User Manual

DA1458x Software Platform Reference

UM-B-051

Abstract

This document describes the software platform for the SmartBond™ DA1458x Product Family, specifically for the DA14580/581/583 devices, as supported by the new v5.x SDK series. It presents the overall system architecture, components, Application Programming Interfaces (APIs) as well as the development tool chain, environment and process.

Contents

Figures

Tables

1 Terms and Definitions

ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
BLE	Bluetooth low energy
BR	Basic Rate
DA1458x	DA1458x SoC Platform of Product Family of devices, for this document specifically referring to the DA14580/581/583 devices
CPU	Central Processing Unit
EDR	Enhanced Data Rate
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GFSK	Gaussian Frequency-Shift Keying
GPIO	General Purpose Input/Output
GTL	Generic Transport Layer
HCI	Host Controller Interface
HW	Hardware
I2C	Inter Integrated Circuit (interface)
ISM	Industrial, Scientific, and Medical (frequency band)
L2CAP	Logical Link Control and Adaptation Protocol
LDO	Low Drop-Out (regulator)
LL	Link Layer
MAC	Media Access Control (network address)
NMI	Non-Maskable Interrupt
NVDS	Non-Volatile Data Storage
OTP	One Time Programmable (memory)
PCB	Printed Circuit Board
PHY	PHYsical layer
RAM	Random Access Memory
ROM	Read-Only Memory
SDIO	Secure Digital Input/Output
SDK	Software Development Kit
SIG	(Bluetooth) Special Interest Group
SoC	System on Chip
SPI	Serial Peripheral Interface
SPOTA	Software Patching Over the Air
SPOTAR	Software Patching Over the Air Receiver
SRAM	Static Random Access Memory
SUOTA	Software Upgrade Over the Air
SW	Software
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
WFI	Waiting For Interrupt
WLAN	Wireless Local Area Network

2 References

[1] DA14580 Datasheet, Dialog Semiconductor.

[2] DA14581 Datasheet, Dialog Semiconductor.

[3] DA14583 Datasheet, Dialog Semiconductor.

[4] RW-BLE Host Interface Specification (RW-BLE-HOST-IS), Riviera Waves.

[5] RW-BLE Host Software (RW-BLE-HOST-SW-FS), Riviera Waves.

[6] RealView Development Suite Getting Started Guide, ARM Ltd.

[7] UM-B-048, Getting Started with DA1458x Development Kits - Basic, User manual, Dialog Semiconductor.

[8] UM-B-049, Getting Started with DA1458x Development Kits - Pro, User manual, Dialog Semiconductor.

[9] Bluetooth Specification Version 4.1, Bluetooth SIG.

[10] Riviera Waves Kernel (RW-BT-KERNEL-SW-FS), Riviera Waves.

[11] GAP Interface Specification (RW-BLE-GAP-IS), Riviera Waves.

[12] GATT Interface Specification (RW-BLE-GATT-IS), Riviera Waves.

[13] ATTDB Interface Specification (RW-BLE-ATTDB-IS), Riviera Waves.

[14] Proximity Profile Interface Specification (RW-BLE-PRF-PXP-IS), Riviera Waves.

[15] UM-B-050, DA1458x Software Developer's Guide, User manual, Dialog Semiconductor.

[16] UM-B-008, DA14580_581_583 Production test tool, User manual, Dialog Semiconductor.

[17] UM-B-013, DA14580/581 External processor interface over SPI, User manual, Dialog Semiconductor.

[18] UM-B-007, DA14580 Software Patching over the Air (SPOTA), User manual, Dialog Semiconductor.

[19] UM-B-011, DA14580/581 Memory file and scatter file, User manual, Dialog Semiconductor.

[20] UM-B-012, DA14580/581 Creation of a secondary boot loader, User manual, Dialog Semiconductor.

[21] UM-B-008, DA14580/581 Sleep mode configuration, User manual, Dialog Semiconductor.

3 Introduction

The aim of this document is to serve as a reference to the embedded software developer by providing at start a practical, high-level comprehension for the Bluetooth low energy standard, then an overview for the system architecture for the DA1458x System on Chip (SoC) family of integrated circuit (IC) devices, consisting of the DA14580/581/583 and finally both an outline of the DA1458x software architecture as well as, an as much as possible complete view of its components and supporting aspects, including development tool chain and environment.

