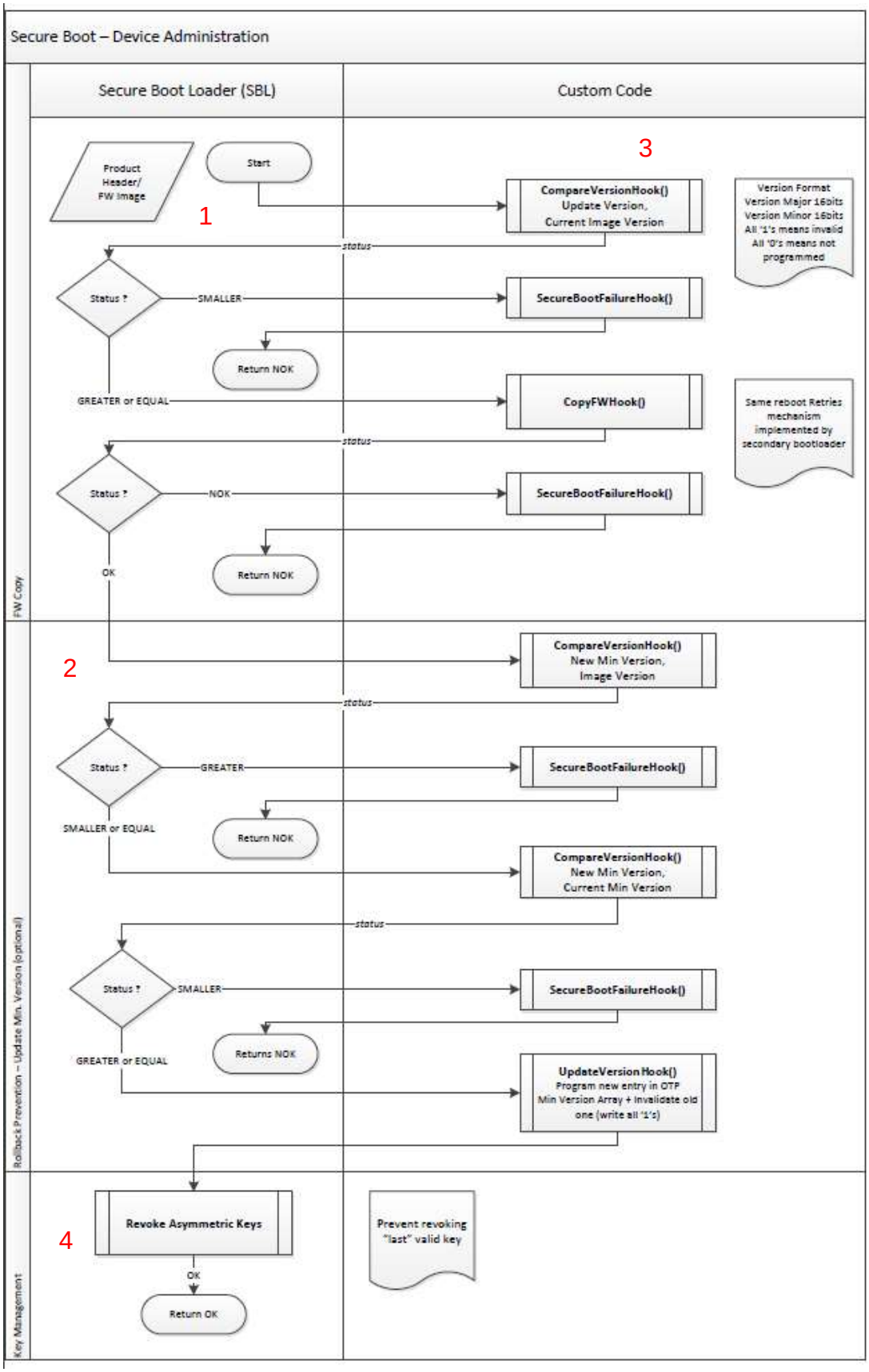

Figure 47: Secure Boot – Device Administration

Note 18 Secure Boot loader is stored in the OTP by default. Proper build configuration of the project must be used in order for the Secure Boot's features to be available. Each configuration with `_Secure` suffix builds Secure Boot Loader as shown in Figure 48.

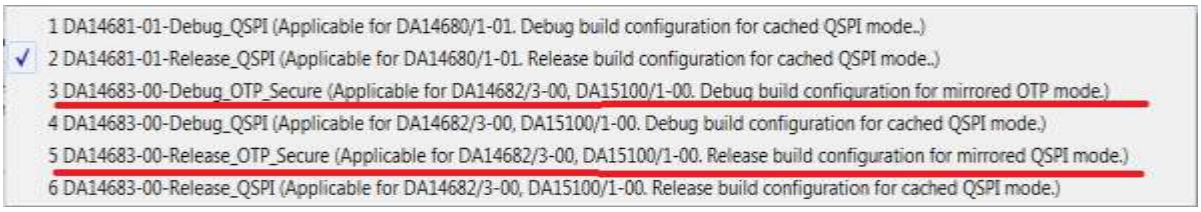

Figure 48: Secure Boot – Build Configurations

Secure Boot doesn't use FreeRTOS and BLE. The SUOTA procedure must be handled by firmware - application image e.g. PXP Reporter. The installation and use of Secure Boot is described in the following section **Error! Reference source not found.**.

11.5.2 Configuration

This section describes the installation and use of Secure Boot. At first user must import in the workspace of SmartSnippets™ Studio the following:

1. ble_suota_loader project. For more information about importing ble_suota_loader project in SmartSnippets™ Studio please refer to [4].
2. Any application which supports SUOTA feature, e.g. pxp_reporter demo project with SUOTA support. For more information about importing and building pxp_reporter demo application please refer to section 5 of [3].
3. Python scripts. To import Python scripts into SmartSnippets™ Studio follow the same procedure as importing any other project in SmartSnippets™ Studio.

The list of aforementioned projects, imported to SmartSnippets™ Studio is shown in Figure 49.

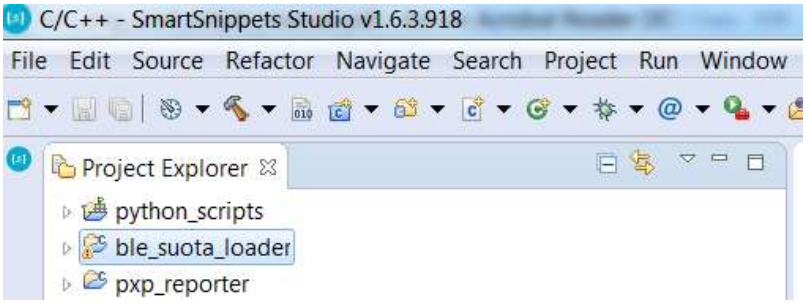

Figure 49: Secure Boot – IDE imported projects

User must follow next steps in order to install Secure Boot:

1. Build firmware image of the imported application (e.g pxp_reporter) using any mode (Release or Debug) with SUOTA support e.g. `DA14683-00-Debug_QSPI_SUOTA`
2. Build ble_suota_loader project using `DA14683-00-Release_OTP_Secure` build configuration (Figure 48)
3. Use `secure_image_config.py` Python script as show in Figure 50. Answering on few questions will be required during script execution as shown in next figures.

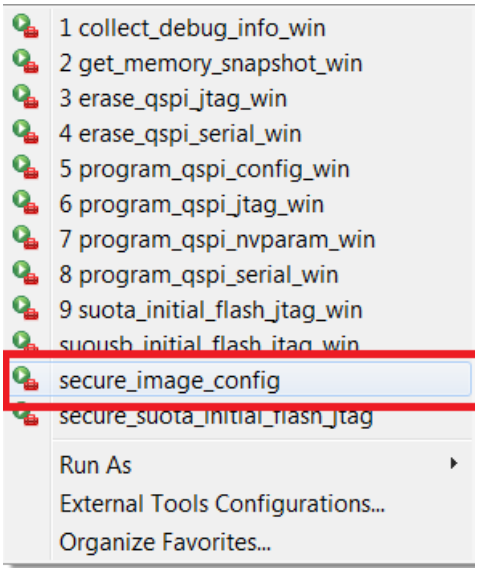

Figure 50: secure_image_config Python script

4. First window after launching `secure_image_config` script is shown in Figure 51. By selecting "Yes" new product keys file is generated automatically.

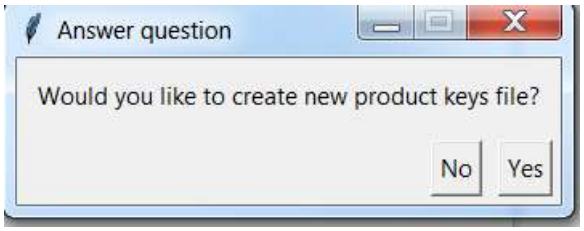

Figure 51: Question window to create new product keys file

5. After selecting "Yes" in the previous dialog next window, Figure 52, pops out. User must decide, which type of elliptic curve would like to use for creating asymmetric keys. The curve will be used during an asymmetric key generation and image signature generation. After this procedure completes, *product_keys.xml* file will be generated in python_scripts project in secure_image subdirectory (Figure 53).

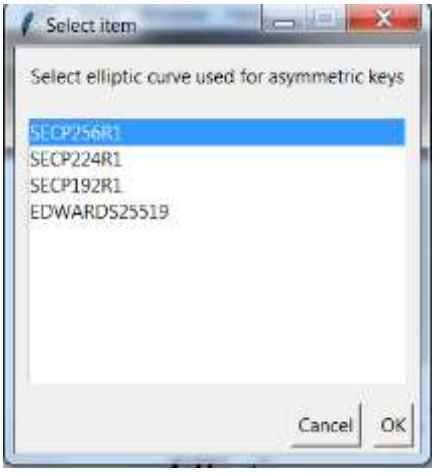

Figure 52: elliptic curves used for creating asymmetric keys

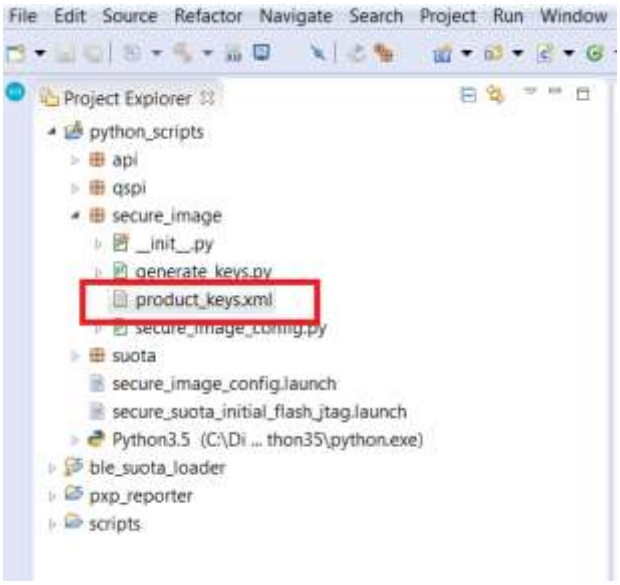

Figure 53: generated *product_keys.xml* file

Note 19 User may need to refresh workspace (or hit F5) in order to make product_keys.xml file visible

Note 20 By selecting “No” in Figure 52 the user must provide the private key manually by inserting a private key index and its value as shown in Figure 54 and Figure 55.

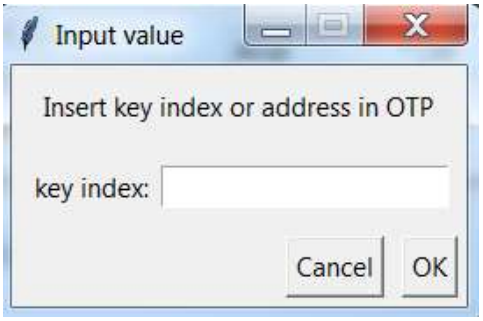

Figure 54: inserting private key index or address

Figure 55: inserting private key value

6. In this step the user decides to create private key manually (by answering “NO”) or automatically (by answering “YES”) by choosing it from existed *product_keys.xml* file (Figure 56).

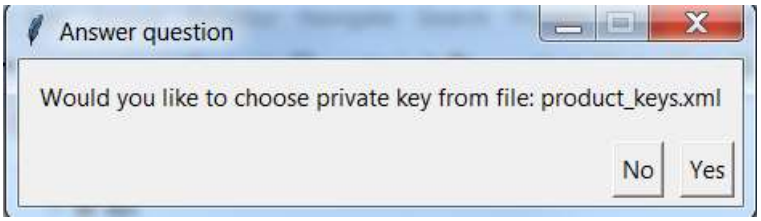

Figure 56: window to select the use of private key

7. User must select one public key index as shown in Fig. 11 which will be used during image signature validation (on the platform). Public keys should be stored in a proper order in the OTP memory.

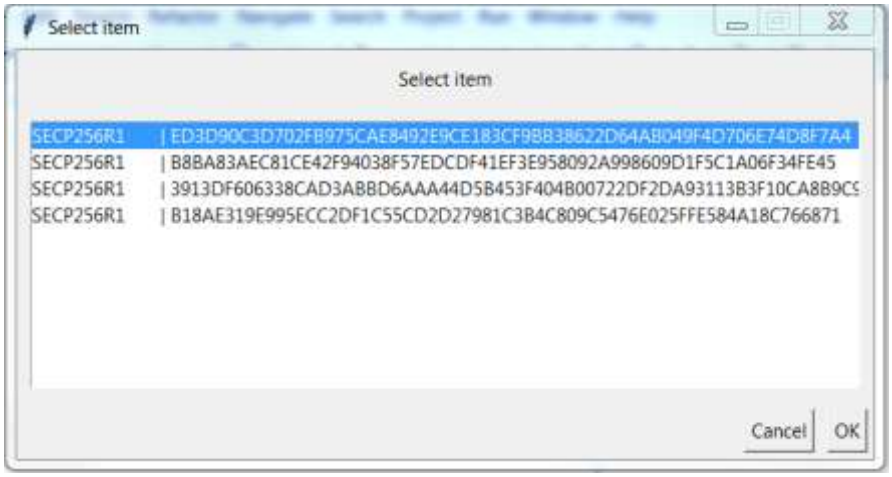

Figure 57: selecting private key from *product_keys.xml* file

8. To each public key index a private key is assigned. This private key is then used during image signature generation. If *product_keys.xml* file already exists then the user is prompted to create a new product_keys.xml file with new keys and save old keys in *product_keys.xml.old* file (Figure 58).

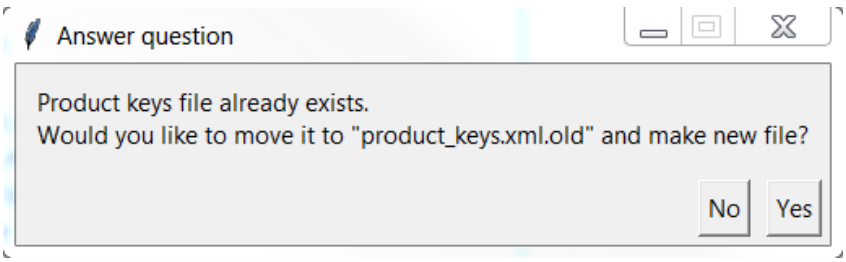

Figure 58: move existing configuration to product_keys.xml.old file

9. For `SECP256R1`, `SECP224R1` or `SECP192R1` types of curves hash method must be selected (Fig. 13). In case of `EDWARDS25519` it is not needed because by default this curve uses SHA-512. It will be used during image signature generation.

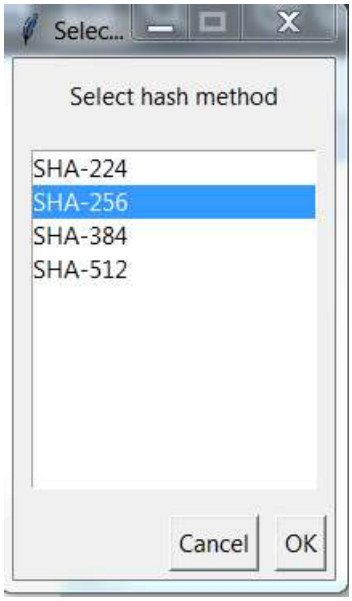

Figure 59: selecting hash method for `SECP256R1`, `SECP224R1` or `SECP192R1`

10. In this optional step, shown in Figure 60, the user can enter public key index/indexes or address/addresses (Figure 61) which will be revoked after image validation on the platform. Allowed values are number of indexes of asymmetric keys (0 – 3 for DA1468x Platform) or addresses of symmetric keys (s0 – s7 for DA1468x Platform).

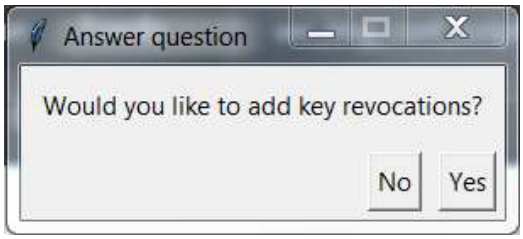

Figure 60: add key revocations selection

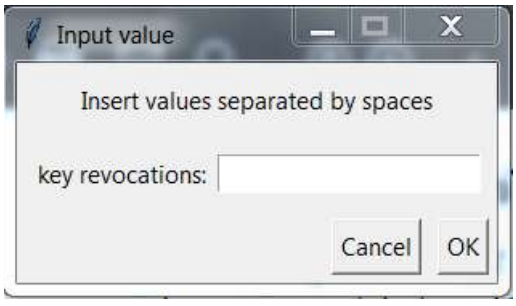

Figure 61: key revocations values window

11. In this step the user can add minimal SW version on the platform (this is optional) as shown in Figure 62. This will be done after image validation on the platform. If the user doesn't write any version in the window show in Figure 63, then SW version of the used image will be used as new minimal SW version.

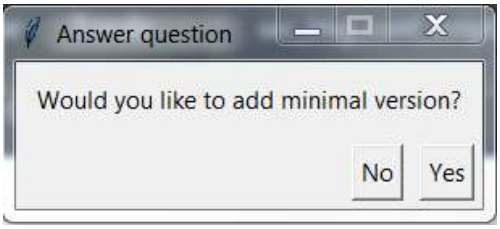

Figure 62: adding minimal version of software version

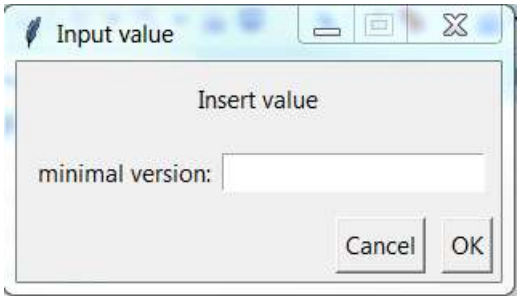

Figure 63: inserting minimal value of software

12. Use `secure_suota_initial_flash_jtag` python script as shown in Figure 64. This script is used to save all generated keys and the secure bootloader in OTP memory and then flash the selected application in secure mode into DA1468x board.

Note 21 Before launching the secure application (e.g. pxp_reporter) and ble_suota_loader must be first built with the following modes:

- ble_suota_loader in Release mode – `DA14683-00-Release_OTP_Secure`
- pxp_reporter in any kind of mode with SUOTA support e.g. – `DA14683-00-Debug_QSPI_SUOTA`.

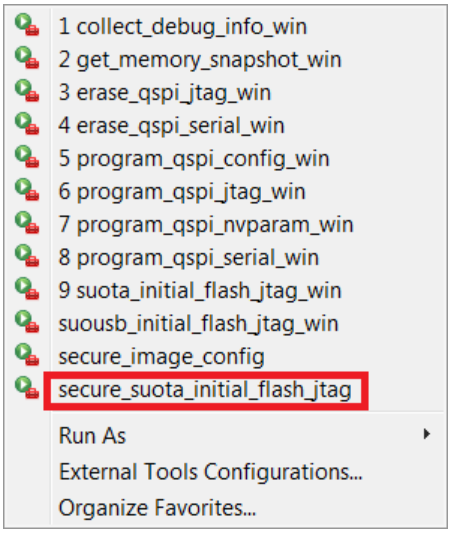

Figure 64: `secure_suota_initial_flash_jtag` script

11.5.3 Files

When the configuration procedure (described in 11.5.2) of Secure Boot ends two xml files are created: *product_keys.xml* (Figure 66) file and *secure_img_cfg.xml* (Figure 67) as shown in Figure 65.

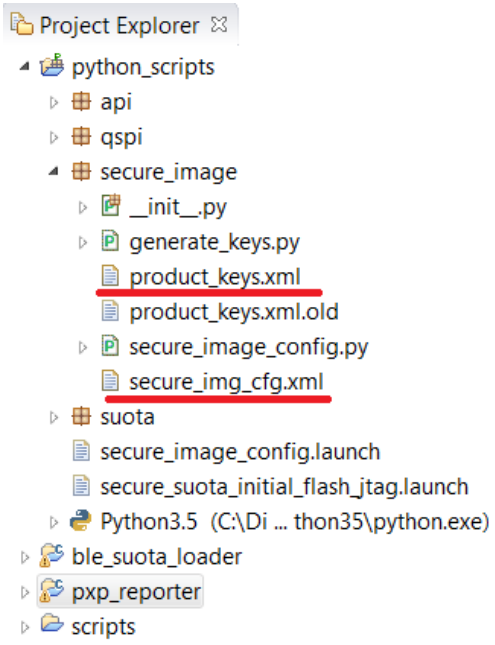

Figure 65: Secure Boot - generated files

product_keys.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<keys>
    <symmetric>
        <symmetric_key>7245299285D26D452ED4B3D187FA3D7E12FF2DF41DD9ED082434CDD7A71EEE11</symmetric_key>
        <symmetric_key>5E0460987BDD874A2FA5C31A6F0D4F66F8716A85910AB59F9065DDC19F137930</symmetric_key>
        <symmetric_key>C0A336FE8FCD31DCC7F0A42F08BFC566D836EB8AE7D4F3169704E20AB8DCCB3B</symmetric_key>
        <symmetric_key>9F09F4ED4A0A34A3FF9D9F360A776625B938F82CA89E6D1841F925D984DFACD9</symmetric_key>
        <symmetric_key>0420E18C32FC5849DE94FC58FB9FF94CA35ED9925BD2EC4B972CEE578185E4B2</symmetric_key>
        <symmetric_key>F6CF4704D10C64736DBE03BC659E47829E90D5E688D73857A08585AB3A363A6F</symmetric_key>
        <symmetric_key>7DFE6C7CAFA022CBB301FC8BCFDC187083B6354DB71519E565ED31FF375B78B8</symmetric_key>
        <symmetric_key>A195991D522258F9B8462EECC2C133BDE9B942F171F4579DEC4A3B78015A6434</symmetric_key>
    </symmetric>
    <asymmetric>
        <asymmetric_keys>
            <private>819966F5D53DE0940D9804C5E700249CF74E02E8A5D8F1C0759F365E8BC9B431</private>
            <public>7A91E1A232343B61F4162E4645C979EEBDF39CF2E766FC85003A850C228ACDD075AEA655186F0559D3D759617AE
            <elliptic_curve>SECP256R1</elliptic_curve>
        </asymmetric_keys>
        <asymmetric_keys>
            <private>E2B076C46499B1747A29C61EF779BF8DC2665041D78AFF268E68874820F17F02</private>
            <public>3E6107195468888C1572EEAE9414FD540DC780A6707964214728370775F170B43FC736C2020634EABCA93545A8
            <elliptic_curve>SECP256R1</elliptic_curve>
        </asymmetric_keys>
        <asymmetric_keys>
            <private>8403D2B2980550E631DE1CBE356AF118DE8A7808BE1C6C6F012BFCCFF362715D</private>
            <public>97A4B2A7146D78D3198286948A4393E55C642D8DB300792E88D58BE6FFC8711CA1E6B136756CFD160595213770
            <elliptic_curve>SECP256R1</elliptic_curve>
        </asymmetric_keys>
        <asymmetric_keys>
            <private>A7D2B9FEB1C2FF2A70F845E4DF13FAED8CFBBC7D98CF79DA0D1FD428435CCA83</private>
            <public>6BCBED29E6240962EACB978751484A84E13D99E1661BE38E0B2A2CE74C165992DE52E157D519B414FA70DA3AF7
            <elliptic_curve>SECP256R1</elliptic_curve>
        </asymmetric_keys>
    </asymmetric>
</keys>
```

Figure 66: *product_keys.xml* file

secure_img_cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<secure_img_cfg>
    <security>
        <key_idx>0</key_idx>
        <private_key>819966F5D53DE0940D9804C5E700249CF74E02E8A5D8F1C0759F365E8BC9B431</private_key>
        <hash_method>SHA-256</hash_method>
        <elliptic_curve>SECP256R1</elliptic_curve>
    </security>
    <device_administration>
        <key_revocation>0</key_revocation>
        <key_revocation>2</key_revocation>
        <key_revocation>s0</key_revocation>
        <key_revocation>s6</key_revocation>
        <key_revocation>s4</key_revocation>
    </device_administration>
</secure_img_cfg>
```

Figure 67: *secure_img_cfg.xml* file

These files are used by other scripts (e.g. `secure_suota_initial_flash_jtag` and `mkimage`) as input files.