3.1 Target Audience

This is a document for embedded software developers, also called embedded firmware engineers that are working on developing applications that use any of the SmartBond™ DA1458x Product Family of devices which are based on the DA1458x System on Chip (SOC) platform.

Developers that are new to the DA1458x System on Chip (SoC) platform are advised to scan through the whole of this document in order to get familiar with what is covered herein as well as where specific information is located. Then the developer can spend some time reading through the initial sections of this reference.

Experienced embedded firmware engineers after going through the contents and some key chapters, will be prepared to dive deeper into the SDK and its provided example applications. Then they are advised to look into detailed technical documentation, in order to have a clearer idea of how applications can be developed and executed on Dialog's DA1458x Bluetooth® low energy devices and how to best utilize the capabilities offered by Dialog's DA1458x SoC platform.

3.2 How to Use This Document

The emphasis for this document is in being a reference, i.e. that the developer does not need to read through the whole of this document; the key for the reader is to become familiar with the concepts described herein, so that during development to be able to use the Software Developer's Guide, to get the required results.

Embedded software developers that are new to BLE and/or to Dialog's DA1458x System on Chip (SoC) platform are advised to review the contents and then read through from section 3.1 to section 4.8, then section 11 to familiarize with the supported development environment. After that they are advised to read and use document [15]. In case that someone needs to get a better understanding and wants to delve deeper to a specific subject, he/she can then come back to this document at the specific chapter that covers this subject in this reference. For deeper analysis, this reference document points to even more in-depth technical description in an Appendix or other documents that covers in much more detail the specific subject.

This reference document does not intent to provide a thorough understanding of Bluetooth low energy, neither it covers the internals of how its data is organized, or how BLE devices communicate with each other and the key design decisions and trade-offs that one may need to take when designing BLE supported designs and applications.

It intends however to provide to the software developer enough understanding to Dialog's DA1458x platform high-level APIs approach for both BLE and its peripherals as well as the confidence on how these enable developing faster and better BLE applications when using the DA1458x SoCs.

Bluetooth low energy technology (BLE) 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, SIG focused towards developing a low complexity radio standard with the least 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 that the following sections are based on the book "Getting Started with Bluetooth Low Energy" by Kevin Townsend, Carles Cufí, Akiba, Robert Davidson.

3.3 Device Modes

Devices that support both BLE and BR/EDR are referred as dual-mode devices. 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 BLE are referred to as single-mode devices.

3.3.1 Single Mode Devices

A Single-mode (BLE or Bluetooth Low Energy) device, is only implementing BLE and is able to communicate with both single-mode and dual-mode devices, however not with devices only supporting BR/EDR. BLE support is a must-have for single-mode devices to handle incoming messages and issue a response.

3.3.2 Dual Mode Devices

A Dual-mode BR/EDR/LE, Bluetooth Low Energy Ready device, implements both BR/EDR and BLE and is able to communicate with any Bluetooth device.

3.4 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 up-to-date Bluetooth devices include an increased number of building blocks. 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.

3.5 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).

3.5.1 Integrated Processor

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

3.5.2 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 BLE connectivity with specialized microcontrollers to be integrated without modifying the overall design.

Figure 1 shows a comparison between the two approaches when Bluetooth is implemented:

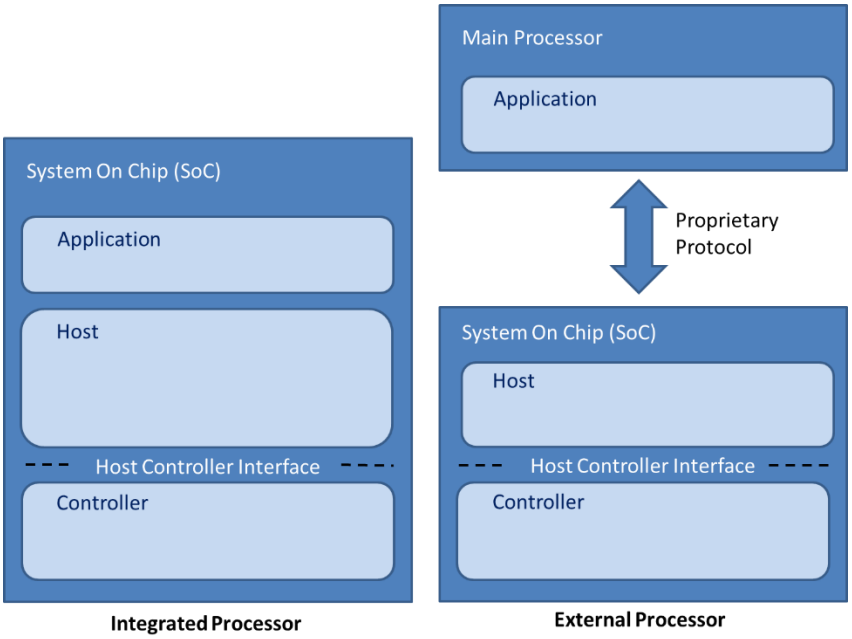


Figure 1: Integrated vs. External Processor BLE Hardware Configurations

3.6 Network Modes

BLE 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 3.7.

3.6.1 Broadcasting

When using connectionless broadcasting, a BLE device sends data out to any scanning device or receiver that is within acceptable listening range. Essentially, this mechanism allows a BLE device to send data out one-way to anyone or anything that is capable of picking 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. These broadcasted data are sent out by using the advertising features of BLE.

3.6.2 Connecting

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

- Central (master): Repeatedly scans the pre-set BLE 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 through the use of additional protocol layers and, more specifically, the Generic Attribute Profile (GATT). Data are organized around units called services and characteristics. Moreover, connections allow for higher throughput and have the ability to establish a secure encrypted link, as well as negotiation of connection parameters to fit the data model.

A BLE device can have multiple services and characteristics, organized in a meaningful structure. Services can contain multiple characteristics, each with their own access rights and descriptive metadata.

3.7 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 conformant 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 a particular objective, whether generic or specific.

3.7.1 Generic Profiles

Generic profiles are defined by the Bluetooth specification and two of them are fundamental as they ensure the interoperability between BLE 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 BLE. This profile is mandatory for all BLE devices, and all must comply with it.
- Generic Attribute Profile (GATT): Addresses data exchanges in BLE 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 BLE.

GAP and GATT are so fundamental to BLE 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.

3.7.2 Use-Case-Specific Profiles

Use-case-specific profiles are usually limited to GATT-based profiles. Largely all of 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.

3.7.2.1 SIG-Defined GATT-Based Profiles

The Bluetooth SIG further to providing a solid reference framework for the control and data layers of devices involved in a BLE network, it also provides a predefined set of use-case profiles based on GATT, that completely cover all procedures and data formats required to implement a wide range of specific use cases, like the following:

- Find Me Profile: It allows devices to physically locate other devices (for example using a smartphone to find a BLE 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 BLE (for keyboards, mice, remote controls).
- Glucose Profile: It securely transfers glucose levels over BLE.
- Health Thermometer Profile: It transfers body temperature readings over BLE.

The Bluetooth SIG's Specification in its Adopted Documents page provides a full list of SIG-approved profiles (more information at 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.

3.7.2.2 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 a particular 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.

3.7.3 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 the initiation of security features.

Essentially, GAP can be considered as the BLE topmost control layer, given that it specifies how devices perform control procedures such as device discovery and secure connection establishment, in order to ensure interoperability thus allowing data exchange between devices from different vendors.

GAP specifies four roles that a device can adopt in a BLE 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 optionally additional data such as the device name.
- Observer: The scanning device, upon receiving the advertisement, 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 matching the peer device's address, the central device will then send out a request to establish a connection (link) with the advertising device having the particular 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.

3.7.4 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 full detail how profile and user data is to be exchanged over a BLE 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 of the GATT-based profiles as defined by SIG. Effectively by covering the precise use cases for the profiles, it ensures interoperability between devices from different vendors; all the standard BLE profiles are therefore based on GATT and must comply with it to operate correctly, which makes GATT a key section of the BLE 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 BLE devices:

- Client: It sends requests to a server, receives responses and potentially server initiated updates as well. The GATT client does not know anything in advance about the server's attributes, so it must first inquire the presence and nature of those attributes by performing service discovery. After completing service discovery, it starts reading and writing attributes found in the server, as well as receiving server-initiated updates. It corresponds to the ATT client.
- Server: It receives requests from a client and issues responses. It also sends server-initiated updates when configured to do so, and it is the role responsible for storing and making the user data available to the client, organized in attributes. Every BLE 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.