Note 22 For more info about SUOTA please refer to section 9 of [3].

Note 23 `initial_flash.py` script performs writing to the One Time Programmable (**OTP**) memory. When this procedure is called with invalid configuration/firmware/bootloader files then the device may become unusable!

Note 24 Scripts are using Python 3

12 Drivers and Adapters

12.1 Introduction

The DA1468x family of devices supports several peripherals on different interfaces. To support them the SmartSnippets™ DA1468x SDK provides Low Level Drivers (LLD) and/or Adapters for each of the available hardware peripherals.

A LLD provides a simple API to use the peripheral and abstract the complexities of using the peripheral registers directly.

An Adapter provides a higher level service allowing different tasks to safely share the peripheral and also integrate with the CPM to manage power-down modes

The rest of this Section will provide an overview of the available Drivers and Adapters.

Note 25 All drivers and adapters in the SmartSnippets™ DA1468x SDK are supplied in full source to aid debugging. However, modifying the drivers is not advised.

12.2 Drivers

This section will cover the LLDs for the peripherals of the DA1468x. The LLDs allow application software code to access and use the device peripherals without detailed knowledge of the hardware implementation, such as bits and their position within hardware registers.

It is recommended to only use the LLD drivers to access the device peripherals as the LLDs are tested and verified. Direct access to hardware resources (e.g. registers or peripheral interfaces) might lead to conflicts with lower level FW functions accessing the same resources through LLDs

and lead to system instabilities. For each hardware peripheral a dedicated header file describes the API functions of the peripheral, lists capabilities and defines control structures which are needed to interact with its particular LLD. The header files can be found under `<sdk_root_directory>/sdk/bsp/peripherals/include`. Table 46 lists all available LLDs.

Table 46: LLD overview

Filename	Description
hw_aes_hash.h	Definition of API for the AES/HASH Engine Low Level Driver.
hw_breath.h	Definition of API for the Breath timer Low Level Driver.
hw_cpm.h	Clock and Power Manager header file.
hw_crypto.h	Interrupt handling for the crypto engines (AES/HASH, ECC)
hw_dma.h	Definition of API for the DMA Low Level Driver.
hw_ecc.h	Definition of API for the ECC Engine Low Level Driver.
hw_ecc_curves.h	ECC Engine curves parameters.
hw_ecc_ucode.h	ECC Engine microcode.
hw_fem_sky66112-11.h	FEM Driver for SKYWORKS SKY66112-11 Low Level Driver API.
hw_gpadc.h	Definition of API for the GPADC Low Level Driver.
hw_gpio.h	Definition of API for the GPIO Low Level Driver.
hw_hard_fault.h	Hard-Fault Handler.
hw_i2c.h	Definition of API for the I2C Low Level Driver.
hw_irgen.h	Definition of API for the IR generator Low Level Driver.
hw_keyboard_scanner.h	Definition of API for the Keyboard scanner Low Level Driver.
hw_led.h	Definition of API for the LED Low Level Driver.
hw_otpc.h	Definition of API for the OTP Controller driver.
hw_qspi.h	Definition of API for the QSPI Low Level Driver.
hw_quad.h	Definition of API for the QUAD Decoder Low Level Driver.
hw_rf.h	Radio module (RF) Low Level Driver API.
hw_soc.h	Definition of API for the SOC Low Level Driver.
hw_spi.h	Definition of API for the SPI Low Level Driver.
hw_tempsens.h	Implementation of the Hardware Temperature Sensor interface abstraction layer.
hw_timer0.h	Definition of API for the Timer0 Low Level Driver.
hw_timer1.h	Definition of API for the Timer1 Low Level Driver.
hw_timer2.h	Definition of API for the Timer2 Low Level Driver.
hw_trng.h	Definition of API for the True Random Number Generator Low Level Driver.
hw_uart.h	Definition of API for the UART Low Level Driver.
hw_usb.h	Header for low level DA1680 USB drive
hw_usb_ch9.h	Header file with USB configuration info for DA1680 USB driver.

Filename	Description
hw_usb_charger.h	Definition of API for the USB Charger.
hw_watchdog.h	Definition of API for the Watchdog timer Low Level Driver.
hw_wkup.h	Definition of API for the Wakeup timer Low Level Driver.
sys_tcs.h	TCS Handler header file.

In addition to the LLDs listed above, there is also a Low Level Pulse Density Modulation (PDM) Audio Interface driver. The PDM audio interface driver uses the Audio Processing Unit (APU) and the Sample Rate Converter (SRC) devices to implement a PDM interface with input and output support. The driver supports input and output directly from/to an application using the SRC I/O registers at various sample rates. It also supports both master and slave PDM mode. The Low Level Interface driver file can be found under `<sdk_root_directory>/sdk/interfaces/audio/include/if_pdm.h`

12.2.1 LLD header Example

The table included below shows the typedefs, the enumerations and the functions for the quadrature decoder hardware as an example.

Note 26 All API calls starting with `hw_xx` indicate an LLD function.

Table 47: LLD header file

Typedefs	
`typedef void(*`	`hw_quad_handler_cb) (void)`
	`QUAD interrupt callback.`
Enumerations	

`typedef enum {`	`HW_QUAD_CHANNEL_NONE = 0,`
	`HW_QUAD_CHANNEL_X = (1 << 0),`
	`HW_QUAD_CHANNEL_Y = (1 << 1),`
	`HW_QUAD_CHANNEL_Z = (1 << 2),`
	`HW_QUAD_CHANNEL_XY = HW_QUAD_CHANNEL_X | HW_QUAD_CHANNEL_Y,`
	`HW_QUAD_CHANNEL_XZ = HW_QUAD_CHANNEL_X | HW_QUAD_CHANNEL_Z,`
	`HW_QUAD_CHANNEL_YZ = HW_QUAD_CHANNEL_Y | HW_QUAD_CHANNEL_Z,`
	`HW_QUAD_CHANNEL_XYZ = HW_QUAD_CHANNEL_X | HW_QUAD_CHANNEL_Y |`
	`HW_QUAD_CHANNEL_Z,`
	`HW_QUAD_CHANNEL_ALL = HW_QUAD_CHANNEL_XYZ`
	`} HW_QUAD_CHANNEL;`

	`Channels definitions.`
Functions	
`static inline void`	`hw_quad_init(uint16_t clk_div)`
	`Initialization of QUAD driver`
`static inline void`	`hw_quad_enable(void)`
	`Enable QUAD driver`
`static inline void`	`hw_quad_disable(void)`
	`Disable QUAD driver.`
`static inline void`	`hw_quad_set_channels(HW_QUAD_CHANNEL ch_mask)`
	`Set channels state`
`static inline void`	`hw_quad_enable_channels(uint8_t ch_mask)`
	`Enable channels`
`static inline void`	`hw_quad_disable_channels(uint8_t ch_mask)`
	`Disable channels`
`static inline HW_QUAD_CHANNEL`	`hw_quad_get_channels(void)`
	`Get channels state`

Typedefs	
Void	hw_quad_register_interrupt(hw_quad_handler_cb handler, uint16_t tnum);
	Turn on QUAD interrupt
static inline bool	hw_quad_is_irq_gen(void)
	Check if interrupt has occurred
Void	hw_quad_unregister_interrupt (void)
	Turn off QUAD interrupt
static inline int16_t	hw_quad_get_x(void)
	Get the number of steps from X channel
static inline int16_t	hw_quad_get_y(void)
	Get the number of steps from Y channel
static inline int16_t	hw_quad_get_z(void)
	Get the number of steps from Z channel

12.2.2 Documentation

All LLD header files contain the description of the individual API methods, any types they define and their input and output parameter as well as their output types. The LLD header files were written to support the documentation generation tool Doxygen (http://www.doxygen.org). This approach allows the generation of an HTML-like description of each of the individual LLD header files including all possible typedefs and defines as well as an accurate description of all API calls and their parameters. Additionally, it gives a short summary of what each one of the API calls does.

Please refer to [2] to find more details on how to generate an HTML description of a particular LLD. Figure 68 presents the main Doxygen page found at

```
<sdk_root_directory>/doc/html/index.html
```

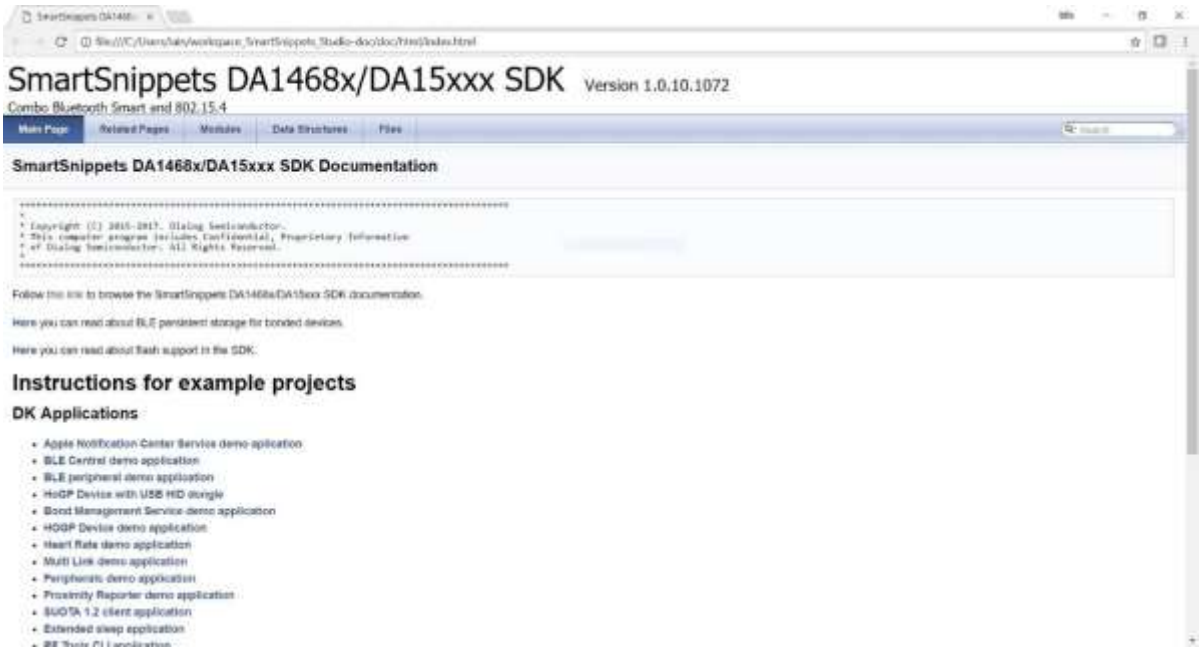

Figure 68: Html file generated by Doxygen

12.3 Adapters

Drivers may also contain a higher layer –the adapter- which allows more than one application thread to request access and get serviced by the driver. Please note that the same “adapter” scheme is

used for data buffers, utilities and other system resources that could be utilized by an application as shown in Figure 69.

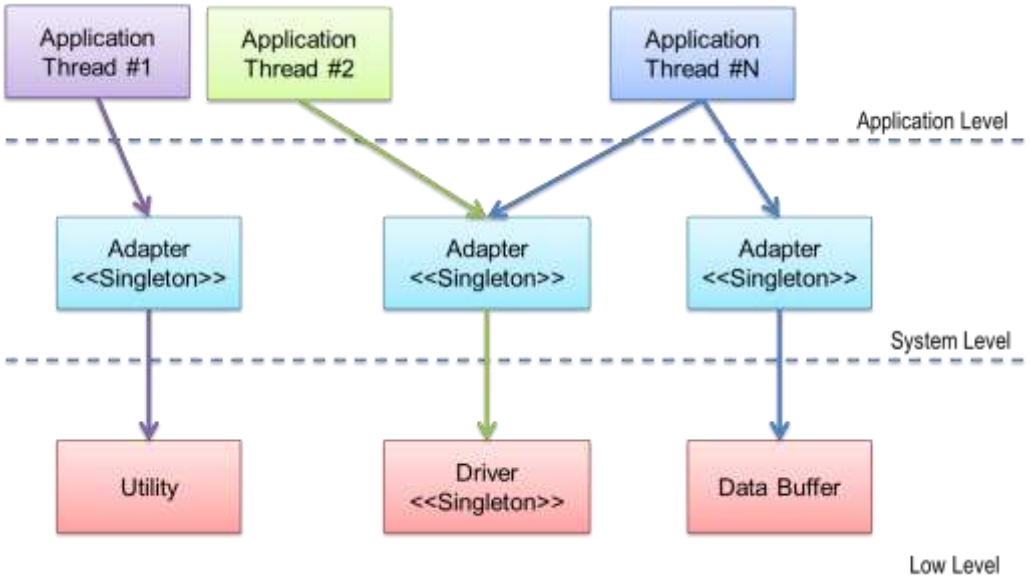

Figure 69: Adapter overview

The adapters as provided by the SmartSnippets™ SDK enables requests for a specific driver or resource from different tasks to be managed to handle resource availability.

The adapters use OS features such as semaphores or events and the resource management API in the `osal` layer to manage multiple simultaneous resource acquisition/release requests. The adapters not only provide access to the peripheral, but also make sure that other tasks which are currently accessing it, suspend their operation until the peripheral is once again released. They also interact with the CPM module so that device will only sleep if all peripherals are inactive.

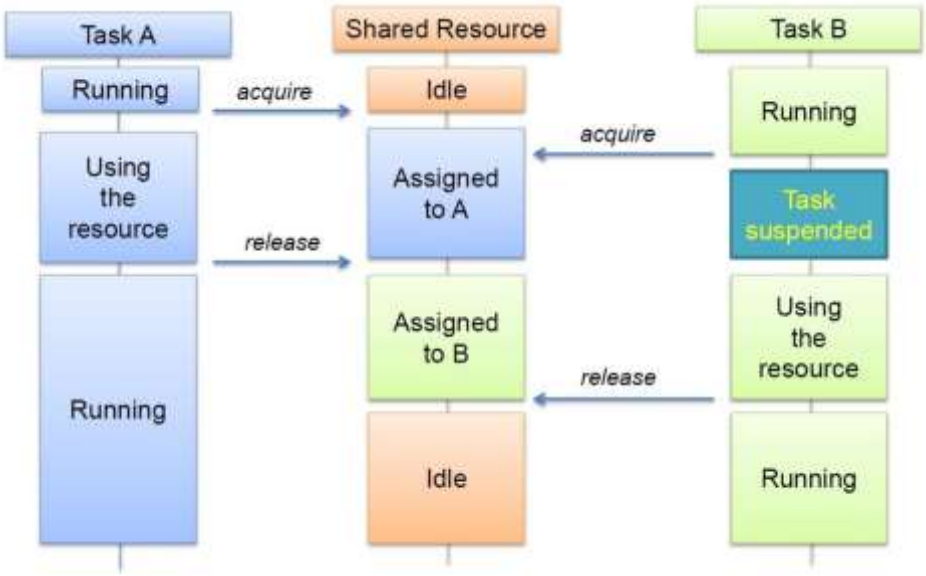

Figure 70: Adapter communication

Note 27 Adapters are not implemented as separate tasks and should be considered as an additional layer between the application and the LLD. The Adapter executes in the context of the calling task.

Note 28 **The recommendation is to use the adapters to access the hardware peripherals where possible**.

The adapter header files can be found under `<sdk_root_directory>/sdk/bsp/adapters/include`. Table 48 lists all available adapters.

Table 48: Adapter overview

Filename	Description
ad_battery.h	Battery adapter API.
ad_crypto.h	ECC and AES/HASH device access API
ad_defs.h	Common definitions for adapters.
ad_flash.h	Flash adapter.
ad_gpadc.h	GPADC adapter API.
ad_i2c.h	I2C device access API.
ad_keyboard_scanner.h	Keyboard Scanner Adapter API
ad_spi.h	SPI adapter API.
ad_uart.h	UART adapter API.
ad_nvms.h	Nonvolatile memory storage API.
ad_nvparam.h	NV Parameters adapter interface.
ad_nvparam_defs.h	Define NV Parameters adapter interface.
ad_nvms_direct.h	NVMS direct access driver.
ad_nvms_ves.h	NVMS VES driver.
ad_rf.h	Radio module access functions.
ad_temp_sens.h	Temperature Sensor adapter API.
flash_partitions.h	Default partition table.
partition_def	Partition table entry definition
partition_table.h	Partition table.
platform_devices.h	Configuration of devices connected to board.
platform_nvparam.h	Configuration of non-volatile parameters on platform.
platform_nvparam_values.h	Parameter value.

The SDK includes a "Peripherals demo application" which provides an excellent example of how many of the adapters are used. Please refer to the related Doxygen documentation and source code.

12.3.1 The UART adapter example

The UART adapter is an intermediate layer between the UART LLD and a user application. It allows the user to utilize the UART interface in a simpler way than using the pure LLD functions.

Features:

- Synchronous writing/reading operations block the calling task while the operation is performed using sempahores rather than relying on a polling loop approach. This means that while the hardware is busy transferring data, the Operating System (OS) scheduler may select another task for execution, thus utilizing processor time more efficiently. After the transfer finishes the calling task is released and resumes execution.
- DMA channel resource management for shared usage among various peripherals (e.g. I2C, UART). Interconnected peripherals may use the same DMA channel if necessary. The adapter takes care of DMA channel resource management.
- Ensuring that only one device can use the UART after acquiring it.
- Putting code between ad_uart_bus_acquire(dev) and ad_uart_bus_release(dev) ensures that only this task can use the UART to communicate with device dev which was previously opened with ad_uart_open. During this period no other device or task can use the UART until the ad_uart_bus_release function is called by the owning task.
- Unlike other serial adapters (I2C and SPI), the UART adapter additionally allows direction-specific resource management. This allows two tasks to access the “read” and the “write” resource simultaneously. Use ad_uart_bus_acquire_ex and ad_uart_bus_release_ex to acquire and release a direction specific resource respectively.

Using the UART Adapter

. dg_configUART_ADAPTER and dg_configUSE_HW_UART definition

To enable the UART adapter, both dg_configUART_ADAPTER and dg_configUSE_HW_UART macros must be defined and set to 1 in the project's config/custom_config_qspi.h header file:

```
#define dg_configUART_ADAPTER                (1)
#define dg_configUSE_HW_UART                 (1)
```

Code 21: Enabling UART Adapter

From this point on, the overall adapter implementation with all its integrated functions becomes available.

2. The platform_devices.h header and UART_BUS macro(s)

Before utilizing the UART adapter the necessary UART_BUS macro(s) must be created using the following definition pattern:

```
UART_BUS(bus_id,                        // valid values: UART1, UART2
         name,                          // name of UART, e.g. COM1
         baud_rate,                     // UART baud rate from enum
                                           HW_UART_BAUDRATE
         data_bits,                     // value from enum HW_UART_DATABITS
         parity,                        // value from enum HW_UART_PARITY
         stop,                          // value from enum HW_UART_STOPBITS
         auto_flow_control,             // 1 if hardware flow control (CTS/RTS) is used,
                                           0 otherwise
         dma_channel)                   // DMA number for Rx channel, Tx will
                                           have next number,
                                     // pass -1 for no
```

Code 22: Parameters of UART bus arguments

Macro(s) should be placed in a platform_devices.h header file, which can be found and copied from <sdk_root_directory>/sdk/bsp/adapters/include to the user's project /config directory. If a new platform_devices.h file is not included there, the application will inherit the default macro(s) definitions from the original platform_devices.h header file, located in the <sdk_root_directory>/sdk/bsp/adapters/include.

These macro(s) describe the parameters of each UART bus and devices connected to it, as shown in Code 23:

```
UART_BUS(UART1, SERIAL1, HW_UART_BAUDRATE_115200, HW_UART_DATABITS_8, HW_UART_PARITY_NONE,
HW_UART_STOPBITS_1, 0, 0, HW_DMA_CHANNEL_1, HW_DMA_CHANNEL_0, 0, 0)
```

Code 23: Parameters of the UART bus

User code

1. **Open the UART device**

The first step is to open the UART device, which was defined by the UART_BUS macro. Calling the function shown in Code 24 opens the device and returns a handle to the main flow for using it in other adapter functions as well.

```
uart_device ad_uart_open(const uart_device_id dev_id);
```

Code 24: Open UART

The initial function call configures the UART block. Subsquent calls from other tasks simply return the already existing handle to the initialized UART, together with the parameters related for each device ID. The dev_id parameter is a second parameter of UART_BUS, for instance, SERIAL1. The returned uart_device handler will then be used in all other adapter functions from now on such as ad_uart_bus_acquire (handler_device) or ad_uart_write (handler_device).

2. **Acquire access to the UART bus**

Before using the UART the application task must request access to it so that no other tasks can use it and potentially corrupt the data transmitted or received. The access is acquired by using the function presented in Code 25:

```
void ad_uart_bus_acquire(uart_device dev);
```

Code 25: Acquire access to UART

This function waits for the UART bus to become available and when it is available locks it down and so reserves it for the current task. The function can be called several times. However, it is essential that the number of ad_uart_bus_release() function calls used for releasing the UART bus matches the number of ad_uart_bus_acquire() calls. When using the ad_uart_bus_acquire() function, only one task has access to the bus and another task is able to use it only after the ad_uart_bus_release() function has been successfully called.

3. **Write/Read to/from the UART device**

Write and read functions can be divided into two methods:

- **Synchronous**

```
void ad_uart_write(uart_device dev, const char *wbuf, size_t wlen);
```

Code 26: Write function

The function shown in Code 26 is used for writing to the UART device in a synchronous manner.

```
int ad_uart_read(uart_device dev, char *rbuf, size_t rlen, OS_TICK_TIME timeout);
```

Code 27: Read function

Similarly, the function presented in Code 27, is used for reading rlen bytes from the UART device, in a synchronous manner as well.

These two functions block the UART bus, however they do not block the operating system. FreeRTOS initially waits for bus access, and then blocks the calling task until a transaction is completed. Once a Write/Read process is finished, the UART bus is free to make another Read/Write operation for the same device.

In synchronous mode the calling task is blocked for the duration of the read or write access but other tasks are not.

- **Asynchronous**

```
void ad_uart_write_async(uart_device dev, const char *wbuf, size_t wlen,
ad_uart_user_cb cb, void *user_data);
```

Code 28: Write function

The function presented in Code 28, works in an asynchronous manner for writing `wlen` bytes to the UART device. Once the data is written to the UART the callback function `cb` is called. After that the `ad_uart_bus_release()` function is called. It is essential that until the callback is received, the caller does not release the `wbuf` memory buffer.

```
void ad_uart_read_async(uart_device dev, char *rbuf, size_t rlen, ad_uart_user_cb
cb, void *user_data);
```

Code 29: Read function

The function presented in Code 29, is used for reading `rlen` bytes from the UART device. The function does not necessary wait for the read operation to finish and starts from gaining access to the UART bus by calling the `ad_uart_bus_acquire()` function. Once the read operation begins, the user must not release the `rbuf` memory buffer. After all data is received the `ad_uart_bus_release()` function is called just before the callback function `cb` is called. To abort an already initiated read operation it is necessary to call the `ad_uart_abort_read_async()` function.

In the asynchronous case the calling task is not blocked by the read or write operation. It can continue with other operations while waiting for callback function to signal the completion of the read or write. Only at this point can `rbuf` be read or `wbuf` be refilled.

4. **Release the UART bus.**

The function presented in Code 30 should be used for releasing the UART bus:

```
void ad_uart_bus_release(uart_device dev);
```

Code 30: Release UART

The function decrements an already acquired counter for a specific dev device and as soon as an internal countdown reaches zero, the UART bus is released and can be used by other tasks.

5. **Closing the UART device.**

After all user operations are done and the device is not needed anymore for additional tasks, it should be closed by using the function presented in Code 31.

```
void ad_uart_close(uart_device device);
```

Code 31: Close UART device

Example of Synchronous access:


```
uart_device dev;
static char wbuf[5] = „Test”;
char rbuf[5];


dev = ad_uart_open(SERIAL1);                /* Open selected device */
ad_uart_bus_acquire(dev);                          /* Acquire access to bus */
ad_uart_write(dev, wbuf, sizeof(wbuf));       /* Write synchronously some data          to
                                            UART device */
ad_uart_read(dev, rbuf, sizeof(rbuf), 100);    /* Read synchronously the data
                                            from UART device */
ad_uart_bus_release(dev);                          /*Release the UART
ad_uart_close(dev);                                /* Close selected device */
```

Code 32: Example of UART adapter usage

12.4 The NVMS Adapter

12.4.1 Overview

The SmartSnippets™ DA1468x SDK includes a Non-Volatile Memory Storage (NVMS) Adapter providing non-volatile memory storage access capabilities to the application (including cached mode). The Adapter provides two main functions:

- Non-Volatile Memory Storage to external Flash devices over standard SPI and Quad SPI performing Write / Read / Erase operations
- Virtual EEPROM (VES) emulation with the following functionalities
  - Wear-levelling
  - Garbage Collection
  - Power failure protection

The VES partition should be used for data that is frequently written to flash. The wear levelling and garbage collection allow the driver to maximize the number of write cycles from software given the limitations of the number of erase & write cycles of the actual flash device.

An overview of NVMS Adapter is shown in Figure 71.

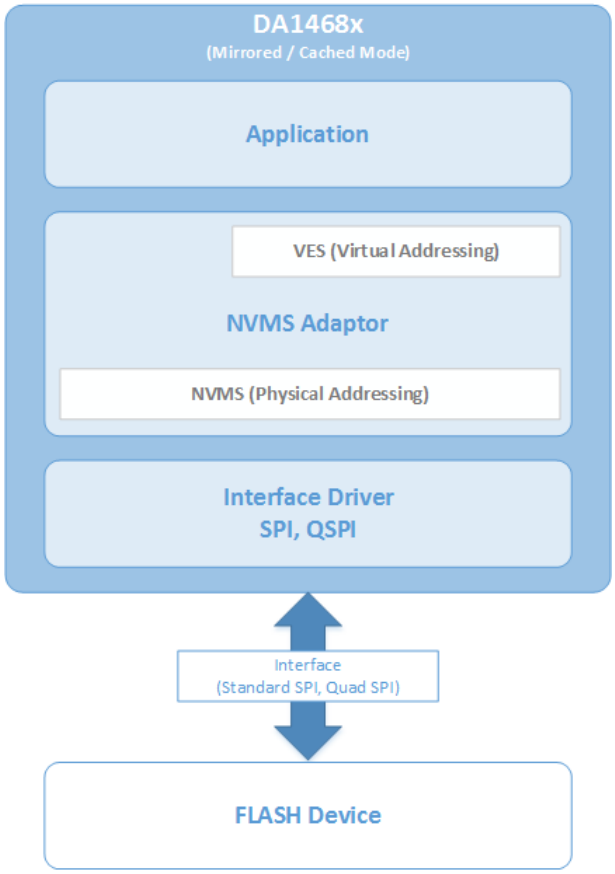

Figure 71: NVMS Overview

In theory NVMS is interface agnostic, as it is independent from the interface type being used (Standard SPI, I2C or QuadSPI) for data storage. However, the QSPI interface is a special interface from the NVMS perspective. As well as performing eXecution in Place (XiP), the QSPI Controller hardware block is also handling specific Flash-aware actions when Flash devices Read/Write/Erase operations are executed. As shown in Figure 73 when DA1468x is in cached mode, special mechanisms from the QSPI block are invoked.

The main consequence of this approach is that all Flash memory models used when utilizing NVMS over QSPI in cached mode must support Erase suspend/resume.

The rest of this section gives an overview on:

- Common NVMS interface for all usage scenarios
- Mechanisms implemented specifically for NVMS over QSPI in cached mode usage scenario
- Virtual EEPROM (VES) emulation
- Flash Memory Map in various usage scenarios.

12.4.2 Interface

NVMS Adapter exposes functions: `ad_nvms_init()`, `ad_nvms_open()`, `ad_nvms_write()` and `ad_nvms_read()`, `ad_nvms_erase()`. Function `ad_nvms_init()` must be called once at platform start to perform all necessary initialization routines, including discovering underlying storage partitions. The Application must open one of the partitions before any read or write activity can be performed. If several partitions are stored on one physical device (i.e. SPI Flash), opening one partition will limit read and writes to this partition only, making all addressing relative to beginning of that partition and not the whole flash. After opening, each partition is accessed in same way, but the exact way that

data is stored depends on partition type. Only in the Virtual EEPROM (VES) partition do all reads and writes use virtual addresses that are independent of actual flash location.

```
nvms_t ad_nvms_open(nvms_partition_id_t id);

…

for (;;) {
        /* addr is any address in partition address space
        * buf can be any address in memory including QSPI mapped flash */
        ad_nvms_read(part, addr, buf, sizeof(buf));


        ad_nvms_write(part, addr, buf, sizeof(buf));
```

Code 33: Usage of NVMS

Function `ad_nvms_open()` can be called many times since it does not allocate any resources.

Function `ad_nvms_read()` can be called with any address within the partition. If address is outside the partition boundaries `ad_nvms_read()` will return 0. If the address is inside the partition but the size would exceed the partition boundary only the data from within the partition will be accessed.

Function `ad_nvms_write()` can be called with any address inside partition address space.

How read or write actually work depends on the accessing method. In the current version of the platform, two accessing methods are supported: **(a) Direct Access** and **(b) VES**.

a. **Direct Access**

Direct access driver uses the relative address from the beginning of the partition but apart from this there is no address translation. Writes are performed exactly at requested addresses. If write would not change data (same data written) it will not be performed at all. If write can be performed without erasing it will be executed. If write can't be performed without an erase, then the erase is also initiated. Currently the direct driver does not support caching so writing small pieces of data on an already used sector will trigger many erase operations. If a small write is required for some reason, then `ad_nvms_erase()` should be called explicitly before each write for efficiency reasons. Power failure during write or erase will result in data loss including data that was not touched by the last write. For power fail safe operation the VES partition should be used.

b. **VES**

For an application to use VES there are two prerequisites:

- The Partition ID should only be NVMS_GENERIC_PART.
- dg_configNVMS_VES variable should be defined.

VES driver provides access to the partition with power failure protection. It uses virtual addressing. The address space available for the user application is smaller than physical flash space occupied by the partition, but user can read and write to this address space without worrying that data will be lost. If power fails during a write, the specific data being written can be lost but other data will not be affected.

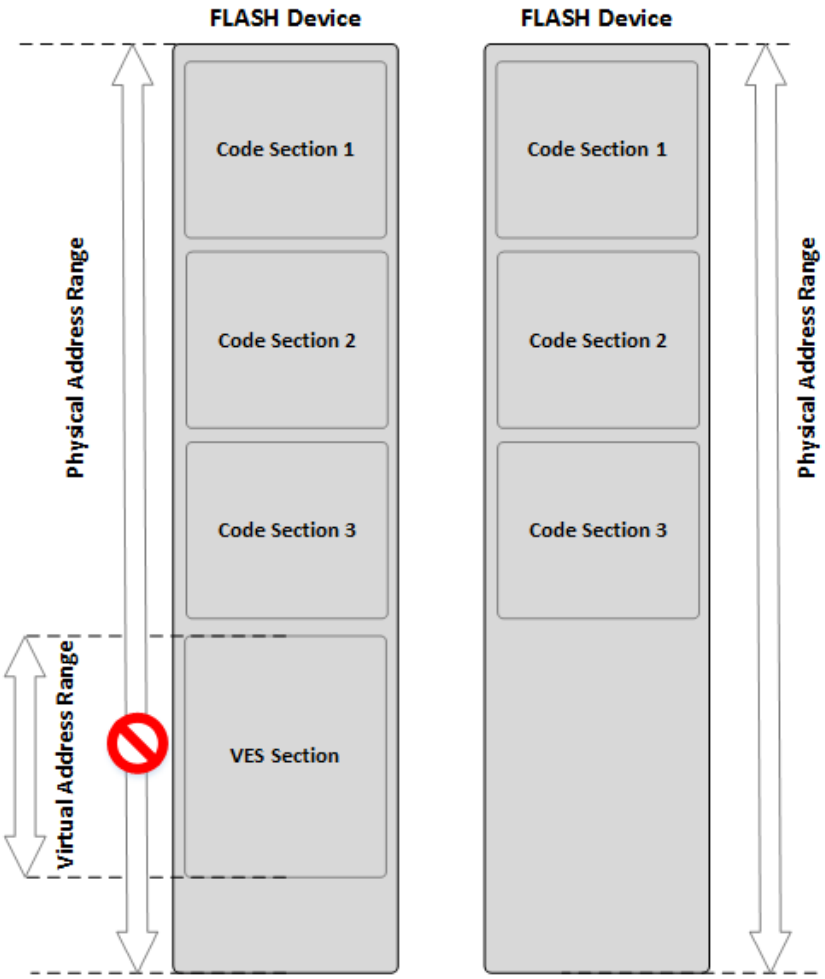

Figure 72: Virtual/Physical Addressing with and without VES

When VES is used:

- The application configures NVMS with the VES section information. The VES section is represented by a virtual address range that is mapped to a physical address range.
- NVMS prevents the application from performing “raw” writes to the allocated VES section

12.4.3 NVMS partition table

Code 34 shows the full list of the NVMS IDs and it can be found in
`<sdk_root_directory>/sdk/bsp/adapters/include/partition_def.h`

```
/**
 * \brief NVMS Partition IDs
 */
typedef enum {
        NVMS_FIRMWARE_PART              = 1,
        NVMS_PARAM_PART                 = 2,
        NVMS_BIN_PART                   = 3,
        NVMS_LOG_PART                   = 4,
        NVMS_GENERIC_PART               = 5,
        NVMS_PLATFORM_PARAMS_PART       = 15,
        NVMS_PARTITION_TABLE            = 16,
        NVMS_FW_EXEC_PART               = 17,
        NVMS_FW_UPDATE_PART             = 18,
        NVMS_PRODUCT_HEADER_PART        = 19,
        NVMS_IMAGE_HEADER_PART          = 20,
} nvms_partition_id_t;
```

Code 34: NVMS Partition IDs

Code 35 shows the format for the data of the NVMS.

```
/**
 * \brief Partition entry.
 */
typedef struct partition_entry_t {
        uint8_t magic;          /**< Partition magic number 0xEA */
        uint8_t type;           /**< Partition ID */
        uint8_t valid;          /**< Valid marker 0xFF */
        uint8_t flags;          /**< */
        uint16_t start_sector;  /**< Partition start sector */
        uint16_t sector_count;  /**< Number of sectors in partition */
        uint8_t reserved2[8];   /**< Reserved for future use */
} partition_entry_t;
```

Code 35: Partition entry

Table 49 describes each entry value.

Table 49: Description of Partition entry

Magic code (1 byte)	Partition ID (1 byte)	Valid flag (1 byte)	Flags (1 byte)	Start sector of partition (2 bytes)	Sector count of partition (2 bytes)	Reserve (8 bytes)
0xEA.	From nvm_partition_id_t, 0xFF means it's an invalid partition entry.	0xFF means it's a valid partition entry.	#define PARTITION_FLAG_READ_ONLY 1 #define RTITION_FLAG_VES 2 0 means it's a normal writable/readable direct partition. 1 means it's read only partition 2 means it's a VES partition	Start sector in flash of this partition.	Sector count of this partition.	For future use.

12.4.4 NVMS over QSPI in cached mode

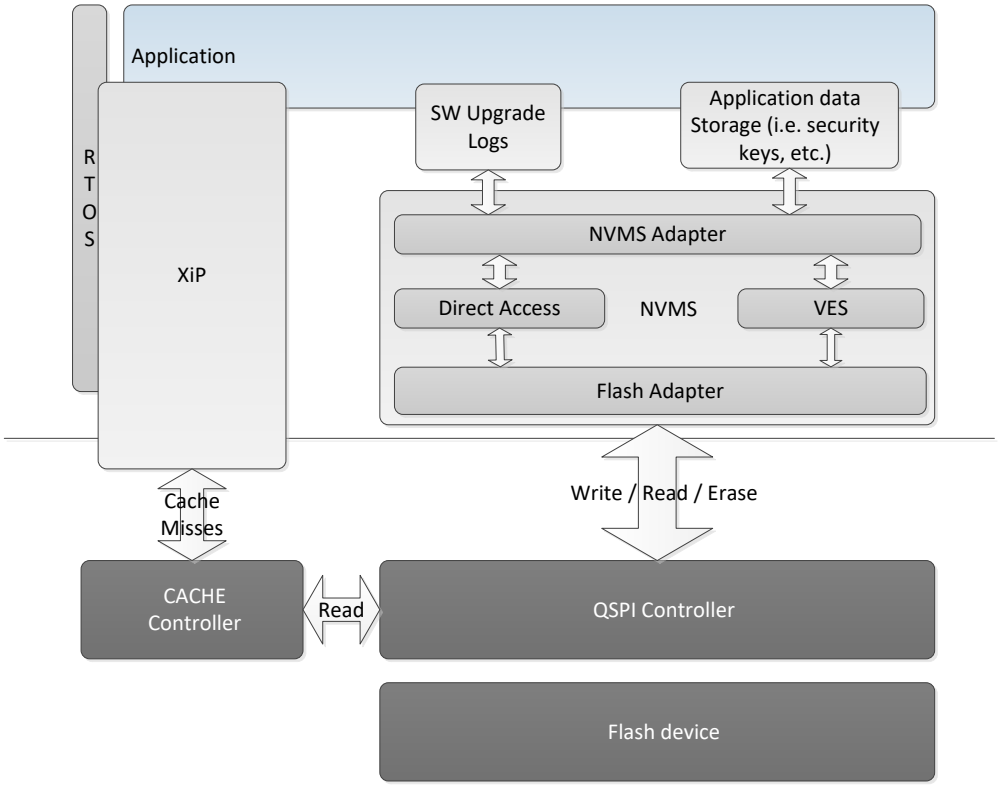

Figure 73: NVMS Adapter NVMS over QSPI and Virtual EEPROM emulation in Cached mode

When executing in place (XiP) from Flash in cached mode, the Flash device is used to store both firmware images and data.

Preemptive RTOS scheduling remains operational while programming the Flash.

PROGRAM and ERASE are the 2 critical Flash operations triggered by NVMS that need to be considered, as they can't be performed at the same time as the READ operations triggered by the cache controller when it fetches cache lines from the FLASH to the cache RAM. To handle this conflict a specific mechanism is needed.

This mechanism is disabled when DA1468x is not in cached mode.

12.4.4.1 Slice PROGRAM operation

When writing a buffer to the Flash, the NVMS Adapter will slice the buffer into several smaller buffers and will issue several uninterruptible PROGRAM QSPI requests. Note that one parameter determining the size of the slice in bytes is stored in the flash itself. This parameter is determined by:

1. The interrupt latency time in microseconds that the application authorizes.
2. The flash model and the time taken to perform a program.

In general, the first byte to be programmed takes longer than the subsequent bytes so a trade-off is possible and the exact value is left to the application developer. However, a default value of 16 bytes is currently used.

12.4.4.2 Suspend/Resume ERASE Operation

Instead of slicing ERASE, the SmartSnippets™ DA1468x SDK leverages from Smartbond™ QSPI Controller SUSPEND/ERASE in auto-mode capability. It is assumed here that all selected Flash models support suspending ERASE.

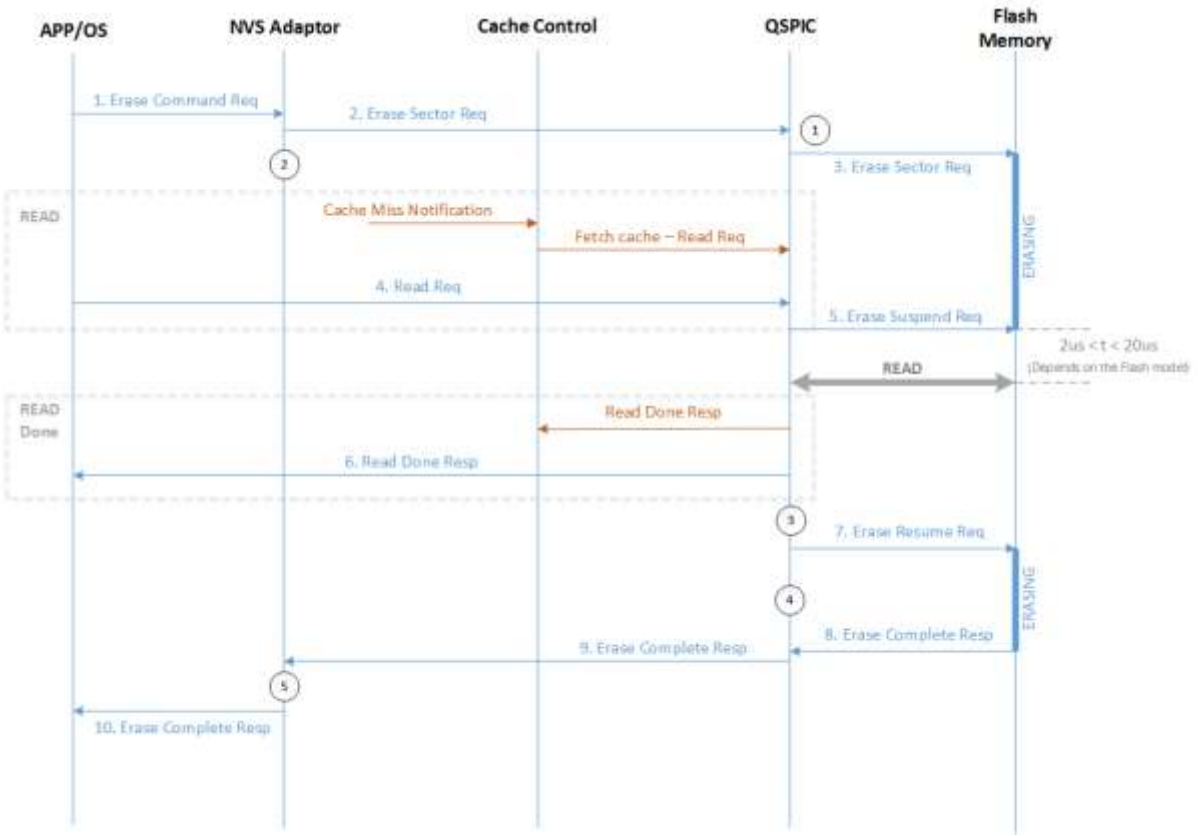

Figure 74: Suspend/Resume ERASE Operation

When requested to erase a sector by NVMS Adapter (staying in auto-mode), the QSPIC will automatically suspend the Erasing operation when a “read” from the Cache controller is triggered due to a miss hit. 2 parameters to be stored in the flash itself are SW configurable:

- ERASE/RESUME Hold: Refer to `QSPIC_ERASECMDB_REG[QSPIC_ERSRES_HLD]`
- RESUME/SUSPEND delay: Refer to `QSPIC_ERASECMDB_REG[QSPIC_RESSUS_DLY]`

As QSPIC is not firing any interrupt on Erase completion, NVMS adapter must poll `QSPIC_ERASECTRL_CMD`.

A detailed analysis of the Suspend/Resume ERASE Operation is shown in Figure 74. The overall process begins when the Application/OS issues an “Erase Command” Request. The NVMS Adapter receives the Request and issues an “Erase Sector” Request to the QSPIC (Set QSPIC_ERASECTRL_REC [QSPIC_ERASE_EN]). This request changes the QSPIC state (Figure 74, Reference Point 1), which checks whether the Flash Memory is idle for a certain number of Clock Cycles before initiating the ERASE process (QSPIC_ERASECMDB_REC [QSPIC_ERSRES_HLD]). Provided that the process can be initiated, QSPIC issues an “Erase Sector” Request to the Flash Memory and erasing begins. In the meantime, NVMS adapter also switches state, by writing CHECK_ERASE_REG and ERASECTRL_REQ bits (Figure 74, Reference Point 2). The Erase process taking place in the Flash Memory is able to be suspended by the QSPIC via an “Erase Suspend” Request in two cases

(i) if a Cache Miss Notification arrives in the Cache control, leading to a “Fetch cache – Read” Request towards the QSPIC, and

(ii) if a Read Request is issued by the Application/OS directly to the QSPIC.

The consequent “Erase Suspend” Request initiates the READ process shown in Figure 74, where QSPIC reads data from the Flash Memory and issues a “Read Done” Response to (i) Cache Control or (ii) Application/OS respectively. Then, the QSPIC state switches again, intending to resume Flash erase if memory is idle for a certain number of Clock Cycles (QSPIC_ERASECMDB_REG [QSPIC_ERSRES_HLD]) as shown in Figure 74, Reference Point 3. Provided that the Flash erase can be resumed, QSPIC issues an “Erase Resume” Request towards the Flash Memory and changes state once more. In its current state (Figure 74, Reference Point 4) QPSIC is programmed to delay any new suspension of the newly-resumed Erase for a pre-defined number of Clock Cycles (QSPIC_ERASECMDB_REG [QSPIC_RESSUS_DLY]). In the meantime, Flash Memory concludes the erasing process and notifies the QSPIC with an “Erase Complete” Response. QSPIC then issues an “Erase Complete” Response to the NVMS adapter, changing its state thus clearing CHECK_ERASE_REG and ERASECTRL_REQ bits (Figure 74, Reference Point 5). Finally, NVMS Adapter issues the conclusive “Erase Complete” Response to the Application/OS.