3.8 Protocol Stack

Similar to all Bluetooth devices from architectural point of view, a single-mode BLE device is divided into three blocks: controller, host, and application. Each of these basic building blocks is consists of several layers which make the device operational, tightly integrated in the so-called Protocol Stack, presented in Figure 2:

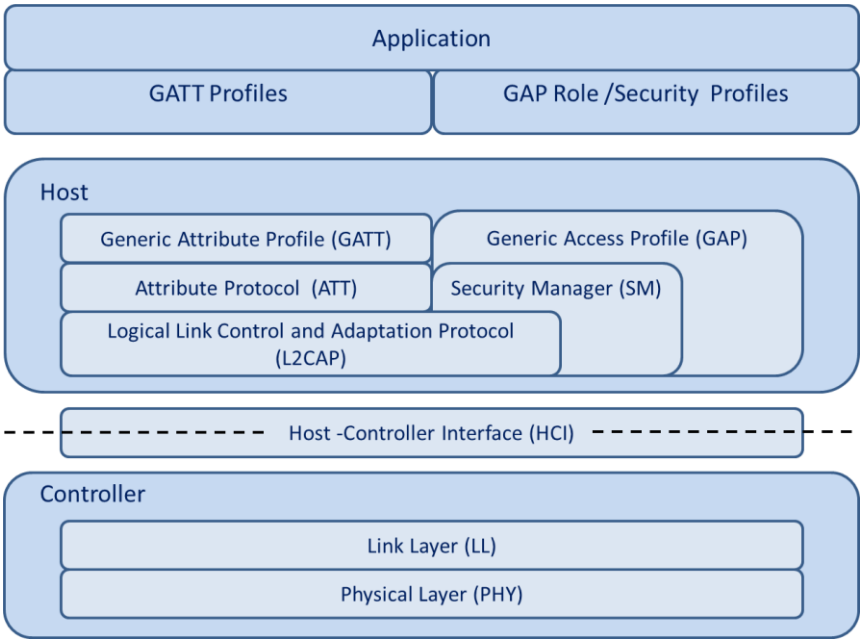


Figure 2: BLE Protocol Stack Layers

The following sections provide a description of the aforementioned building blocks along with the layers each one encompasses.

3.9 Controller

The Controller includes all the lower level functionality necessary for a BLE device to communicate; it is comprised by the Physical Layer (PHY), the Link Layer (LL) and the controller side of the Host Controller Interface (HCI).

3.9.1 Physical Layer (PHY)

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

3.9.2 Link Layer (LL)

The Link Layer (LL) directly interfaces with the PHY; it is the hard real-time constrained layer of the protocol stack as it has to comply with all the timing requirements defined in the specification. Given that many of the calculations performed by the LL are computationally expensive, automated functions are usually implemented in hardware to avoid overloading the Central Processing Unit (CPU) that runs all software layers in the stack, therefore the LL implementation comes through a combination of custom hardware and software .The functionality provided by LL usually includes Preamble, Access Address, and air protocol framing, CRC generation and verification, data whitening, random number generation and AES encryption and 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 BLE device can be a master, a slave, or both depending on the use case and the corresponding requirements. A master is able to 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, and 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 3:

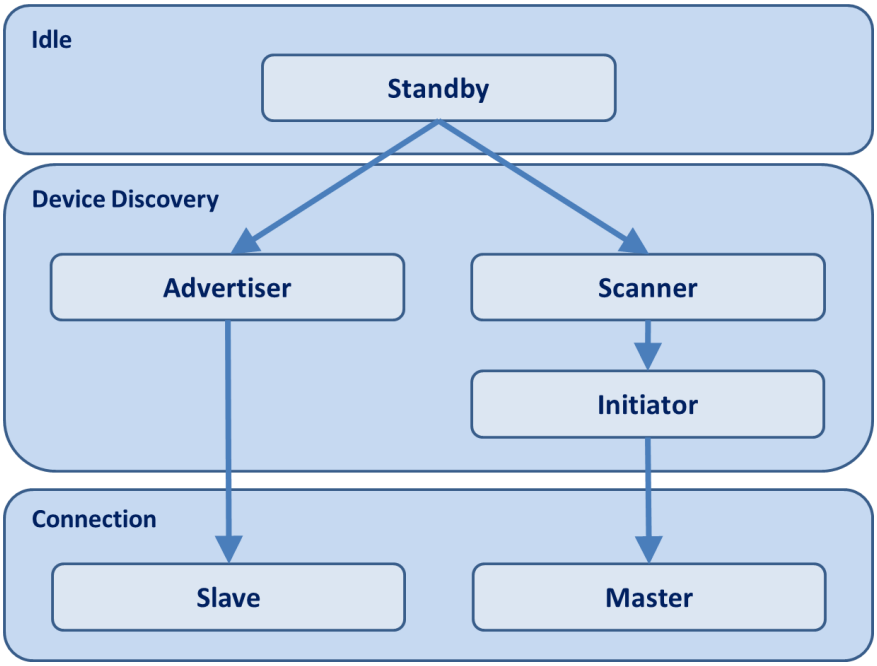


Figure 3: Link Layer States

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, 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).