12.5 Logging

In the SDK the supported method of logging is to use the standard C API `printf()` in the application code. There are four mutually exclusive configuration options that will over-ride `printf()` in the file `sdk/bsp/startup/config.c`. These config options need to be set in `config/custom_config_qspi.h`

Note 29 As printf takes a variable length list of arguments and supports many formatting options the execution time and memory usage are unbounded. This can easily break an embedded system.

The recommendation is to be very careful in using printf()

Ideally use printf() in application tasks as they are lowest priority and tend to have larger stacks.

If it needs to log inside a high priority RTOS task such as a timer callback then do not pass any variables to be parsed. Just print a short string with no formatting to avoid blowing the small 100 byte stack for the timer task and corrupting other variables.

The possible configuration options are

1. `CONFIG_RETARGET`

In this mode the logging data is redirected to a UART with an over-ridden version of the low level API `_write()`. The UART used is set using the further configuration option

```
#define CONFIG_RETARGET_UART UART2
```

In this case the function periph_init() must enable the UART2 pins in the pin mux.

```
hw_gpio_set_pin_function(HW_GPIO_PORT_1, HW_GPIO_PIN_3, HW_GPIO_MODE_OUTPUT,
                                                    HW_GPIO_FUNC_UART2_TX);
hw_gpio_set_pin_function(HW_GPIO_PORT_2, HW_GPIO_PIN_3, HW_GPIO_MODE_INPUT,
                                                    HW_GPIO_FUNC_UART2_RX);
```

With this configuration the `printf()` statements appear on the host PC on the lower numbered COM port that is enumerated for the USB cable (`COMx` on Windows (both Pro DK and Basic DK) `/dev/ttyUSB0` for Pro DK and `/dev/ttyACM0` for Basic DK on Linux).

2. `dg_configSYSTEMVIEW`

In this mode the logging data is redirected to SEGGER's SystemView tool running on the Host PC. See Appendix D for instructions on setting up SystemView. Note that SystemView only supports integer arguments (ie %d)

3. `CONFIG_RTT`

In this mode the logging data is redirected to a Segger Real Time Transfer (RTT) link which uses JTAG to communicate the data to the Segger RTT tools running on the Host PC.

4. `CONFIG_NO_PRINT`

In this mode nothing is logged and in fact the `printf()` function is overridden by an empty stub function.

13 Optimizations

13.1 Optimize BLE framework footprint

This section describes macros that can be used to reduce the application image size. Usually an application implements only one of the supported Bluetooth low energy roles (for example it is only central or peripheral), so code relevant to unused Bluetooth low energy roles can be excluded from the final build. Additionally, most of the time a Bluetooth low energy application would be a GATT server or a GATT client, and therefore extra functionality can be removed. Table 50 shows the available preprocessor macros that could be used to reduce the footprint of the user application.

Table 50: Available Macros for the optimization of BLE framework footprint

Macro	Default	Description
dg_configBLE_PERIPHERAL	1	Set to 0 if the application is not using BLE-peripheral related code.
dg_configBLE_CENTRAL	1	Set to 0 if the application is not using BLE-central related code.
dg_configBLE_OBSERVER	1	Set to 0 if the application is not using BLE-observer related code.
dg_configBLE_BROADCASTER	1	Set to 0 if the application is not using BLE-broadcaster related code.
dg_configBLE_GATT_CLIENT	1	Set to 0 if the application is not using GATT client related code.
dg_configBLE_GATT_SERVER	1	Set to 0 if the application is not using GATT server related code.
dg_configBLE_L2CAP_COC	1	Set to 0 if the application is not using L2CAP connection oriented channels related code.

Note 30 All macros are defined as 1 (enabled) by default. To disable a macro define it as 0 in the project custom configuration file `config/custom_config_qspi.h` so that it will override the default setting.

As an example Code 36 shows the Macros that are defined as 0 to optimize the BLE framework footprint in the `pxp_reporter` demo application as it only needs to be a peripheral and run a GATT server.

```
* BLE device config
 */
#define dg_configBLE_CENTRAL                    (0)
#define dg_configBLE_GATT_CLIENT                (0)
#define dg_configBLE_OBSERVER                   (0)
#define dg_configBLE_BROADCASTER                (0)
#define dg_configBLE_L2CAP_COC                          (0)
```

Code 36: BLE framework preprocessor Macros

13.2 Optimizing FreeRTOS heap usage

This section discusses optimizing the FreeRTOS heap. The OS heap is a RAM buffer reserved for all dynamically allocated objects. This section focuses on how to profile FreeRTOS heap usage so that the configured heap size is optimal.

13.2.1 FreeRTOS Memory Management

Every application based on FreeRTOS must select and use the FreeRTOS memory management module. This memory management module enables objects to be dynamically allocated (by calling FreeRTOS `pvPortMalloc()` function) and eventually freed (by calling `pvPortFree()` function).

The FreeRTOS distribution provides four memory management implementations with different features and trade-offs: heap_1, heap_2, heap_3 and heap_4.

The SDK uses heap_4. This scheme permits memory to be freed, implements a simple first-fit algorithm with a coalescence algorithm that combines adjacent freed blocks into a single large block. More information about the various FreeRTOS heap_x modules can be found on www.freertos.org.

All dynamically allocated objects (allocated using `pvPortMalloc()` function) are taken from a fixed size buffer. This buffer is the FreeRTOS Heap.

The OS Heap size is defined statically in each application configuration file. Heap size is application dependent; hence each application requires a different size for the OS Heap. On one hand, the OS Heap must be big enough to support dynamic allocated memory requirements. On the other hand, the Retention Memory budget impacts Power Consumption while sleeping, so heap needs to be minimized. The goal is to have just enough heap memory allocated for the application.

FreeRTOS Heap contains the OS execution contexts of the application tasks so it needs to be retained. A look at the map file (`*.map`) generated during build, shows the amount of data retained by heap_4.o module (the SDK redefines `privileged_data` to mean data that needs to be in retained RAM section 6.1.3). The application `privileged_data` is the major part of the retained memory budget, with the remainder being retained static variables from other modules like tasks, queues and timers.

```
12456                   0x07fda288              last_calib_temp
12457   privileged_data_zi
12458                   0x07fda28c      0x3418 ./sdk/FreeRTOS/portable/MemMang/heap_4.o
12459   privileged_data_zi
12460                   0x07fdd6a4      0x40 ./sdk/FreeRTOS/queue.o
12461                   0x07fdd6a4              xQueueRegistry
12462   privileged_data_zi
12463                   0x07fdd6e4      0xfc ./sdk/FreeRTOS/tasks.o
12464                   0x07fdd6e4              pxCurrentTCB
12465                   0x07fdd6e8              pxDelayedTaskList
12466                   0x07fdd6ec              pxOverflowDelayedTaskList
12467                   0x07fdd6f0              xSuspendedTaskList
12468                   0x07fdd704              xTasksWaitingTermination
12469                   0x07fdd718              pxReadyTasksLists
12470                   0x07fdd77c              uxTopReadyPriority
12471                   0x07fdd780              xPendingReadyList
12472                   0x07fdd794              uxSchedulerSuspended
12473                   0x07fdd798              xSchedulerRunning
12474                   0x07fdd79c              xTickCount
12475                   0x07fdd7a0              uxCurrentNumberOfTasks
12476                   0x07fdd7a4              xYieldPending
12477                   0x07fdd7a8              xNumOfOverflows
12478                   0x07fdd7ac              xDelayedTaskList1
12479                   0x07fdd7c0              xDelayedTaskList2
12480                   0x07fdd7d4              uxTaskNumber
12481                   0x07fdd7d8              uxTasksDeleted
12482                   0x07fdd7dc              uxPendedTicks
12483   privileged_data_zi
12484                   0x07fdd7e0      0x38 ./sdk/FreeRTOS/timers.o
```

Figure 75: Amount of data retained by the heap_4.o module

Table 51: Amount of data retained by the FreeRTOS for this specific example

FreeRTOS	Retention	
	Code	Data
heap_4.o	0	13336
queue.o	0	64
tasks.o	0	252
timers	0	56
ad_nvms_ves.o	0	4
Total	0	13712

13.2.2 OS Heap & Tasks Stack size

OS Heap Size is application dependent. In FreeRTOS whenever a heap allocation cannot be serviced, the hook `vApplicationMallocFailedHook()` is called to handle the error.

In addition to the heap, each Task created by the application requires its own stack. The stack is allocated form the heap and its size should be:

- Big enough so that its stack pointer remains in the stack. FreeRTOS checks on every context switch whether the stack has overflowed and calls `vApplicationStackOverflowHook()` when an overflow is detected.
- As small as possible, so that Power Consumption while sleeping is limited (section 13.3).

The size of the required stack varies according to the number of nested function calls, the number of parameters that are passed in function calls and the number and the type of local variables in functions.

If the “worst case” execution path is known, it might be easy to calculate the optimal stack size for a task. However, knowing this “worst-case” execution path is not always simple, so a more practical method is proposed in the next paragraph.

13.2.3 Optimizing FreeRTOS Heap

The following steps describe a practical and empirical approach to the optimization of the FreeRTOS Heap memory.

Step 1:

For every task, the application developer should continuously monitor and optimize stack size. This should be done as early as possible during the development phase and eventually during testing as well. FreeRTOS provides the `uxTaskGetStackHighWaterMark()` function that helps measuring Stack utilization. All applications tasks need to be monitored. SDK middleware tasks such as BLE Adapter, BLE Manager and USB charger should also be tracked.

Step 2:

Continuously Monitor and adjust OS Heap utilization. This assumes that every task stack is being continuously monitored and optimized. FreeRTOS provides a `xPortGetMinimumEverFreeHeapSize()`

function that helps measuring Total heap utilization. To minimize the impact on the application Task Stack and total Heap monitoring should:

- Be done when the application is Idle. Therefore the preferred place in the context of FreeRTOS is in `vApplicationIdleHook()`.
- Not require a bigger Idle stack size.

SmartSnippets™ DA1468x SDK provides a development configuration that activates OS Heap monitoring. The project `<sdk_root_directory>\projects\dk_apps\demos\ble_adv` contains code to monitor FreeRTOS Heap usage. To enable this, the user needs to enable `dg_configTRACK_OS_HEAP` macro inside the `config/custom_config_qspi.h` file as shown in Code 37.

```
//
// Enable the settings below to track OS heap usage, for profiling
//
//#if (dg_configIMAGE_SETUP == DEVELOPMENT_MODE)
//#define dg_configTRACK_OS_HEAP                  1
//#else
//#define dg_configTRACK_OS_HEAP                  0
//#endif
```

Code 37: Enabling FreeRTOS Heap Tracking

13.3 Retention RAM optimization and configuration

Note 31 By default, retention RAM optimization is disabled.

This section describes the different retention RAM configurations that can be used to achieve the lowest possible power consumption for a specific application. As described in [1], the memory controller of the DA1468x chip provides a unified memory space for the RAM while allowing the reshuffling of the first 3 RAM cells. This way, it is possible for the application to retain only the absolutely required amount of RAM, thus saving power.

There are five different RAM cells in total, as shown in Figure 76. Each one can be selected to be retained or not. The 5th RAM cell must always be retained because BLE ROM variables are stored in it. The memory region `[0x7fc0000 – 0x7fc0200]` must also be retained because the Interrupt Vector Table (IVT) is stored in it.

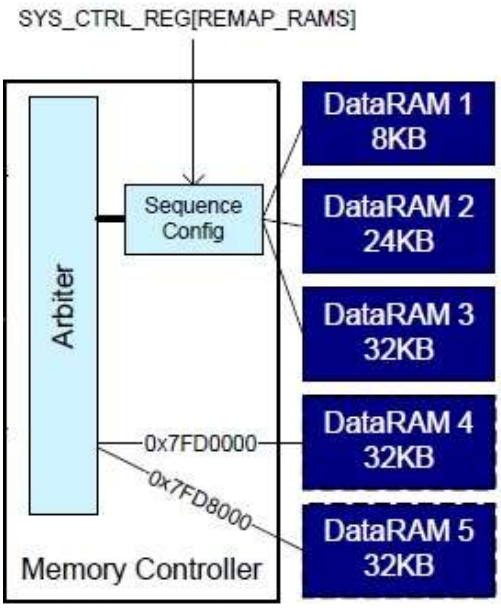

Figure 76: Memory blocks

The Sequence Configuration block controls the ordering of the first 3 memory cells according to the REMAP_RAMS value. Shuffling the memory cells allows the minimum number of RAM blocks to be retained to get the lowest power-down current consumption. The configuration setting dg_configSHUFFLING_MODE encodes the ordering of the first 3 RAM cells (RAM1, RAM2 and RAM3). The possible configurations are listed in Table 52.

Table 52: DataRAM cells sequence

Value	Cell	Address	Size (kB)
0x0	DataRAM1	0x7FC0000	8
	DataRAM2	0x7FC2000	24
	DataRAM3	0x7FC8000	32
0x1	DataRAM2	0x7FC0000	24
	DataRAM1	0x7FC6000	8
	DataRAM3	0x7FC8000	32
0x2	DataRAM3	0x7FC0000	32
	DataRAM1	0x7FC8000	8
	DataRAM2	0x7FCA000	24
0x3	DataRAM3	0x7FC0000	32
	DataRAM2	0x7FC8000	24
	DataRAM1	0x7FCE000	8

For example, if 50 KB of RAM needs to be retained then the optimal dg_configSHUFFLING_MODE value is 0x1. In this way, RAM5 cell (mandatory, 32 KB) and RAM2 cell (mapped at 0x7fc0000 after shuffling, 24 KB) may be retained by setting the configuration macro dg_configMEM_RETENTION_MODE to 0x1D, resulting in 56 KB of total retained RAM.

For the DA1680/1-01, there are three different memory layouts depending on the build configuration. The following sections describe these memory layouts in detail.

In the linker scripts of the applications distributed with the SDK, there are typically two retained memory sections defined. The first section, RetRAM0, contains the BLE ROM variables and the exchange table used for the communication with the BLE core, any code that must be retained and the zero initialized and RW variables retained by the application. The other section, RetRAM1, may contain, apart from the IVT, large blocks of zero-initialized retained data, like OS or BLE heaps. Other allocations of which type of data is placed in each section are possible by modifying the linker script accordingly.

Note 32 Linker scripts are centralized and so there is not a dedicated linker script per project. Centralized linker scripts are backwards compatible and can be overridden by a custom linker script if needed so. There are two flavors of linker scripts one for BLE and one for non-BLE projects. Centralized linker scripts are found in <sdk_root_folder>/sdk/bsp/ldscripts/.

13.3.1 Memory setups for QSPI Cached execution mode

The following sections describe various memory configurations for executing the application code from the QSPI flash memory. In these memory setups, the application code located in the QSPI Flash memory is executed in place and the Cache memory is enabled.

The presented projects are BLE and non-BLE and they are split into two categories:

- non-optimized (all RAM cells are retained)
- optimized for low power consumption (some RAM cells are retained)

13.3.1.1 DA14680/681 – QSPI Cached BLE non-optimized project (all RAM cells are retained)

In Figure 77, an example memory layout is given. Here the non-optimized BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses RAM cells 4 (32 KB) and 5 (32 KB). In this setup, the overall retained memory is 128 KB (all RAM cells are retained).

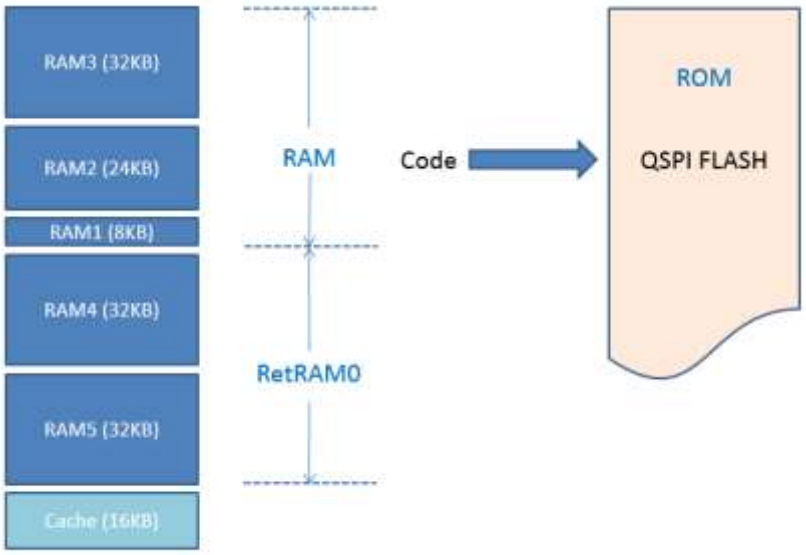

Figure 77: DA14680/681 – QSPI Cached BLE non-optimized project

Note 33 The BLE ROM variables start at 0x07FDC000 address.

13.3.1.2 DA14680/681 – QSPI Cached BLE optimized project (RAM1, RAM2, RAM4, RAM5 cells are retained)

In Figure 78, an example memory layout is given. Here the optimized BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses RAM4 and RAM5 cells (64 KB) while RetRAM1 uses RAM1 and RAM2 cells (32 KB). In this setup, the overall retained memory is 96 KB.

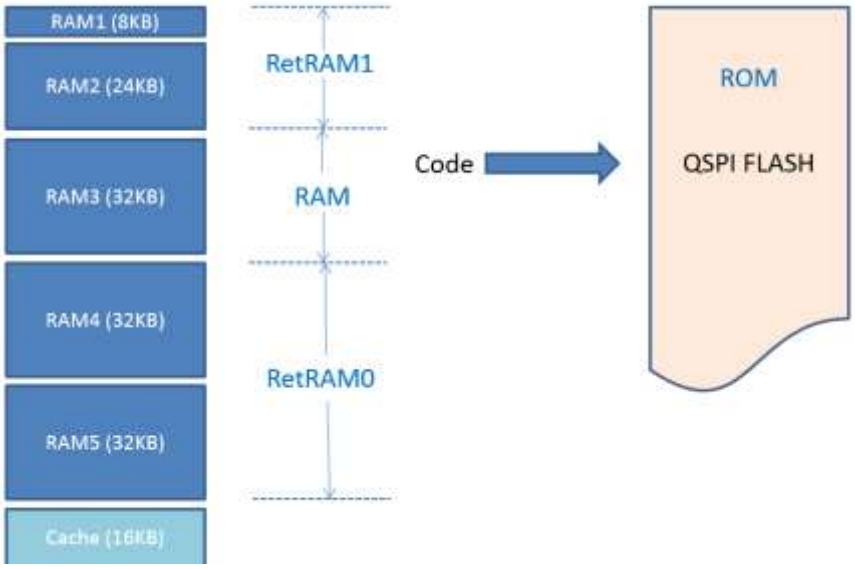

Figure 78: DA14680/681 – QSPI Cached BLE optimized project

Retention RAM optimization settings

The following defines must be added in the `config/custom_config_qspi.h` file of the application to enable RAM optimization.

```
#define dg_configOPTIMAL_RETRAM                 (1)
#define dg_configMEM_RETENTION_MODE             (0x1B)
#define dg_configSHUFFLING_MODE                 (0x0)
```

An example taken from `pxp_reporter` demo application is shown in Code 38.

```
#define dg_configOPTIMAL_RETRAM                 (1)

#if (dg_configOPTIMAL_RETRAM == 1)
        #if (dg_configBLACK_ORCA_IC_REV == BLACK_ORCA_IC_REV_A)
                #define dg_configMEM_RETENTION_MODE             (0x1B)
                #define dg_configSHUFFLING_MODE                 (0x0)
        #else
                #define dg_configMEM_RETENTION_MODE             (0x07)
                #define dg_configSHUFFLING_MODE                 (0x0)
        #endif
#endif
```

Code 38: RAM optimization settings

Note 34 BLE ROM variables start at `0x07FDC000` address.

13.3.1.3 DA14680/681 – QSPI non-BLE non-optimized project (all RAM cells are retained)

In Figure 79, an example memory layout is given. Here the non-optimized non-BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses RAM cells 4 (32 KB) and 5 (32 KB). In this setup, the overall retained memory is 128 KB (all RAM cells are retained).

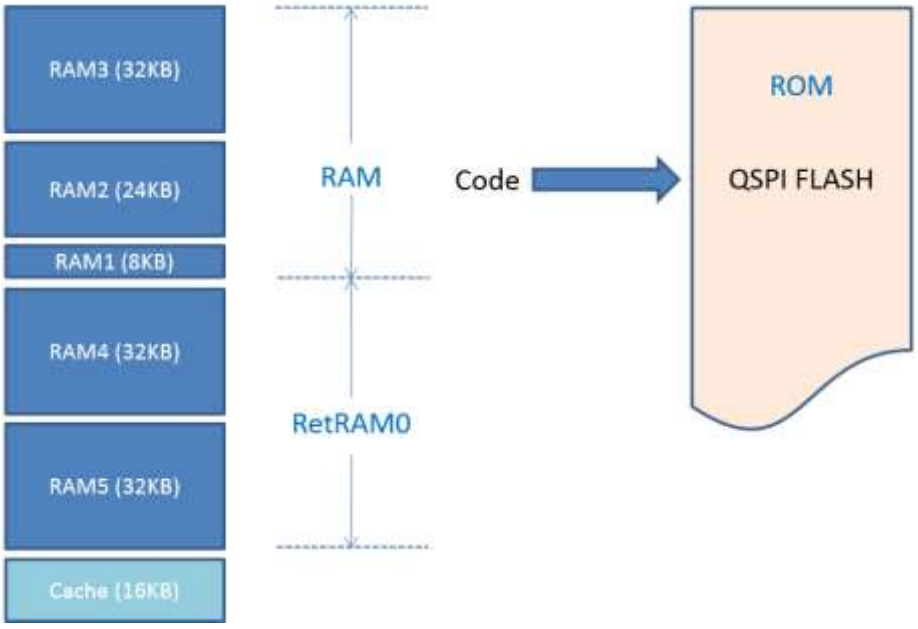

Figure 79: DA14680/681 – QSPI non-BLE non-optimized project

13.3.1.4 DA14680/681 – QSPI non-BLE optimized project (RAM2 cell is retained)

In Figure 80, an example memory layout is given. Here the optimized non-BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses only RAM cell 2 (24 KB). In this setup, the overall retained memory is 24 KB.

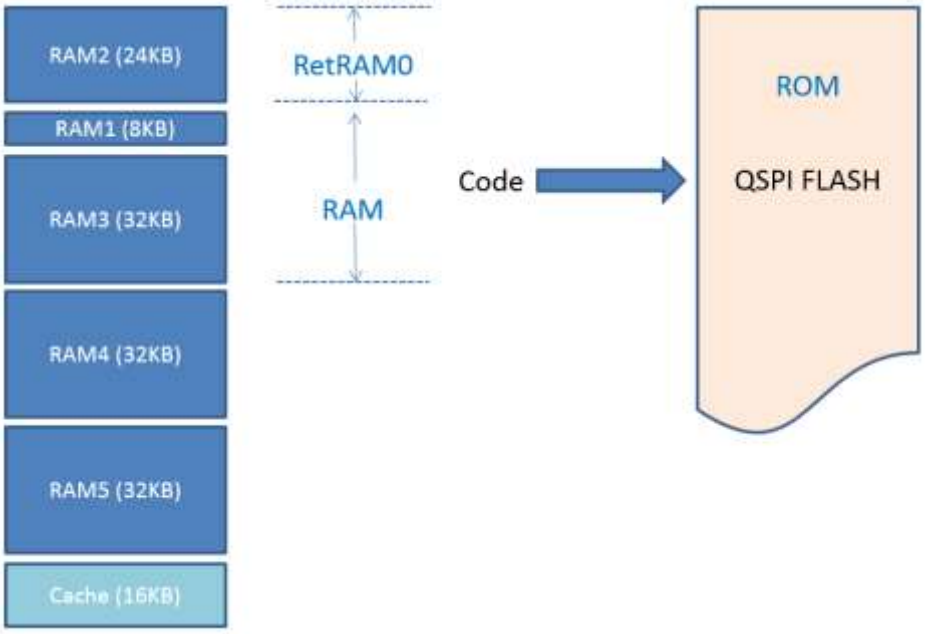

Figure 80: DA14680/681 – QSPI non-BLE optimized project

Retention RAM optimization settings

The following defines must be added in the `custom_config_qspi.h` file of the application to enable RAM optimization.

```
#define dg_configOPTIMAL_RETRAM                 (1)
#define dg_configMEM_RETENTION_MODE             (0x02)
#define dg_configSHUFFLING_MODE                 (0x1)
```

13.3.1.5 DA14682/683, DA15100/1 – QSPI Cached BLE non-optimized project (all RAM cells are retained)

In Figure 81, an example memory layout is given. Here the non-optimized BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses RAM cells 3 (32 KB), 2 (24 KB) and 1 (8 KB). In this setup, the overall retained memory is 128 KB.

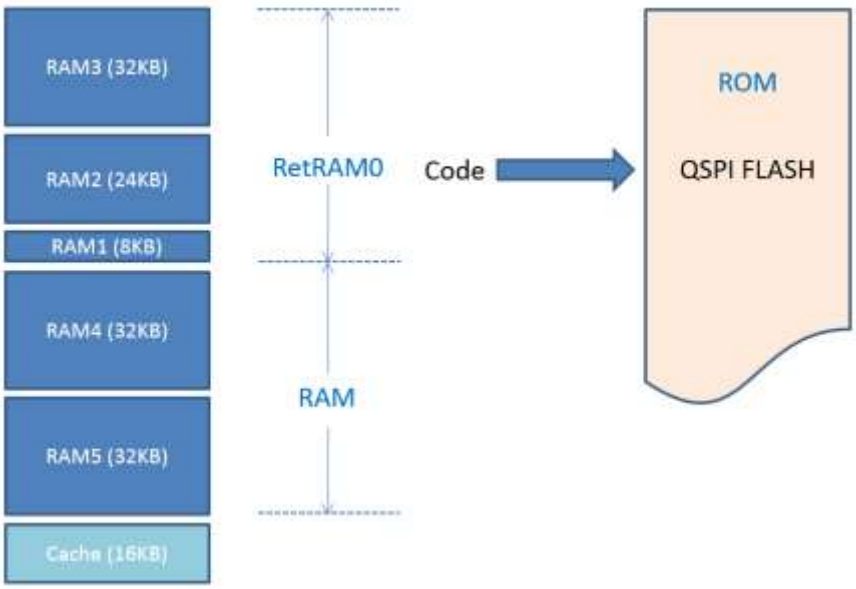

Figure 81: DA14682/683, DA15100/1 – QSPI Cached BLE non-optimized project

Note 35 BLE ROM variables start at 0x07FC0200 address.

13.3.1.6 DA14682/683, DA15100/1 – QSPI Cached BLE optimized project (RAM1, RAM2, RAM3 cells are retained)

In Figure 82, an example memory layout is given. Here the optimized BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses RAM cells 1 (8 KB), 2 (24 KB) and 3 (32 KB). In this setup, the overall retained memory is 64 KB.

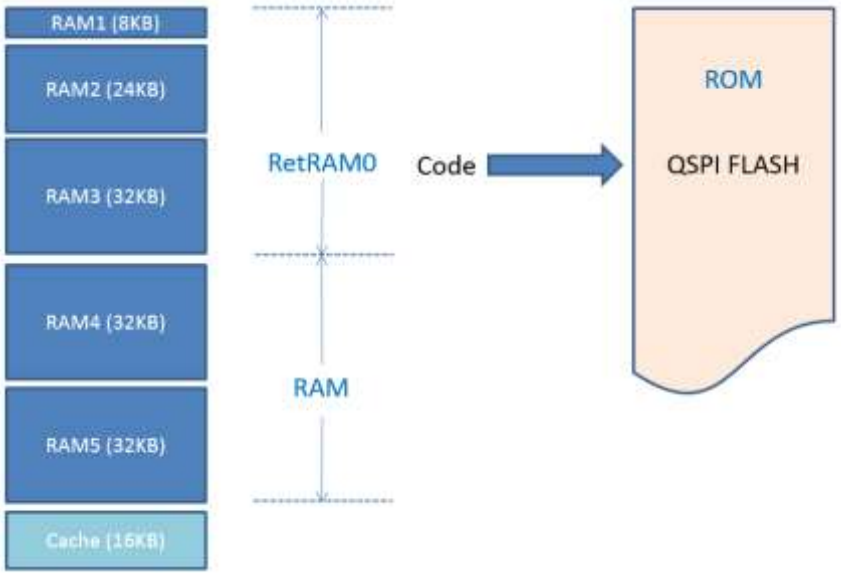

Figure 82: DA14682/683, DA15100/1 – QSPI Cached BLE optimized project

Retention RAM optimization settings

The following defines must be added in the `config/custom_config_qspi.h` file of the application to enable RAM optimization.

```
#define dg_configOPTIMAL_RETRAM                 (1)
#define dg_configMEM_RETENTION_MODE             (0x07)
#define dg_configSHUFFLING_MODE                 (0x0)
```

Note 36 BLE ROM variables start at `0x07FC0200` address.

13.3.1.7 DA14682/683, DA15100/1 – QSPI Cached non-BLE non-optimized project (all RAM cells are retained)

In Figure 83, an example memory layout is given. Here the non-optimized non-BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses RAM cells 1 (8 KB), 2 (24 KB) and 3 (32 KB). In this setup, the overall retained memory is 128 KB (all RAM cells are retained).

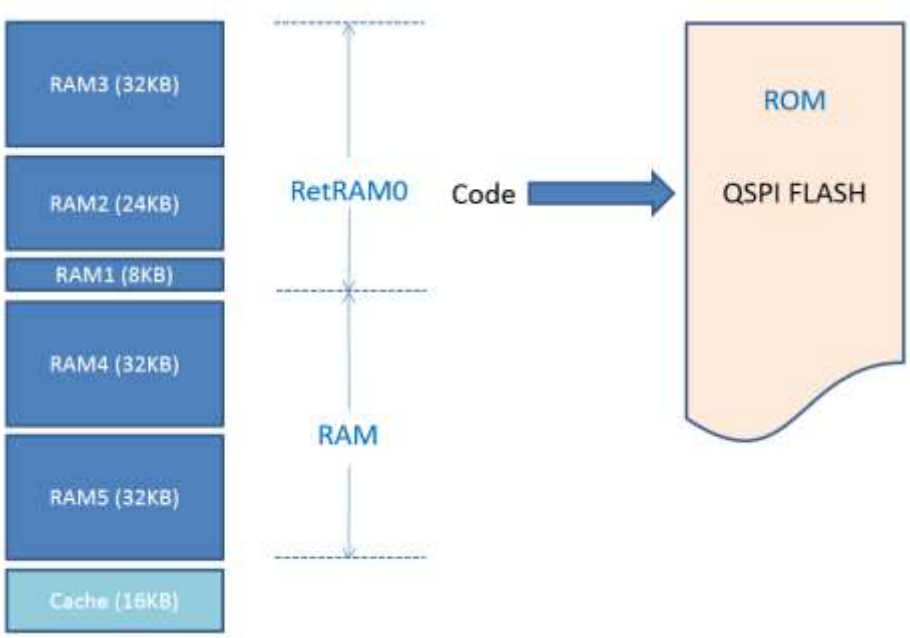

Figure 83: DA14682/683, DA15100/1 – QSPI non-BLE optimized project

13.3.1.8 DA14682/683, DA15100/1 – QSPI non-BLE optimized project (RAM2 cell is retained)

In Figure 84, an example memory layout is given. Here the optimized non-BLE project is executed in QSPI cached mode. In this example, RetRAM0 uses only RAM cell 2 (24 KB). In this setup, the overall retained memory is 24 KB (all RAM cells are retained).

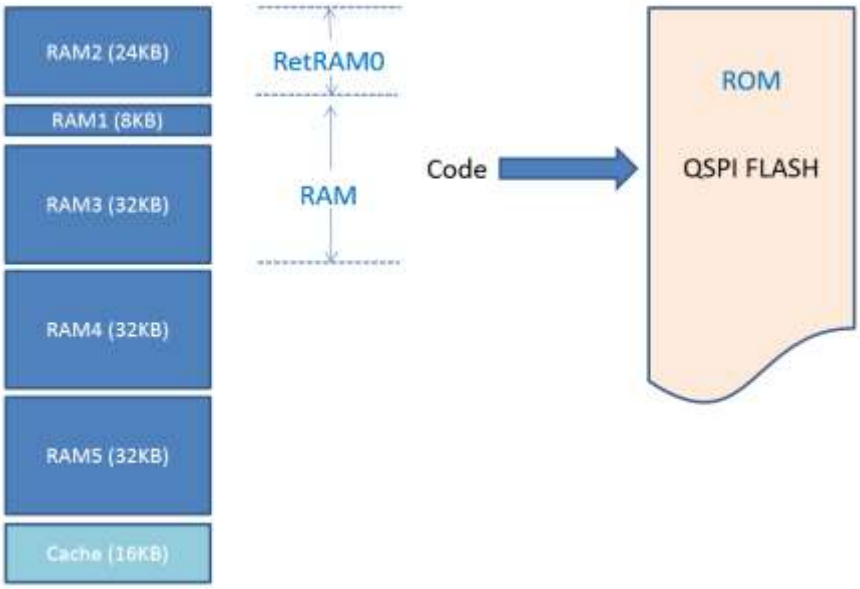

Figure 84: DA14682/683, DA15100/1 – QSPI non-BLE optimized project

Retention RAM optimization settings

The following defines must be added in the `config/custom_config_qspi.h` file of the application to enable RAM optimization.

```
#define dg_configOPTIMAL_RETRAM                 (1)
#define dg_configMEM_RETENTION_MODE             (0x02)
#define dg_configSHUFFLING_MODE                 (0x1)
```

13.3.2 Memory setups for RAM execution mode

This section describes several configurations that allow executing the application code from RAM memory (although defined as ROM in linker). In these memory configurations the application code located in the defined as ROM retained RAM memory cells, is executed in place. The Cache memory is enabled.

The presented projects are BLE and non-BLE and they are split into two categories:

- non-optimized (all RAM cells are retained)
- optimized for low power consumption (some RAM cells are retained)

13.3.2.1 DA14680/681 – RAM BLE non-optimized project (all RAM cells are retained)

In Figure 85, an example memory layout is given. Here the non-optimized BLE project is executed in RAM execution mode. In this example the ROM memory has 64 KB size and uses RAM cells 1 (8 KB), 2 (24 KB) and 3 (32 KB). RetRAM0 uses RAM cell 4 (32 KB) and 5 (32 KB). In this setup, the overall retained memory is 128 KB (all RAM cells are retained).

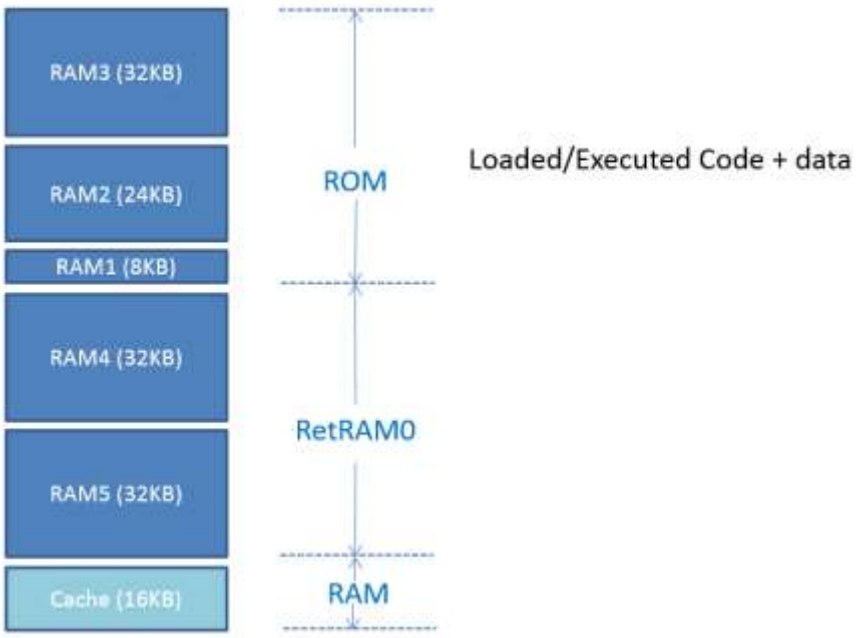

Figure 85: DA14680/681 – RAM BLE non-optimized project

Note 37 BLE ROM variables start at 0x07FDC000 address.

13.3.2.2 DA14680/681 – RAM non-BLE non-optimized project (all RAM cells are retained)

In Figure 86, an example memory layout is given. Here the non-optimized non-BLE project is executed in RAM execution mode. In this example the ROM memory has 64 KB size and uses RAM cells 1 (8 KB), 2 (24 KB) and 3 (32 KB). RetRAM0 uses RAM cell 4 (32 KB) and 5 (32 KB). In this setup, the overall retained memory is 128 KB (all RAM cells are retained).

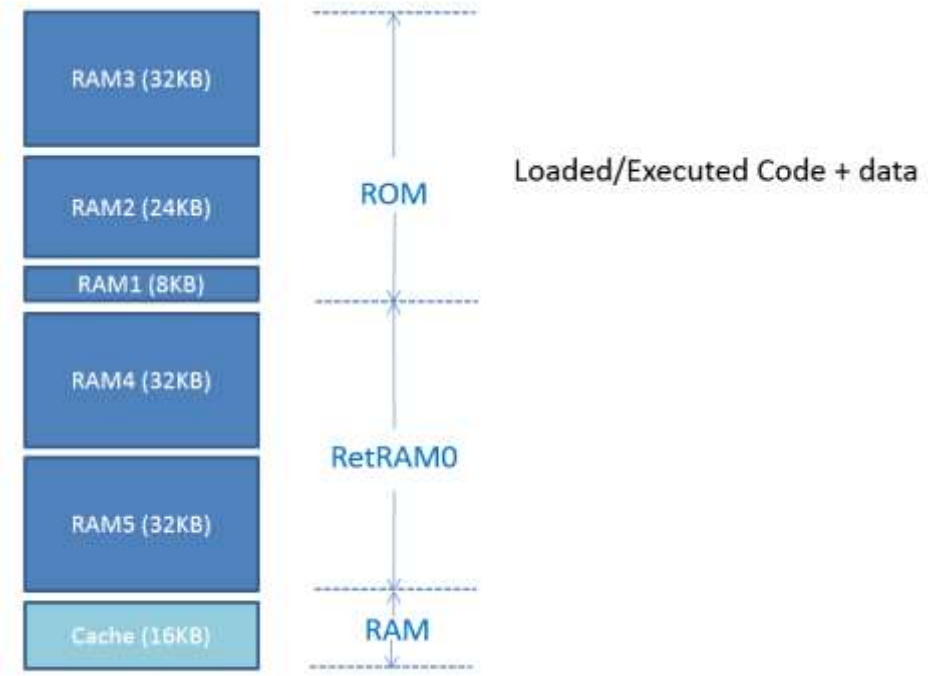

Figure 86: DA14680/681 – RAM non-BLE non-optimized project

13.3.2.3 DA14682/683, DA15100/1 – RAM BLE non-optimized project (all RAM cells are retained)

In Figure 87, an example memory layout is given. Here the non-optimized BLE project is executed in RAM execution mode. In this example the ROM memory has 128 KB size. RetRAM0 uses RAM cell 3 (32 KB. In this setup, the overall retained memory is 128 KB (all RAM cells are retained).

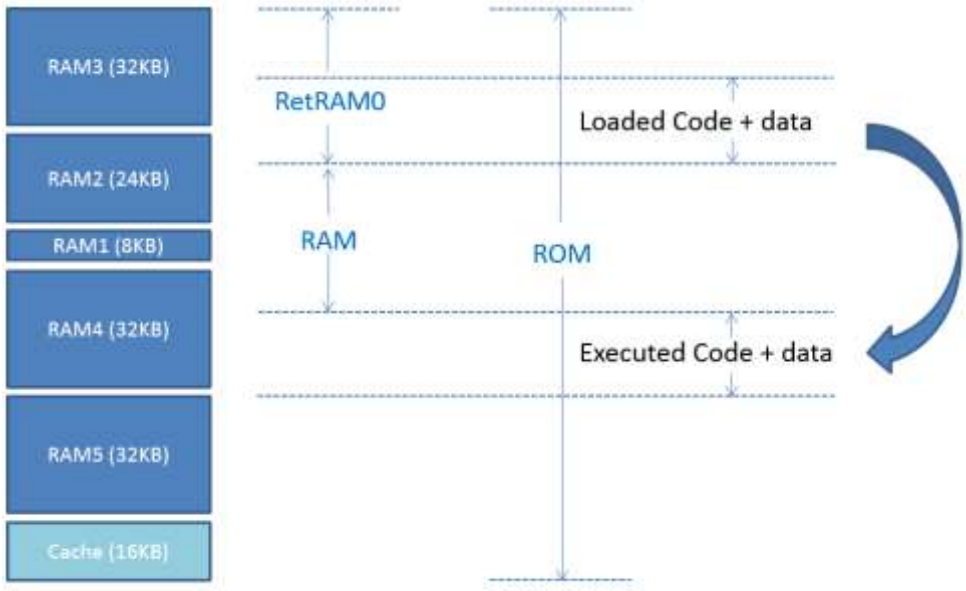

Figure 87: DA14682/683, DA15100/1 – RAM BLE non-optimized project

Note 38 BLE ROM variables start at `0x07FC0200` address.

13.3.2.4 `DA14682/683, DA15100/1` – RAM non-BLE non-optimized project (all RAM cells are retained)

In Figure 88, an example memory layout is given. Here the non-optimized BLE project is executed in RAM execution mode. In this example the ROM memory has 128 KB size. RetRAM0 uses RAM cell 3 (32 KB. In this setup, the overall retained memory is 128 KB (all RAM cells are retained).

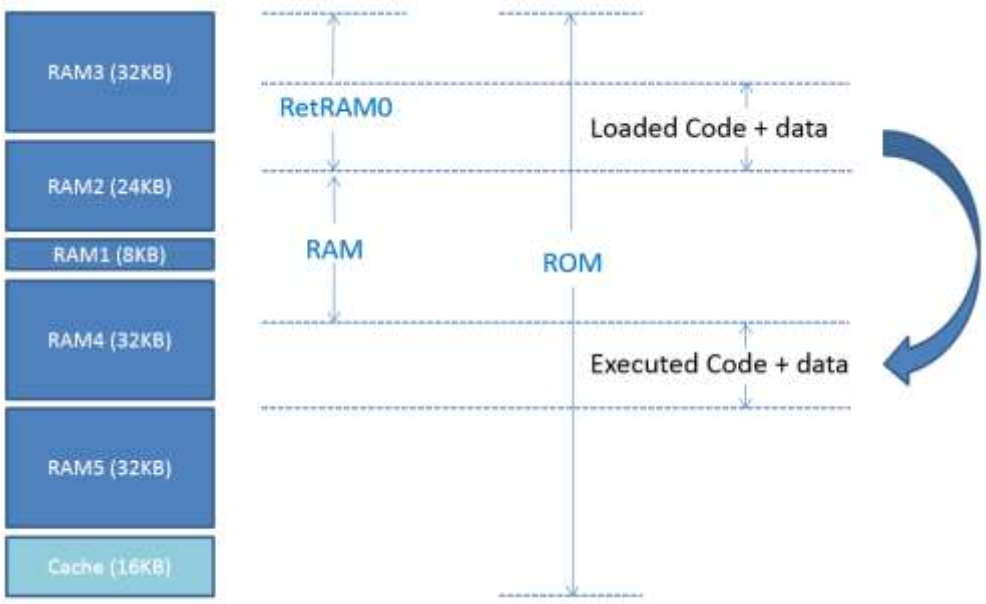

Figure 88: `DA14682/683, DA15100/1` – RAM non-BLE non-optimized project

13.3.3 Memory setup for the OTP Cached execution mode (DA14680/1-01)

The only difference with the QSPI cached mode is that the code resides in the OTP instead of the QSPI Flash. Due to the limited size of the OTP, the maximum size of the application image is 58 KB. Figure 89 depicts the memory setup for an OTP cached application with image size 58 KB, retaining 64 KB of RAM during sleep (cells 1, 2 and 5).

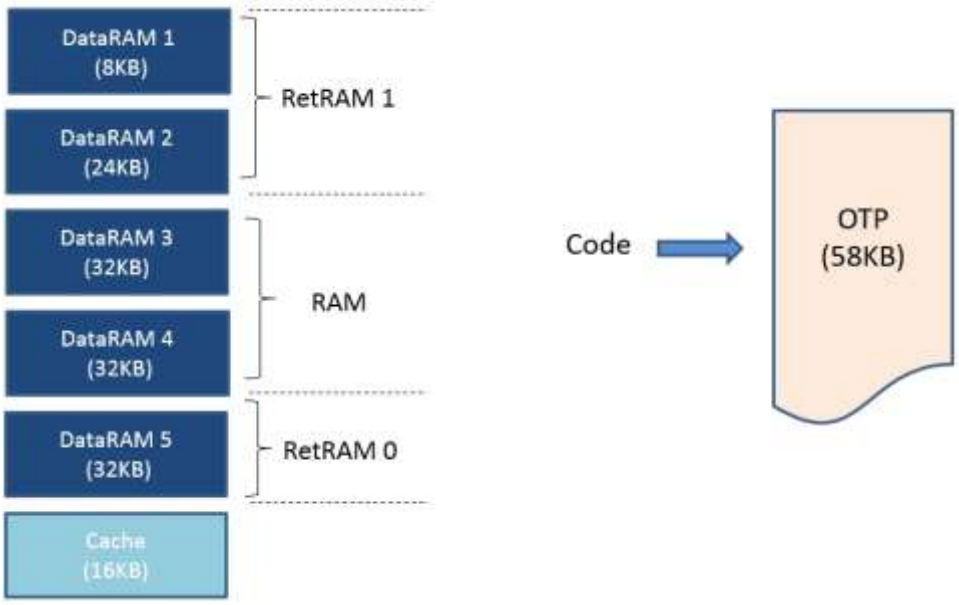

Figure 89: Memory setup for the OTP Cached execution mode (DA14680/1-01)

13.3.4 Memory setup for the OTP Mirrored execution mode (DA14680/1-01)

Note 39 The SDK has been developed to support only cached mode from Flash. This section is provided only as reference.