3.9.2.1 Bluetooth Device Address

The Bluetooth device address is the primary identifier of a Bluetooth device, similar to what an Ethernet Media Access Control (MAC) address is for network devices. It is a 48-bit (6-byte) number that uniquely identifies a device among peers. There are two types of device addresses, and a device is possible 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 has to 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 preprogrammed or dynamically generated at runtime on the device. There are numerous use cases in which such addresses are useful in BLE.

3.9.2.2 Advertising and Scanning

The BLE specification allows only one packet format and two types of packets, advertising and data.

Advertising packets are used for two purposes, in particular to:

- broadcast data for applications that do not need the overhead of a full connection establishment
- discover slaves and connect with them

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 resent the packet until it is finally acknowledged by the receiver.

3.9.3 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.).

3.10 Host

The Host block consists of a set of layers, each with specific role and functionality, the cooperation of which makes the overall block operational. As shown in Figure 2 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).

3.10.1 Host Controller Interface – Host Side

The HCI interface at the Host side provides a mean of communication to the controller via a standardized interface. Similar to the Controller Side HCI, this layer can be implemented either through a software API, or over a hardware interface such as UART or SPI.

3.10.2 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 BLE 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 BLE packets. The reverse process takes place in the receiving end, where the fragmented large upper layer packets are reassembled by multiple small BLE packets and transmitted upstream 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.

3.10.3 Attribute Protocol

The ATT layer enables a BLE device to provide certain pieces of data, known as attributes, to another BLE device. In the context of ATT, the device exposing attributes is referred to as the server, and the peer device interested and 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 meaning that in case of a pending request, (i.e. no response yet received for an already issued request), no further requests can be submitted until the response 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 Universally Unique Identifier (UUID), a set of permissions, and finally a value. Effectively, the attribute handle is a mere 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 operation if that is not the case.

3.10.4 Security Manager

The Security Manager (SM) layer defines the means for pairing and key distribution and provides functions for the other layers of the protocol stack to securely connect and exchange data with another BLE device. It includes both a protocol and a series of security algorithms that are designed to provide the BLE protocol stack with the ability to generate and exchange security keys to allow 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: Always corresponds to the Link Layer master
- Responder: 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 in order for the device to be able to switch to 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, typically stored in non-volatile memory and therefore allowing 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, whereas bonding actually creates 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. Certain documentation and APIs sometimes use the term pairing with bonding instead of simply bonding, since a bonding procedure always includes a pairing phase before.

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

3.10.5 Application

The application, like in all other types of systems, is at the highest layer and the one responsible for containing the logic, user interface, and data handling of everything related to the actual use-case that the application implements. The architecture of an application is highly dependent on each particular implementation and in BLE it typically uses the capabilities provided by the BLE profiles.

3.11 DA1458x System on Chip Platform

3.11.1 Overview

Dialog's SmartBond™ DA1458x Product Family is based on the DA1458x System on Chip Platform. The DA1458x platform includes a number of blocks as shown in Figure 4.