In mirrored mode, the application code is copied into RAM before it is executed. There are two available memory setups for the OTP Mirrored execution mode. The main difference is whether 1 or 2 sections will be used for non-retained data. In both configurations Cache memory is off and so may be used by the application. There is no RetRAM1 section since all memory is retained in this mode. The application code has to be retained during sleep as there is no mechanism to restore it at wake-up. Due to this requirement, RAM cell shuffling is irrelevant while the value of the `dg_configMEM_RETENTION_MODE` must be `0x1F`.

In the first configuration, the RAM section of the linker script (normally non-retained data) is placed in the RAM cell of Cache. The ECC buffer cannot be placed in this region. This setup is shown in Figure 32.

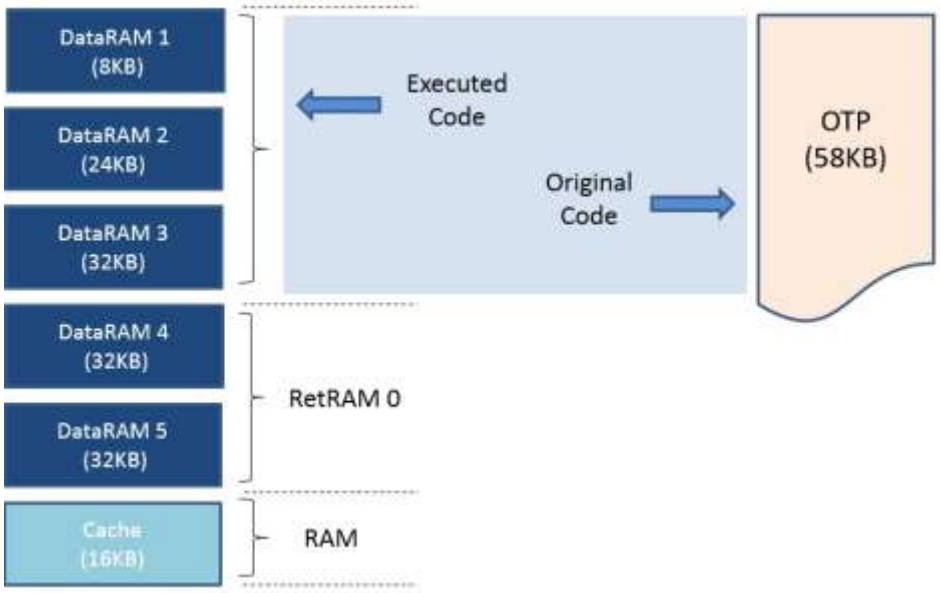

Figure 90: Setup 1 for the OTP Mirrored execution mode (DA14680/1-01)

In the second configuration, the linker script has two RAM sections. The first one (RAM1) is placed in the RAM cell of the Cache as in the first configuration. Here there is also space after the end of the application image in RAM and the beginning of RetRAM0 section which can also be used for data (RAM2). Note that the ECC buffer can be placed in RAM2. This configuration is shown in Figure 91.

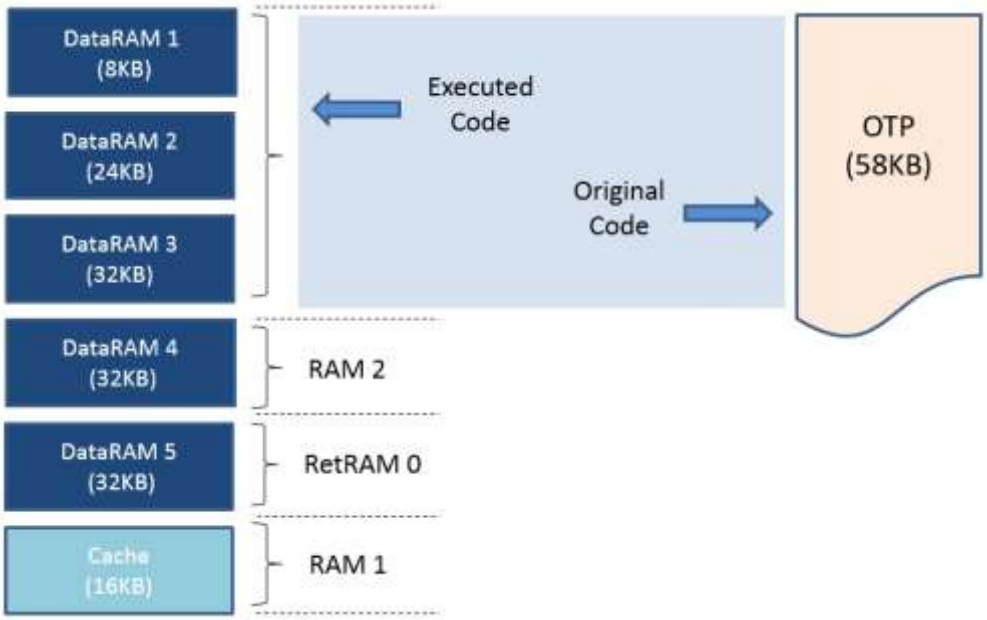

Figure 91: Setup 2 for the OTP Mirrored execution mode (DA14680/1-01)

Appendix A SmartSnippets DA1468x SDK structure

A.1 Directory structure

This section describes the structure of the directories of the SmartSnippets™ DA1468x SDK. The root directory contains the subfolders shown below. Each subdirectory is described in the following sections.

Table 53: SmartSnippets™ root directory structure

Directory	Subdirectory	Description
DA1468x_SDK_BTLE_v_1.0.10.xxx	*binaries*	Executables.
	config	Configuration files for the SmartSnippets DA168x SDK.
	doc	Documentation.
	projects	Project examples.
	sdk	Software development kit.
	utilities	Utilities (image creation, programming etc.).

A.2 Binaries directory

The Binaries directory contains the executable binaries of the Windows and Linux applications which are needed to generate the final image file and to interact with the Pro and Basic DK boards. Currently these binaries are in the binaries folder of the SmartSnippets™ DA1468x SDK:

Table 54 binary files inside SmartSnippets™ DA1468x SDK

Filename	Notes
bin2image	Utility for creating a bootable image from an executable raw binary (for Linux).
bin2image.exe	Utility for creating a bootable image from an executable raw binary (for Windows).
cli_programmer	Utility to download image file to ProDK development board (for Linux).
cli_programmer.exe	Utility to download image file to ProDK development board (for Windows).
Libprogrammer.dll	Utility which supports download/programming of image files to ProDK board (for Windows).
Libprogrammer.so	Utility which supports download/programming of image files to ProDK board (for Linux).
Mkimage	Utility for creating a firmware image for SmartSnippets DA1468x SDK (for Linux).
mkimage.exe	Utility for creating a firmware image for SmartSnippets DA1468x SDK (for Windows).

A.3 Config directory

The Config directory contains configuration files for the SmartSnippets™ DA1468x SDK.

Table 55: Config folder

Documents	Description
Embsys	DA1468x register definitions
DA1468x_1.0.10_SDK_config	Configuration file for the SmartSnippets DA1468x SDK.
ATTACH.launch	Global debug configuration launcher
QSPI.launch	Global debug configuration launcher
RAM.launch	Global debug configuration launcher
studio_config	SmartSnippets Studio configuration

A.4 Doc directory

The Doc directory shall contain all relevant documentation which makes sure the reader understands the set-up and can use the SmartSnippets™ DA1468x SDK version 1.0.10. At present the folder contains the following documents:

Table 56: Doc folder

Documents	Description
Html folder, Doxygen	SmartSnippets DA1468x SDK Documentation generated by Doxygen.
Installation_and_debugging_procedure	The file presents steps which should be done by a user to properly run example demos from SDK.
Licensing	License Agreement for the software.
sdk_eclipse_formatter.xml	Coding style formatter description for Eclipse.
VERSION	Version of the SmartSnippets DA1468x SDK.

A.5 Projects directory

A.5.1 dk_apps directory

The dk_apps directory contains a collection of sample projects which can run on the DA1468x family of devices. These projects all use the board support package which consists of all peripherals drivers, the RTOS, the BLE stack if applicable, etc. This will be described in more detail in a later section of this document. The following subfolders are present in the dk_apps directory.

Table 57: dk_apps directory structure

Directory	Subdirectory	Description
dk_apps	ble_profiles	Sample projects implementing BLE available profiles.
	demos	Sample projects, such as standalone peripheral tests.
	features	Contains projects that exhibit some basic features of the SmartSnippets DA1468x SDK.
	reference_designs	Contains all reference design projects, which are out of scope of the SmartSnippets DA1468x SDK. As an exception, we include plt_fw in a form of a “reference design”.
	templates	Several projects which offer templates for different device configurations such as support of a real-time terminal for easier

Directory	Subdirectory	Description
		debugging.

A.5.2 Host_apps directory

Table 58: Host App directory

Directory	Subdirectory	Description
host_apps	*Examples*	Includes usb_cdc_echo_test

A.5.3 SDK directory

The SDK directory is the central part of the complete SmartSnippets™ DA1468x SDK and contains the code and header files for BLE framework support, middleware components (such as e.g. audio, firmware upgrade and security toolboxes) as well as low-level drivers and adapter implementations. The directories are structured as shown in the tables below.

Table 59: SDK directory structure

Directory	Subdirectory	Description
sdk	*bsp*	Includes header and source files for board support package.
	Interfaces	Includes header and source files for supported interfaces such as e.g. audio/crypto toolbox, BLE adapter, etc.
	Middleware	Includes header and source files for supported middleware and services such as e.g. fw_upgrade and logging support.

Table 60: bsp directory structure

Directory	Subdirectory	Description
bsp	*adapters*	Contains projects for all supported adapters such as e.g. SPI, I2C and UART adapter.
	config	Configuration header files.
	free_rtos	Contains projects for FreeRTOS implementation.
	Include	HW specific 14680 macros, structure and register definitions, interrupt priority definitions and low level API calls.
	ldscripts	linker scripts for Bluetooth low energy projects and non-Bluetooth projects
	memory	Contains code for access QSPI flash when running in auto mode.
	misc	Contains ROM symbol table.
	osal	OS abstraction layer implementation, customized queue and resource management implementation.
	peripherals	Low level drivers for all supported peripherals.
	startup	Contains code for the initialization of the ARM processor.
	system	Contains power/clock manager code and header files.
	arm_license	License agreement

Table 61: interfaces directory structure

Directory	Subdirectory	Description
interfaces	*audio*	Contains the PDM audio interface driver.
	ble	Contains the implementation of the BLE framework.
	ble_services	Contains sample implementations of the BLE services.
	ble_stack	Contains a lower level BLE stack implementation library.
	ble_clients	Contains sample BLE client implementations.
	crypto	Empty.
	ftdf	FTDF PHY test
	usb	Contains the USB low level driver.

Table 62: middleware directory structure

Directory	Subdirectory	Description
middleware	*audio*	Empty.
	ble_net	Empty.
	cli	CLI service for the Dialog SmartSnippets DA1468x SDK.
	console	Serial console service for the Dialog SmartSnippets DA1468x SDK.
	dgtl	DGTL framework
	fw_upgrade	Empty.
	ip_net	Empty.
	logging	Implementation of a thread safe, UART based logging module.
	mcif	Monitor and Control I/F API.
	Monitoring	FreeRTOS task monitoring tools
	rf_tools	Contains project for the RF_tools.
	segger_tools	Modules used to support SEGGER's RTT and SystemView
	security	Empty.

A.5.4 Utilities directory

The Utilities directory contains a collection of useful tools in source format – mostly to interact with the Pro and Basic DK boards. Table 63 below shows the content of this directory.

Table 63: Utilities directory structure

Directory	Subdirectory	Description
Utilities	*bin2image*	Utility for creating a bootable image from an executable raw binary.
	cli_programmer	Command line interface programmer source code. For details please refer to Appendix A.
	mkimage	Utility for creating a firmware image for SmartSnippets DA1468x SDK.
	Scripts	Scripts for Windows and Linux for QSPI flash programming, Debugging with GDB, OTP patching, etc.
	nvparam	Creates an image of NV Parameters which can be then written

Directory	Subdirectory	Description
		directly to proper partition on flash.

Appendix B Command Line Interface (CLI) Programmer

B.1 CLI Programmer – Overview

`cli_programmer` is a command line tool for reading & writing to FLASH/OTP/RAM. It also provides some extra functions like loading & executing an image from RAM. The tool communicates with the target device over uart port or swd/jtag interface. It executes on Windows and Linux platforms.

Note 40 Writing an image to flash requires adding a header to the image. This process is handled by the `bin2image` tool, or the `cli_programmer write_qspi_exec` command.

B.2 Application command description

Open a terminal and navigate to the folder `<sdk_root_directory>/binaries/` To run `cli_programmer` the interface (GDB server or serial port) and the requested command must be supplied.

```
> cli_programmer [<options>] <interface> <command> [<args>]
```

For the interface name the user must use the name presented by the operating system. For the serial port the file name is e.g. `COM5` (Windows) or `/dev/ttyUSB0` (Linux) and for the SWD interface (J-Link debugger with the GDB server) is '`gdbserver`'.

Table 64: Commands and arguments

Option	Description
`write_qspi <address> <file> [<size>]`	Writes up to `` `size` `` bytes of `` `file` `` into the FLASH at `` `address` ``. If `` `size` `` is omitted, a complete file is written.
`write_qspi_bytes <address> <data1> [<data2> [...]]`	Writes bytes specified on command line into the FLASH at `` `address` ``.
`write_qspi_exec <image_file>`	Writes binary file (.bin) to flash at address 0, after adding header for execution in place (cached mode).
`write_suota_image <image_file> <version>`	Writes SUOTA enabled `` `image_file` `` to executable partition. The user supplied `` `version` `` string goes to image header.
`read_qspi <address> <file> <size>`	Reads `` `size` `` bytes from the FLASH memory, starting at `` `address` `` into `` `file` ``. If `` `file` `` is specified as either `'-'` or `'--'`, data is output to stdout as hexdump. The hexdump is either 16-bytes (-) or 32-bytes (--) wide.
`erase_qspi <address> <size>`	Erases `` `size` `` bytes of the FLASH, starting at `` `address` ``. Note: an actual erased area may be different due to the size of an erase block.
`chip_erase_qspi`	Erases the whole FLASH.
`copy_qspi <address_ram> <address_qspi> <size>`	Copies `` `size` `` bytes from the RAM memory, starting at `` `address_ram` `` to FLASH at `` `address_flash` ``. This is an advanced command and is not needed by end user.
`is_empty_qspi [start_address size]`	Checks that FLASH contains only `0xFF` values. If no arguments are specified starting address is `0` and size is

Option	Description
	1M. Command prints whether flash is empty and if offset of first non-empty byte.
read_partition_table	Reads the partition table (if any exists) and prints its contents.
write <address> <file> [<size>]	Writes up to `size` bytes of `file` into the RAM memory at `address`. If `size` is omitted, a complete `file` is written.
read <address> <file> <size>	Reads `size` bytes from the RAM memory, starting at `address` into `file`. If `file` is specified as either '-' or '--', data is output to stdout as hexdump. The hexdump is either 16-bytes (-) or 32-bytes (--) wide.
write_otp <address> <length> [<data> [<data> [...]]]	Writes `length` words to the OTP at `address`. `data` are 32-bit words to be written, if less than `length` words are specified, remaining words are assumed to be 0x00.
read_otp <address> <length>	Reads `length` 32-bit words from the OTP address `address`.
write_otp_file <file>	Writes data to the OTP as defined in `file` (default specified values are written).
read_otp_file <file>	Reads data from the OTP as defined in `file` (cells with default value provided are read) contents of each cell is printed to stdout.
write_tcs <length> [<reg_addr> <reg_data> [<reg_addr> <reg_data> [...]]]	Writes `length` 64-bit words to the OTP TCS section at first available (filled with 0) section of size `length`. `reg_addr`: the register address. It will be written as a 64-bit word [`reg_addr`,`~reg_addr`]. `reg_data`: the register data. It will be written as a 64-bit word [`reg_data`, `~reg_data`].
Boot	Boots the 2nd stage bootloader or the application binary (defined with -b) and exits.
read_chip_info	Reads chip information from chip revision registers and OTP header.

Table 65: General options

Option	Description
-h	Prints help screen and exits.
--save-ini	Saves CLI programmer configuration to the `cli_programmer.ini` file and exits.
-b file	Filename of 2nd stage bootloader or an application binary.

Table 66: GDB server specific options

Option	Description
-p <port_num>	TCP port number that GDB server listens to. The Default value is 2331.
-r <host>	GNU server host. The default is `localhost`.

Option	Description
--no-kill	Don't stop already running GDB server instance.
--gdb-cmd <cmd>	GDB server command used for executing and passing the right parameters to GDB server.

Without this parameter no GDB server instance will be started or stopped. As GDB server command line can be quite long, to avoid typing it every time the tool is executed and the best way to store this command line in cli_programmer.ini file is by using the '--save-ini' command line option.

Table 67: Serial port specific options

Option	Description
-s <baudrate>	Baud rate used for UART by uartboot. The parameter is patched to the uploaded uartboot binary (in that way passed as a parameter). This can be 9600, 19200, 57600 (default), 115200, 230400, 500000, 1000000.
-i <baudrate>	Initial baud rate used for uploading the `uartboot` or a user supplied binary. This depends on the rate used by the bootloader of the device. The default behavior is to use the value passed by '-s' or its default, if the parameter is not given. The argument is ignored by the `boot` command. '-s' option should be used in this case.
--tx-port <port_num>	GPIO port used for UART Tx by the `uartboot`. This parameter is patched to the uploaded uartboot binary (in that way passed as a parameter). The default value is 1. This argument is ignored when the `boot` command is given.
--tx-pin <pin_num>	GPIO pin used for UART Tx by uartboot. This parameter is patched to the uploaded uartboot binary (in that way passed as a parameter). The default value is 3. The argument is ignored when the `boot` command is givern.
--rx-port <port_num>	GPIO port used for UART Rx by uartboot. This parameter is patched to the uploaded uartboot binary (in that way passed as a parameter). The default value is 2. The argument is ignored when the `boot` command is given.
--rx-pin <pin_num>	GPIO pin used for UART Rx by uartboot. This parameter is patched to the uploaded uartboot binary (in that way passed as a parameter). The default value is 3. The argument is ignored when the `boot` command is given.
-w timeout	Serial port communication timeout is used only during download of uartboot binary, if during this time board will not respond cli_programmer exits with timeout error.

Table 68: bin2image options

Option	Description
`--prod-id DA14681-01`	DA14680-01, DA14681-01. Selects the chip product ID. This option applies only when `write_qspi_exec cmd` is used. It instructs `cli_programmer` to set the flash header which corresponds to the selected chip revision.

When `cli_programmer` is executed it tries to read the `cli_programmer.ini` file which may contain various `cli_programmer` options. Instead of creating this file manually, the user should use the '`--save-ini`' command line option. The format of the `cli_programmer.ini` adheres to standard Windows ini file syntax. The `cli_programmer` looks for ini file in the following locations:

- current directory
- home directory
- `cli_programmer` executable directory

B.3 Command examples

Example 1

Upload binary data to FLASH.

Windows:

```
>    cli_programmer COM40 write_qspi 0x0 data_i
```

Linux:

```
>   cli_programmer /dev/ttyUSB0 write_qspi 0x0 data_i
```

Example 2

Upload binary data to FLASH using maximum serial port baudrate.

```
>    cli_programmer -s 1000000 -i 57600 COM40 write_qspi 0x0 data_i
```

Example 3

Read data from FLASH to local file.

```
>    cli_programmer COM40 read_qspi 0x0 data_o 0x100
```

Example 4

Upload custom binary `test_api.bin` to RAM and execute it, using UART Tx/Rx P1_0/P1_5 (uses boot rom booter baud rate at 57600)

```
>    cli_programmer -b test_api.bin COM40 boot
```

Example 5

Upload custom binary `test_api.bin` to RAM and execute it, using UART Tx/Rx P1_3/P2_3 (uses boot rom booter baud rate ata 9600)


```
>    cli_programmer –s 9600 -b test_api.bin COM40 boot
```

Example 6

Modify FLASH at specified location with arguments passed in command line.

```
>    cli_programmer COM40 write_qspi_bytes 0x80000 0x11 0x22 0x33
```

Example 7

Run a few commands with uartboot, using UART Tx/Rx P1_0/P1_5 at baud rate 115200

(initial rate for uartboot uploading must be 57600).

```
>    cli_programmer -i 57600 -s 115200 COM40 write_qspi 0x0 data_i


>    cli_programmer -i 57600 -s 115200 COM40 read_qspi 0x0 data_o 0x100
```

Example 8

Run a few commands with uartboot, using UART Tx/Rx P1_3/P2_3 at baud rate 115200

(initial rate for uartboot uploading is 9600).

```
>    cli_programmer -i 9600 -s 115200 --tx-port 1 --tx-pin 3 --rx-port 2 --rx-pin 3
COM40 write_qspi 0x0 data_i


>    cli_programmer -i 9600 -s 115200 --tx-port 1 --tx-pin 3 --rx-port 2 --rx-pin 3
COM40 read_qspi 0x0 data_o 0x100
```

Example 9

Read FLASH contents (10 bytes at address `0x0`).

Start gdbserver manually!

```
>    cli_programmer gdbserver read_qspi 0 -- 10
```

Example 10

Write register `0x50003000` with value `0x00FF` and register `0x50003002` with value `0x00AA`.

```
>    cli_programmer gdbserver write_tcs 4 0x50003000 0x00FF 0x50003002 0x00AA
```

Example 11

Write settings to the `cli_programmer.ini` file. Long bootloader path is passed with -b option and command line to start GDB server is passed with '--gdb-cmd'. In this example GDB server command line contains arguments and path to executable has space so whole command line is put in quotes and quotes required by Windows path are additionally escaped.

```
>    cli_programmer –b
c:\users\jon\<sdk_root_directory>/bsp\system\loaders\uartboot\Release\uartboot.bin --
save-ini --gdb-cmd "\"C:\Program Files\SEGGER\JLink_V510d\JLinkGDBServerCL.exe\" -if
SWD -device Cortex-M0 -singlerun -silent -speed auto"
```


Example 12

Program a DA14681-01 chip with an executable flash image.

```
>       cli_programmer --prod-id DA14681-01 gdbserver write_qspi_exec
../../../../projects/dk_apps/features/tickless/DA14681-01-Debug_QSPI/tickless.bin
```

Example 13

Write 6 bytes specified in command line to flash at address `0x80000`.

```
>     cli_programmer gdbserver write_qspi_bytes 0x80000 0x11 0x22 0x33 0x44 0x55 0x66
```

Example 14

Write SUOTA enable application to proper location in flash.

```
>    cli_programmer gdbserver write_suota_image pxp_reporter.bin "1.1.0.1 a"
```

Example 15

Write OTP address 0x07f80128 with the following contents: `B0:0x00, B1:0x01, B2:0x02, B3:0x03, B4:0x04, B5:0x05, B6:0x06, B7:0x07`

```
>     cli_programmer gdbserver write_otp 0x07f80128 2 0x03020100 0x07060504
```

Read OTP address 0x07f80128.

```
>     cli_programmer gdbserver read_otp 0x07f80128 2
```

If written with the contents from above write example, it should return:

```
   0025    00 01 02 03 04 05 06 07
```

B.3.1 Installation and debugging procedure

The `cli_programmer` make use of the `libprogrammer` library which implements the underlying functionality on the host side. The `cli_programmer` can be linked either statically or dynamically with `libprogrammer`.

The `cli_programmer` uses `uartboot` application which acts as a secondary bootloader which `cli_programmer` downloads to the target for performing the read/write operations.

The project is found in `<sdk_root_directory>/utilities/cli_programmer/cli`.

Table 69: Build configurations

Configuration	Description
Debug	Debug version for Linux.
Debug_static	Debug version linked with a static version of the `libprogrammer` and it's recommended for Linux. It also builds `uartboot` project and includes it in `cli_programmer` executable.
Debug_static_win32	Debug version for Windows linked with a static version of `libprogrammer`.
Release	Release version for Linux.
Release_static	Release version linked with a static version of `libprogrammer` and it's recommended for Linux. It also builds uartboot project and includes it in `cli_programmer` executable.

Configuration	Description
Release_static_win32	Release version for Windows linked with a static version of `libprogrammer`.

B.3.2 Build instructions

Build instructions:

- Import `libprogrammer`, `cli_programmer` and `uartboot` into SmartSnippetsTM Studio
- Build `libprogrammer` , `cli_programmer` and `uartboot` in `Release_static` configuration (recommended)
- Run `cli_programmer` with proper parameters as described in Appendix B.2 and B.3.

Note 41 A prebuilt version of `cli_programmer` can be found under SmartSnippets™ DA1468x SDK's `binaries` folder.

Appendix C QSPI programming guide

C.1 General

This guide describes the methods and tools used for:

- Programming QSPI flash
- Debugging programs which execute from QSPI flash

After programming the QSPI, the image will execute by resetting or power cycling the board.

C.2 Prerequisites

Compile the following tools (sources and projects available on SmartSnippets™ DA1468x SDK):

cli_programmer:

Compile `<sdk_root_directory>/utilities/cli_programmer` project.

Linux: Use `Debug_static` build configuration in SmartSnippets™ Studio

Windows: Compile `cli_programmer.sln` using Visual studio OR use the binaries from SmartSnippets™ DA1468x SDK's `binaries` folder

(`cli_programmer.exe, libprogrammer.dll`)

bin2image:

Compile `<sdk_root_directory>/utilities/bin2image` project.

Linux: Run `make` from project's folder

Windows: Check `README.win32` in project's folder OR use the binary from SmartSnippets™ DA1468x SDK's `binaries` folder (`bin2image.exe`)

uartboot.bin:

This is the intermediate bootloader that `cli_programmer` uses for communicating with the target.

The `uartboot` firmware will automatically detect the device variant. This means that the provided `uartboot` binary file does not have to be rebuilt if a different device variant of the DA1468x family is used.

Compile `./sdk/bsp/system/loaders/uartboot/` in SmartSnippets™ Studio using the release configuration

Note 42 `cli_programmer` uses `uartboot.bin` for communicating with the target.

Copying `uartboot.bin` to the folder of the `cli_programmer` binary will allow `cli_programmer` to automatically detect & use `uartboot.bin`.

Alternatively, the path to `uartboot.bin` can be provided to `cli_programmer` using `-b commandline` option.

Import `scripts` project into the current SmartSnippets™ Studio workspace.

This is needed to starts the `cli_programmer` from within SmartSnippets™ Studio IDE

C.3 Compiling for execution from flash

SmartSnippets™ DA1468x SDK projects come with SmartSnippets™ Studio build configurations which compile for execution for FLASH (cached) or RAM.

Configuring a project for execution from FLASH/RAM breaks down to the following steps:

1. Configure the memory mapping in linker script.
2. Using SmartSnippets™ Studio:
   a. Edit `ldscripts/mem.ld.h` in project's folder (instructions can be found in file's header comments).
   b. During project build the `mem.ld` file will be automatically updated.
3. Not using SmartSnippets™ Studio:
   a. Edit project's `ldscripts/mem.ld` file.
4. Configure project to compile for execution from FLASH.
5. In `<sdk_root_folder>/sdk/bsp/custom_config_xxx.h` header file, set the macros described below as follows and compile normally:
   a. For execution from FLASH (cached):

```
#define dg_configEXEC_MODE                  MODE_IS_CACHED
#define dg_configCODE_LOCATION              NON_VOLATILE_IS_FLASH
```

Code 39: Execution from Flash (cached)

   b. For execution from FLASH (mirrored):

```
#define dg_configEXEC_MODE                  MODE_IS_MIRRORED
#define dg_configCODE_LOCATION              NON_VOLATILE_IS_FLASH
```

Code 40: Execution from Flash (mirrored)

   c. For execution from RAM:

```
#define dg_configCODE_LOCATION              NON_VOLATILE_IS_NONE
```

Code 41: Execution from RAM

Note 43 The BINARY output (`<project_name>.bin`, NOT the `<project_name>.elf`) will be used in the next steps.

C.4 Flashing an QSPI image

Using SmartSnippets™ Studio:

Select the folder which includes `<project_name>.bin` (e.g. the project's Debug folder) and execute one of the following scripts from SmartSnippets™ Studio external tools menu button.

Table 70: QSPI programming scripts on Windows Host

Script name	Notes
Program_qspi_serial_win (Note 1)	Use this script in case you want to program the selected binary to external QSPI memory using a serial interface. Please follow instructions given on the SmartSnippets Studio console window.
Program_qspi_jtag_win (Note 1)	Use this script in case you want to program the selected binary to external QSPI memory using the JTAG interface.

Note 44 When calling one of these scripts for the first time you will be prompted to enter configuration options. You may change the selected configuration any time using the `program_qspi_config_win` script.

Table 71: QSPI programming scripts on Linux Host

Script name	Notes
Program_qspi_serial_linux (Note 1)	Use this script in case you want to program the selected binary to external QSPI memory using a serial interface. Please follow instructions given on the SmartSnippets Studio console window.
Program_qspi_jtag_linux (Note 1)	Use this script in case you want to program the selected binary to external QSPI memory using the JTAG interface.

Note 45 When calling one of these scripts for the first time you will be prompted to enter configuration options. You may change the selected configuration any time using the `program_qspi_config_linux` script.

C.5 Debugging from QSPI

C.5.1 General

The user may use J-Link SWD interface to attach to the running target & debug, OR reset board & debug as follows:

Using SmartSnippets™ Studio :

The SmartSnippets™ DA1468x SDK includes SmartSnippets™ Studio launch configurations provided with the SmartSnippets™ DA1468x SDK projects (`ATTACH` for attaching to running target and `*_qspi.launch` for resetting & attaching to QSPI, RAM for debugging from RAM (where applicable)).

Outside SmartSnippets™ Studio :

The SmartSnippets™ DA1468x SDK includes sample scripts & gdb commands in `utilities/scripts/qspi folder`:

- `boot_qspi_dbg*` scripts reset board and put a breakpoint in `main()`. The scripts invoke J-Link gdb server and issue gdb commands
- `gdb_cmd_qspi_*` files are the gdb command files used by the above scripts

Instructions on using these scripts are provided in the next paragraph.

C.5.2 Debugging with gdb scripts

1. The scripts include references to the executables:
   - `JLinkGDBServerCL`
   - `arm-none-eabi-gdb`

   and can be found in the following SmartSnippets™ DA1468x SDK path:

   `<sdk_root_directory>/utilities/scripts/qspi`

2. Make sure that you have the paths to these executables included in your platform's system path. If not, edit script files and add the absolute paths to these executables, for instance in `boot_qspi_dbg.bat`: replace `JLinkGDBServerCL.exe` with: `C:/Program Files/SEGGER/JLink_V512h/JLinkGDBServerCL.exe`
3. Edit the Debug scripts (`boot_qspi_dbg.*`) and replace `PUT_YOUR_APP_ELF_HERE.elf` with the name of the `.elf` file you want to debug. This should be the `.elf` file of the binary image flashed in the QSPI.
4. Execute the `boot_qspi_dbg.*` script.

5. The J-Link gdb server running must be operational, connected to target, downloading & running the boot loader and CPU halting at `main()` breakpoint. After this point a "continue" command can be issued and debug process may proceed, using the same gdb server instance.
6. Alternatively, it is possible to invoke a second instance of J-Link gdb server and attach it to the target.

Note 46 Since hardware breakpoints are used, only 4 breakpoints are available.

Appendix D SEGGER SystemView integration instructions

SEGGER SystemView is a real-time recording and visualization tool that reveals the true runtime behavior of an application. To enable SystemView, follow the instructions below.

Configuring SmartSnippets™ Studio projects to support SystemView:

To enable SystemView for a specific build configuration of any SmartSnippets™ Studio project, configure the project to build SystemView's source files and include header file directories:

1. Select a project which has the "OS_FREERTOS" definition enabled. Bare metal projects are currently not supported.
2. Right click project's "sdk" subfolder and select **New > Folder**

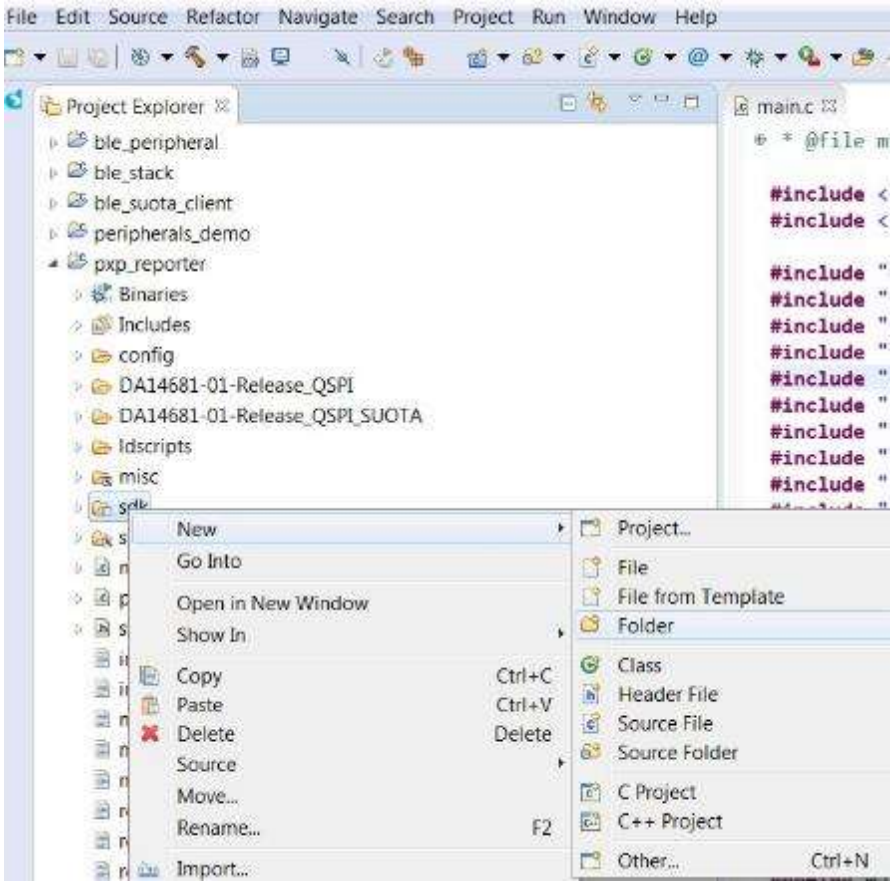

Figure 92: Create a new folder

3. In the pop-up window select: **Advanced > "Link to alternate location" > Browse**... and select the `<sdk_root_directory>\sdk\middleware\segger_tools` folder as shown in Figure 93.

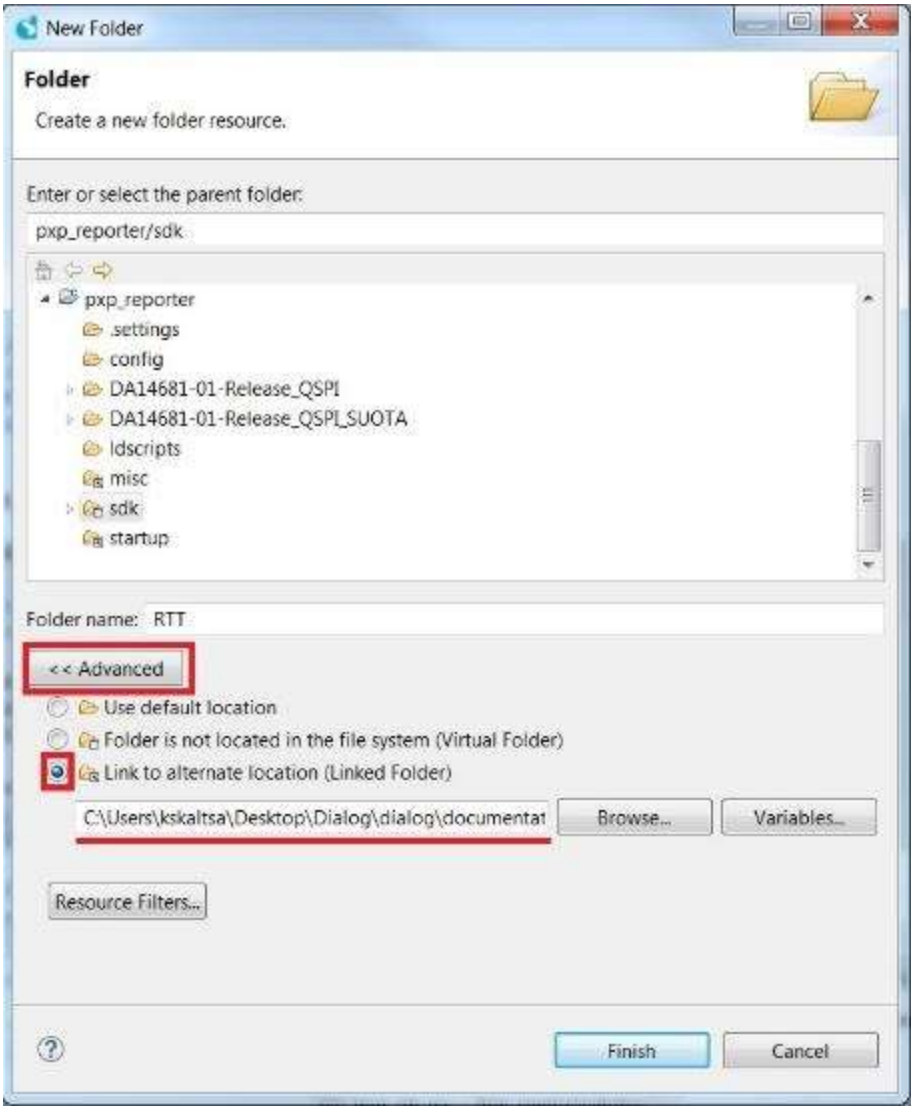

Figure 93: Select the Linker Folder

4. Right click project's name and go to **Properties > C/C++ Build > Settings > Tool Settings > Cross ARM C Compiler > Includes > Include Paths** (see Figure 94), add the following Workspace folders and click apply:

```
${workspace_loc:/${ProjName}/sdk/segger_tools/Config}
${workspace_loc:/${ProjName}/sdk/segger_tools/OS}
${workspace_loc:/${ProjName}/sdk/segger_tools/SEGGER}
```

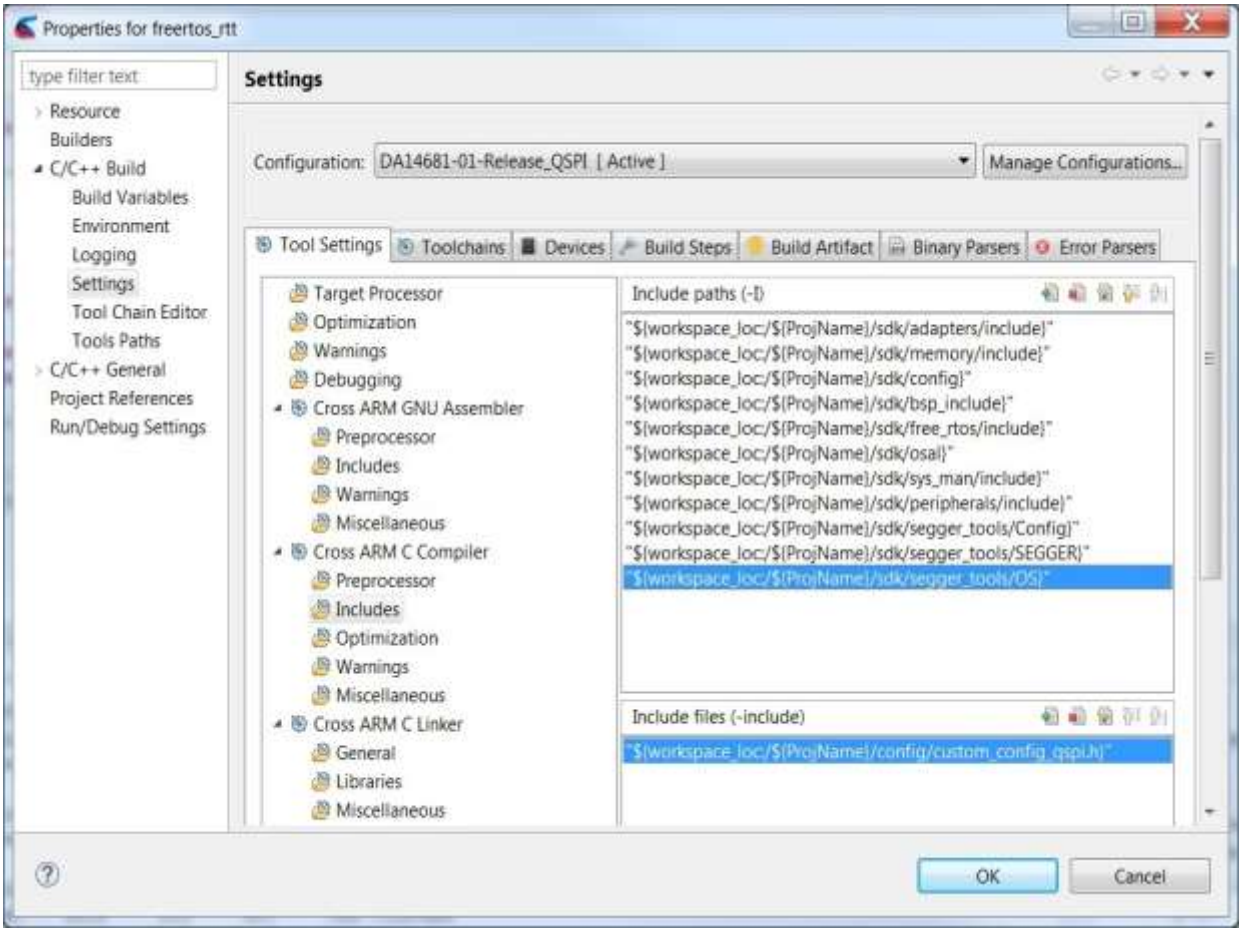

Figure 94: Include folder paths

5. Open the project's `config/custom_config_*.h` file and add Code 42 to add and enable the System View configuration:

```
#define dg_configSYSTEMVIEW                    (1)
```

Code 42: Enable System View configuration

- **Note** that `configTOTAL_HEAP_SIZE` should be increased by `dg_configSYSTEMVIEW_STACK_OVERHEAD` bytes for each system task. For example, if there are 8 system tasks, `configTOTAL_HEAP_SIZE` should be increased by:
- (8 * `dg_configSYSTEMVIEW_STACK_OVERHEAD`) bytes.

6. To call `SEGGER_SYSVIEW_Conf()` from application add Code 43. A good place to do this is inside `system_init()` after the configuration of system clocks:

```
#if dg_configSYSTEMVIEW
SEGGER_SYSVIEW_Conf();
#endif
```

Code 43: Call System View

7. Build and download the application image to DA1468x. Then run the application either by pressing the Reset Button (Release Build) or by start debugging the application.
8. To disable SystemView, set `dg_configSYSTEMVIEW` to 0 and rebuild the application.

Running SystemView on the Host:

1. Start the Segger System View application from SmartSnippets™ Studio.

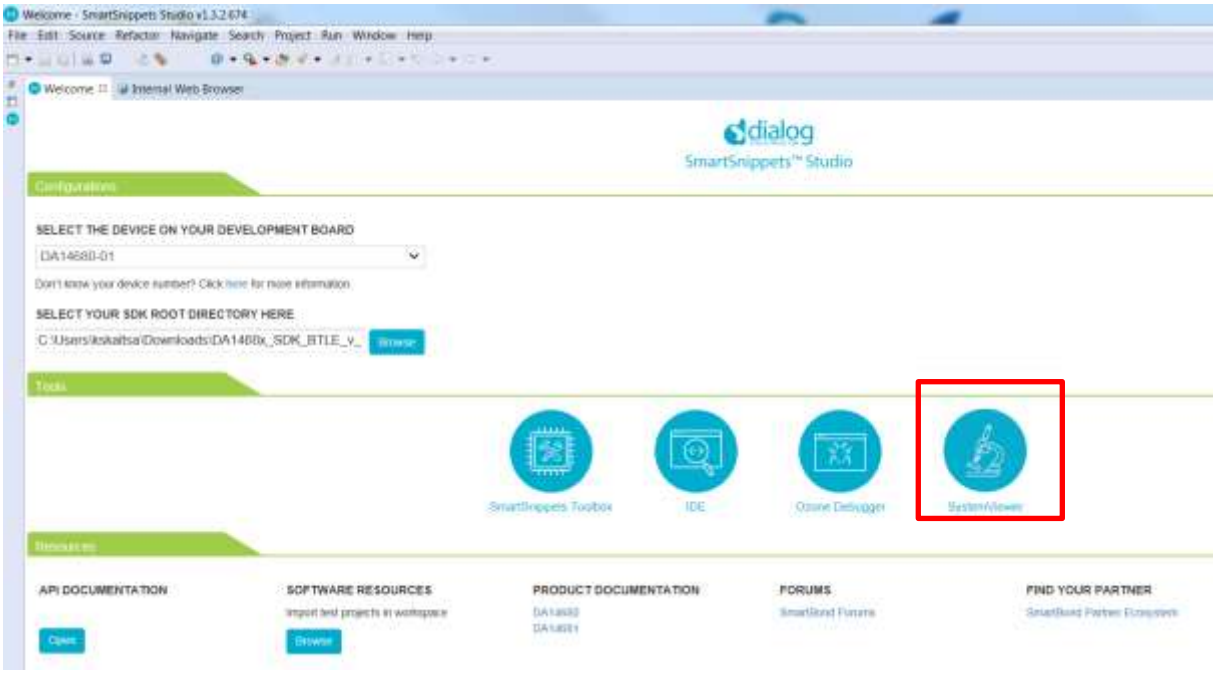

Figure 95: System Viewer application

2. Configure the SEGGER System View as shown in Figure 96.

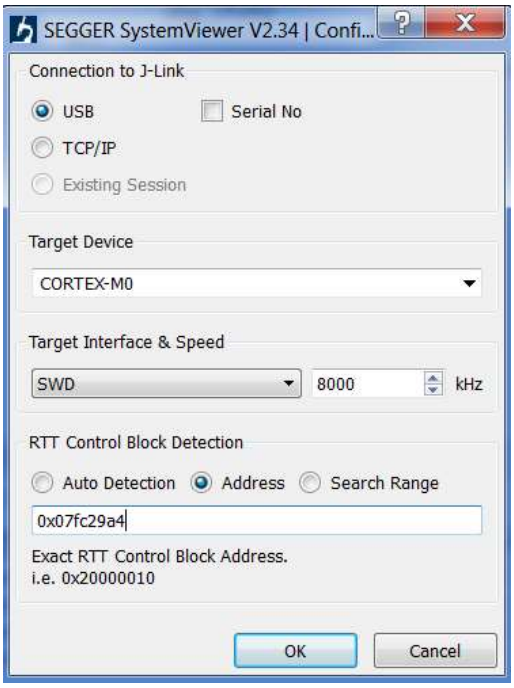

Figure 96: Configuring the SEGGER System Viewer

Note 47 The address for the RTT Control Block Detection is located in application's .map file. For example, for PXP reporter the .map file is located at `pxp_reporter\DA14681-01-Release_QSPI`. Press Ctrl+F and search the address of the `_SEGGER_RTT` variable.

3. On the main SystemView window, press the upper right "Start Recording Button" button.

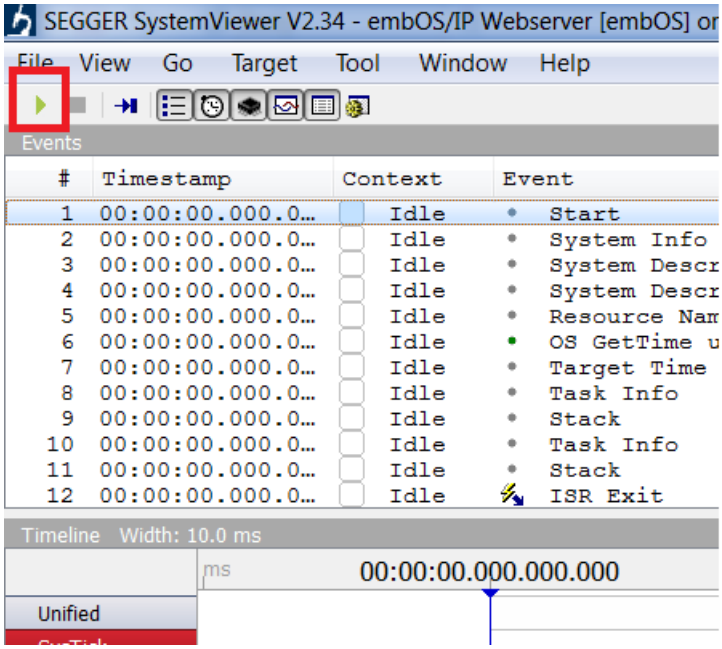

Figure 97: Start Recording

4. One can Stop and Restart the recording at any time using the buttons from SystemView PC Application.

Additional information:

1. The processing overhead of SystemView is not negligible and can potentially affect system dynamics or cause assertions due to the delays inserted from ISR monitoring. To minimize the impact on time critical ISRs there are some configuration options that allow the user to enable/disable the monitoring of certain aspects of the system, as shown below:


```
/*
 * Enable/Disable SystemView monitoring for BLE related ISRs (BLE_GEN_Handler /
BLE_WAKEUP_LP_Handler).
 * */
#ifndef dg_configSYSTEMVIEW_MONITOR_BLE_ISR
#define dg_configSYSTEMVIEW_MONITOR_BLE_ISR     (1)
#endif

/*
 * Enable/Disable SystemView monitoring for CPM related ISRs (SWTIM1_Handler /
WKUP_GPIO_Handler).
 * */
#ifndef dg_configSYSTEMVIEW_MONITOR_CPM_ISR
#define dg_configSYSTEMVIEW_MONITOR_CPM_ISR     (1)
#endif

/*
 * Enable/Disable SystemView monitoring for USB related ISRs (USB_Handler /
VBUS_Handler).
 * */
#ifndef dg_configSYSTEMVIEW_MONITOR_USB_ISR
#define dg_configSYSTEMVIEW_MONITOR_USB_ISR     (1)
#endif
```

Code 44: Enable/disable the monitoring

2. In applications with heavy IRQ usage it is possible that the currently used 2kb RTT buffer cannot hold all the monitored events and this may cause RTT overflows which simple means that some events are lost. The number of lost events is visible in the System View GUI RTT overflows property.

   To avoid this SEGGER suggest:

- Minimize the interactions of the debugger with J-Link while the target is running. (i.e. disable live watches)
- Select a higher interface speed in all instances connected to J-Link. (i.e. The debugger and System Viewer)
- Choose a larger buffer for System View. (1 - 4 kByte)
- Run System Viewer stand-alone without a debugger.

3. System View uses sdk's RTC, which is clocked by one of the LP clocks. For example, if the LP clock is XTAL32K each timer tic corresponds to ~31us. Events that last less than 30us will be visualized as events that last 31us.
4. Application/Device name can be set in `SEGGER_SYSVIEW_Config_FreeRTOS.c` `SYSVIEW_APP_NAME/SYSVIEW_DEVICE_NAME` definitions. At the same file function `_cbSendSystemDesc()` is located which sends comma separated IRQ names to the Host PC. Because the internal used buffer is only 128 bytes, not all IRQs are named there. One could use multiple `SEGGER_SYSVIEW_SendSysDesc()` calls to name all IRQs but this means more processing requirements. It is recommended to leave just one packet there.
5. Memory overhead is 2Kb RAM for RTT buffers, 256 bytes heap for every thread, 256 bytes stack for the shared IRQ stack and ~5Kb rom.
6. It is possible to redirect printf messages on SystemView's host window:
   - if `CONFIG_RETARGET` is defined, messages go to UART
   - if `CONFIG_RTT` is defined, messages go to RTT
   - if `CONFIG_NO_PRINT` is defined, messages are discarded
   - If nothing of the above is defined, but the `dg_configSYSTEMVIEW` is enabled, messages go to System View's GUI terminal.

Currently System View supports only integer variables, so %s strings are not printed properly. `printf` using SystemView inserts delays (because of the temporary disabled IRQs) that may trigger assertions so it is not recommended to be used.

Appendix E System Clocks

Table 72: System Clocks

Clocks	Description
XTAL16M	Crystal oscillator for the system clock (16MHz or 32MHz). Crystal Constraint should be up to +/-40ppm.
XTAL32K	Crystal oscillator for the low power clock. Crystal Constraint should be up to +/-500ppm.
RCX	XTAL_LP replacement (10.5 KHz).
RC16	RC oscillator (~15.5MHz) for the initial CPU clocking until the XTAL is settled.
RC32K	RC oscillator (32KHz) for clocking the HW-FSM at power up.
PLL96	A PLL which will increase the system clock to 96MHz.

Appendix F Batteries

Table 73: Battery types

Battery types	Description
BATTERY_TYPE_LICOO2	Lithium cobalt oxide (LiCoO2).
BATTERY_TYPE_LIMN2O4,	Lithium ion manganese oxide battery.
BATTERY_TYPE_LIFEPO4	Lithium iron phosphate.
BATTERY_TYPE_LINICOAIO2	Lithium Nickel Cobalt Aluminum Oxide.

Appendix G Power

Table 74: Power Definitions

Power	Definition
LDO_ret	This low drop out (LDO) provides power to internal registers that need to retain their content when the system is sleeping.
LDO_IO_RET/ LDO_IO_RET2	These LDOs provide power to input/output blocks that need to retain their configuration.
DCDC	Direct Current – to – Direct Current.
LDO_VBAT_RET	This LDO makes sure that only the current needed for retention is drown form the battery.
CC	Constant Current.
CV	Constant Voltage.
VBAT	Battery supply voltage.

Appendix H Trim and Calibration

The following table presents a list of the registers which are included in the TCS. Not all of them are required but no other than those in the Table 75 can be included.

Table 75: Trim and Calibration Section expected values per chip version

TRIM & CALIBRATION SECTION				Chip Revision
#	Register	What	Who	DA14680/1-01
1	CLK_32K_REG	Program RC32K_TRIM	Dialog at Production Testing	Used
2	CLK_RCX20K_REG	Program RCX20K_TRIM	Dialog at Production Testing	Used
3	CLK_16M_REG	RC16M_TRIM	Dialog at Production Testing	Used
4	CLK_FREQ_TRIM_REG	Crystal Dependent	Customer at Product Line Testing	Used
5	XTALRDY_CTRL_REG	Crystal Dependent	Customer at Product Line Testing	Used
6	BANDGAP_REG		Dialog at Production Testing	Used
7	CHARGER_CTRL2_REG		Dialog at Production Testing	Used
8	RF_BIAS_CTRL1_BLE_REG		Dialog at Production Testing	Used
9	RF_LNA_CTRL1_REG		Dialog at Production Testing	Used
10	RF_LNA_CTRL2_REG		Dialog at Production Testing	Used
11	RF_VCOCAL_CTRL_REG		Dialog at Production Testing	Used

TRIM & CALIBRATION SECTION				Chip Revision
12	RF_MIXER_CTRL1_BLE_REG		Dialog at Production Testing	Used
13	RF_VCO_CTRL_REG		Dialog at Production Testing	Used
14	RF_SPARE1_BLE_REG		Dialog at Production Testing	Used
15	RF_DIV_IQ_RX_REG		Dialog at Production Testing	Not Used
16	RF_DIV_IQ_TX_REG		Dialog at Production Testing	Not Used
17	BOD_CTRL2_REG	Which rail to BOD protect	Customer at Product Line Testing	Used
18	RF_BIAS_CTRL1_FTDF_REG		Dialog at Production Testing	Used
19	RF_MIXER_CTRL1_FTDF_REG		Dialog at Production Testing	Used
20	LED_CONTROL_REG		Dialog at Production Testing	Not Used
21	Free			
22	Free			
23	Free			
24	Free			

Note 48 TCS value of XTALRDY_CTRL_REG is in clock cycles for 32000/32768. In the case of RCX, it will not be applied. Instead, the hard-coded value of the SmartSnippets™ DA1468x SDK will be applied.

Appendix I Configuration parameters

Table 76 is a list of configuration parameters. For more up-to-date information please refer to Doxygen files.

Table 76: List of configuration parameters

Macro	Documented In	Description
dg_configUSE_LP_CLK		Low Power clock used (LP_CLK_32000, LP_CLK_32768, LP_CLK_RCX, LP_CLK_ANY).
dg_configEXEC_MODE	C.3	Configuration of a project for cached or mirrored execution from FLASH.
dg_configCODE_LOCATION	C.3	Configuration of a project for execution from RAM.
dg_configEXT_CRYSTAL_FREQ		Frequency of the crystal connected to the XTAL Oscillator: 16MHz or 32MHz.
dg_configIMAGE_FLASH_OFFSET		Offset of the image if not placed at the beginning of QSPI Flash.
dg_configUSER_CAN_USE_TIMER1		Timer 1 usage. When set to 0, Timer1 is reserved for the OS tick.
dg_configMEM_RETENTION_MODE		Retention memory configuration. 5 bits field; each bit controls whether the relevant memory block will be retained (1) or not (0).
dg_configSHUFFLING_MODE		Memory Shuffling mode. See SYS_CTRL_REG:REMAP_RAMS field.
dg_configUSE_WDOG		Watchdog Service. 1: enabled 0: disabled
dg_configFLASH_POWER_DOWN	Table 29	Puts QSPI Flash to “Power Down” for the duration of the sleep period.
dg_configPOWER_1V8_ACTIVE		The rail from which the Flash is powered, if a Flash is used. FLASH_IS_NOT_CONNECTED FLASH_CONNECTED_TO_1V8 FLASH_CONNECTED_TO_1V8P When set to 1, the 1V8 rail is powered, when the system is in active state.
dg_configPOWER_1V8_SLEEP		When set to 1, the 1V8 is powered during sleep.

Macro	Documented In	Description
dg_configPOWER_1V8P		When set to 1, the 1V8P rail is powered.
dg_configBATTERY_TYPE	Table 31	Defines the battery type that is used in the system.
dg_configBATTERY_CHARGE_VOLTAGE	Table 31	Defines the charging voltage setting for the charger hardware.
dg_configBATTERY_TYPE_CUSTOM_ADC_VOLTAGE	Table 31	In case of a custom battery, this parameter must be defined to provide the charging voltage level of the battery (in ADC measurement units).
dg_configBATTERY_LOW_LEVEL	Table 31	If not zero, this is the lowest allowed limit of the battery voltage.
dg_configPRECHARGING_THRESHOLD	Table 31	The threshold below which pre-charging starts
dg_configCHARGING_THRESHOLD	Table 31	The threshold that, when met, pre-charging stops and charging starts.
dg_configBATTERY_CHARGE_CURRENT	Table 31	This is the charging current setting for the charger hardware.
dg_configBATTERY_PRECHARGE_CURRENT	Table 31	This is the pre-charging current setting for the charger hardware.
dg_configBATTERY_CHARGE_NTC	Table 31	It controls whether the thermal protection will be enabled or not.
dg_configPRECHARGING_TIMEOUT	Table 31	The maximum time that pre-charging will last.
dg_configUSE_SOC		When set to 1, State of Charge function is enabled.
dg_configUSE_USB		When set to 1, the USB interface is used
dg_configUSE_USB_CHARGER	Table 31	It enables / disables the use of the Charger from the application. • 0 : disables the charger (must be used when no battery is attached) • 1 : enables the charger
dg_configUSE_USB_ENUMERATION	Table 31	It controls whether enumeration with the USB
dg_configALLOW_CHARGING_NOT_ENUM	Table 31	It controls whether the Charger will start charging using charge current up to 100mA until the enumeration completes.

Macro	Documented In	Description
dg_configUSE_NOT_ENUM_CHARGING_TIMEOUT	Table 31	According to the USB Specification, there is a time limit that a device, which is connected to the USB bus but not enumerated, can draw power. This configuration setting controls whether the Charger will respect this time limit or not.
dg_configPRECHARGING_INITIAL_MEASURE_DELAY	Table 31	This is the time to wait before doing the first voltage measurement after starting pre-charging.
dg_configCHARGING_CC_TIMEOUT	Table 31	The maximum time that the charging hardware will stay in the CC phase.
dg_configCHARGING_CV_TIMEOUT	Table 31	The maximum time that the charging hardware will stay in the CV phase.
dg_configUSB_CHARGER_POLLING_INTERVAL	Table 31	While being attached to a USB cable and the battery has been charged, this is the interval that the VBAT is polled to decide whether a new charge cycle will be started.
dg_configBATTERY_CHARGE_GAP	Table 31	This is the safety limit used to check for battery overcharging.
dg_configBATTERY_REPLENISH_GAP	Table 31	This is the threshold below the maximum voltage level of the battery where charging will be restarted in order to recharge the battery.
dg_configUSE_ProDK		When set to 1, the ProDK is used (controls specific settings for this board).
dg_configUSE_SW_CURSOR		Use SW cursor.
dg_configCACHEABLE_QSPI_AREA_LEN		Set the size (in bytes) of the QSPI flash cacheable area. All reads from offset 0 up to (not including) offset dg_configCACHEABLE_QSPI_AREA_LEN will be cached. In addition, any writes to this area will trigger cache flushing, to avoid any cache incoherence. The size must be 64KB-aligned, due to the granularity of CACHE_CTRL2_REG[CACHE_LEN]. • 0 : Turn off cache. • -1 : Don't configure cacheable area size (i.e. leave as set by booter).
dg_configFLASH_ADAPTER		When enabled the FLASH adapter is included in the compilation of the SDK.

Macro	Documented In	Description
		• 0 : Disabled • 1 : Enabled
dg_configNVMS_ADAPTER		When enabled the NVMS (Non Volatile Memory Storage) adapter is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configNVMS_VES	12.4.2	Must be defined in order to use VES (Virtual EEPROM).
dg_configNVPARAM_ADAPTER		When enabled the NVPARAM (Analog to Digital Converter) adapter is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configGPADC_ADAPTER		When enabled the GPADC (Non Volatile Parameters) adapter is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configBLE_PERIPHERAL	Table 50	Set to 0 if the application is not using BLE-peripheral related code.
dg_configBLE_CENTRAL	Table 50	Set to 0 if the application is not using BLE- central related code.
dg_configBLE_OBSERVER	Table 50	Set to 0 if the application is not using BLE-observer related code.
dg_configBLE_BROADCASTER	Table 50	Set to 0 if the application is not using BLE-broadcaster related code.
dg_configBLE_GATT_CLIENT	Table 50	Set to 0 if the application is not using GATT client related code.
dg_configBLE_GATT_SERVER	Table 50	Set to 0 if the application is not using GATT client related code.
dg_configBLE_L2CAP_COC	Table 50	Set to 0 if the application is not using L2CAP connection oriented channels related code.
dg_configBLE_EVENT_COUNTER_ENABLE		Enable Event Counters in BLE ISR. If the application has not defined dg_configBLE_EVENT_COUNTER_ENABLE in its custom_config file, this is defined to the default value of 0 to

Macro	Documented In	Description
		disable the Event Counters in BLE stack ISR.
dg_configBLE_ADV_STOP_DELAY_ENABLE		Enable ADV_UNDERRUN workaround. If the application has not defined dg_configBLE_ADV_STOP_DELAY_ENABLE in its custom_config file, this is defined to the default value of 0 to disable the ADV_UNDERRUN workaround in the BLE adapter.
dg_configBLE_SKIP_LATENCY_API		Enable Secure Connections. If the application has not defined dg_configBLE_SECURE_CONNECTIONS in its custom configuration file, this is defined by default to 1 to enable LE Secure Connections.
dg_configBLE_CONN_EVENT_LENGTH_MIN		Minimum Connection Event Length. Minimum length for Connection Event in steps of 0.625ms. This is used in outgoing connection requests, connection parameter requests and connection updates.
dg_configBLE_CONN_EVENT_LENGTH_MAX		Maximum Connection Event Length. Maximum length for Connection Event in steps of 0.625ms. This is used in outgoing connection requests, connection parameter requests and connection updates.
dg_configBLE_DATA_LENGTH_RX_MAX		Maximum Receive Data Channel PDU Payload Length. If the application has not defined dg_configBLE_DATA_LENGTH_RX_MAX in its custom_config file, this is defined to the maximum value allowed by Bluetooth Core v_4.2, which is 251 octets.
dg_configBLE_DATA_LENGTH_TX_MAX		Maximum Transmit Data Channel PDU Payload Length. If the application has not defined dg_configBLE_DATA_LENGTH_TX_MAX in its custom_config file, this is defined to the maximum value allowed by Bluetooth Core v_4.2, which is 251 octets.
dg_configBLE_DUPLICATE_FILTER_MAX		Duplicate Filtering List Maximum size. This defines the size of the list used for duplicate filtering. When the duplicate filtering list is full, additional advertising reports or scan responses will be dropped.

Macro	Documented In	Description
dg_configBLE_PAIR_INIT_KEY_DIST	Table 17	Security keys to be distributed by the pairing initiator. This defines which security keys will be requested to be distributed by the pairing initiator during a pairing feature exchange procedure.
dg_configBLE_PAIR_RESP_KEY_DIST	Table 17	Security keys to be distributed by the pairing responder. This defines which security keys will be requested to be distributed by the pairing responder during a pairing feature exchange procedure.
dg_configBLE_SECURE_CONNECTIONS	Table 17	Enable Secure Connections. If the application has not defined dg_configBLE_SECURE_CONNECTIONS in its custom configuration file, this is defined by default to 1 to enable LE Secure Connections.
dg_configTRACK_OS_HEAP	13.2.3	Activation of OS Heap tracking.
dg_configUSE_DGTL		Enable DGTL interface. When this macro is enabled, the DGTL framework is available for use. The framework must furthermore be initialized in the application using dgtl_init(). Additionally, the UART adapter must be initialized accordingly. Please see sdk/middleware/dgtl/include/ for further DGTL configuration (in dgtl_config.h) and API.
dg_configI2C_ADAPTER		When enabled the I2C (Inter-Integrated Circuit) adapter is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configUSE_HW_I2C		When enabled the Inter-Integrated Circuit low level driver is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configSPI_ADAPTER		When enabled the SPI (Serial Peripheral Interface) adapter is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled

Macro	Documented In	Description
dg_configUSE_HW_SPI		When enabled the Serial Peripheral Interface low level driver is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configUART_ADAPTER	12.3.1	Must be defined and set to 1, in a project, in order to enable the UART adapter
dg_configUSE_HW_UART	12.3.1	Must be defined and set to 1, in a project, in order to enable the UART adapter
dg_configTESTMODE_MEASURE_SLEEP_CURRENT		When set to 1, the system will go to sleep and never exit allowing for the sleep current to be measured.
dg_configTEMPSENS_ADAPTER		When enabled the TEMPSENS (Temperature Sensor) adapter is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configUSE_HW_IRGEN		When enabled the Infra-Red Generator low level driver is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configUSE_HW_QUAD		When enabled the Quadrature decoder low level driver is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configRF_ADAPTER		When enabled the Radio adapter is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configUSE_HW_RF		When enabled the Radio module low level driver is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configUSE_HW_TIMER0		When enabled the Timer 0 low level driver is included in the compilation of

Macro	Documented In	Description
		the SDK. • 0 : Disabled • 1 : Enabled
dg_configUSE_HW_TIMER1		When enabled the Timer 1 low level driver is included in the compilation of the SDK. • 0 : Disabled • 1 : Enabled
dg_configUSE_HW_TIMER2		When enabled the Timer 2 low level driver is included in the compilation of the SDK. 0 : Disabled 1 : Enabled
dg_configPM_MAX_ADAPTERS_CNT		Maximum adapters count. Should be equal to the number of Adapters used by the Application. It can be larger (up to 254) than needed, at the expense of increased Retention Memory requirements. It cannot be 0.
dg_configUSE_CLI		Enable Command Line Interface module.
dg_configUSE_CONSOLE		Enable serial console module.
dg_configUSE_CLI_STUBS		Enable Command Line Interface stubbed API.
dg_configUSE_CONSOLE_STUBS		Enable serial console stubbed API.
dg_configUSE_BOD		brief Brown-out Detection • 1: used • 0: not used
dg_configUSE_DCDC		When set to 1, the DCDC is used.
dg_configDISABLE_BACKGROUND_FLASH_OPS		Disable background operations. When enabled, outstanding QSPI operations will take place during sleep time increasing the efficiency. • 1 : Disabled • 0 : Enabled
dg_configCRYPTO_ADAPTER		The adapter for the cryptographic engines (AES/HASH and ECC).
dg_configUSE_HW_WKUP		When enabled the Wakeup Timer low level driver is included in the

Macro	Documented In	Description
		compilation of the SDK. • 0 : Disabled • 1 : Enabled

Revision history

Revision	Date	Description
1.0	19-Nov-2015	First released version
2.0	22-Apr-2016	Update for SmartSnippets DA1468x SDK Release 1.0.4.812
2.1	17-Jun-2016	Update for SmartSnippets DA1468x SDK Engineering Release 1.0.5.885
3.0	26-Jul-2016	Update for SmartSnippets DA1468x SDK Release 1.0.6.968
4.0	07-Dec-2016	Update for SmartSnippets DA1468x SDK Release 1.0.8
5.0	21-Jul-2017	Update for SmartSnippets DA1468x SDK Release 1.0.10
5.0.1	9-Nov-2017	Update for SmartSnippets DA1468x SDK Release 1.0.10
5.0.2	27-Nov-2017	Update for SmartSnippets DA1468x SDK Release 1.0.10
6.0	08-Dec-2017	Update for SmartSnippets DA1468x SDK Release 1.0.12
6.1	19-Jan-2022	Updated logo, disclaimer, copyright.

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.

RoHS Compliance

Dialog Semiconductor's suppliers certify that its products are in compliance with the requirements of Directive 2011/65/EU of the European Parliament on the restriction of the use of certain hazardous substances in electrical and electronic equipment. RoHS certificates from our suppliers are available on request.