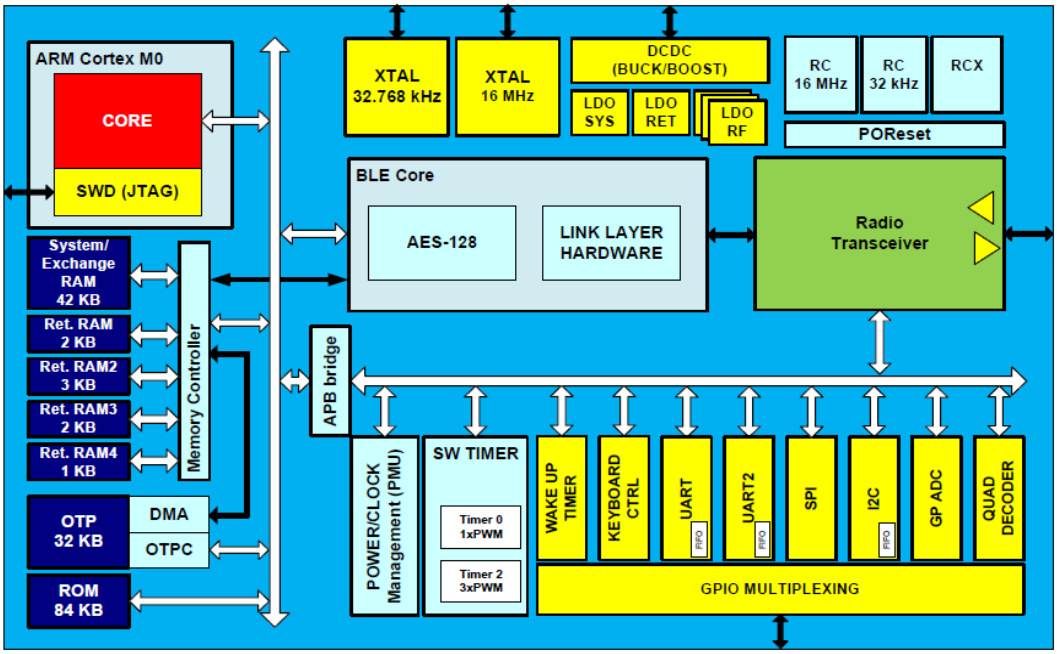


Figure 4: DA1458x System on Chip Platform Main Blocks

In the following sections a brief overview of the platform's main building blocks is provided; for detailed information please refer to the specific device datasheet.

3.11.2 ARM Cortex-M0 CPU

The ARM Cortex-M0 processor is a 32-bit Reduced Instruction Set Computing (RISC) processor with a von Neumann architecture (single bus interface). It uses an instruction set called Thumb, which was first introduced and supported in the ARM7TDMI processor; however, several newer instructions from the ARMv6 architecture as well as a few instructions from the Thumb-2 technology are also included. Thumb-2 technology extended the previous Thumb instruction set to allow all operations to be carried out in one CPU state. The instruction set in Thumb-2 includes both 16-bit and 32-bit instructions; most instructions generated by the C compiler use the 16-bit instructions, and the 32-bit instructions are used when the 16-bit version cannot carry out the required operations. This results in high code density and avoids the overhead of switching between two instruction sets. In total, the Cortex-M0 processor supports only 56 base instructions, although some instructions can have more than one form. Although the instruction set is small, the Cortex-M0 processor is highly capable because the Thumb instruction set is highly optimized. Academically, the Cortex-M0 processor is classified as load-store architecture, as it has separate instructions for reading and writing to memory, and instructions for arithmetic or logical operations that use registers.

3.11.2.1 Features

- Thumb instruction set. Highly efficient, high code density and able to execute all Thumb instructions from the ARM7TDMI processor.
- High performance. Up to 0.9 DMIPS/MHz (Dhrystone 2.1) with fast multiplier.
- Built-in Nested Vectored Interrupt Controller (NVIC). This makes interrupt configuration and coding of exception handlers easy. When an interrupt request is taken, the corresponding

interrupt handler is executed automatically without the need to determine the exception vector in software.

- Interrupts can have four different programmable priority levels. The NVIC automatically handles nested interrupts.
- The design is configured to respond to exceptions (e.g. interrupts) as soon as possible (minimum 16 clock cycles).
- Non-maskable interrupt (NMI) input for safety critical systems.
- Easy to use and C friendly. There are only two modes (Thread mode and Handler mode). The whole application, including exception handlers, can be written in C without any assembler.
- Built-in System Tick timer for OS support. A 24-bit timer with a dedicated exception type is included in the architecture, which the OS can use as a tick timer or as a general timer in other applications without an OS.
- SuperVisor Call (SVC) instruction with a dedicated SVC exception and PendSV (Pendable SuperVisor service) to support various operations in an embedded OS.
- Architecturally defined sleep modes and instructions to enter sleep. The sleep features allow power consumption to be reduced dramatically. Defining sleep modes as an architectural feature makes porting of software easier because sleep is entered by a specific instruction rather than implementation defined control registers.
- Fault handling exception to catch various sources of errors in the system.
- Support for 24 interrupts.
- Little endian memory support.
- Wake up Interrupt Controller (WIC) to allow the processor to be powered down during sleep, while still allowing interrupt sources to wake up the system.
- Halt mode debug. Allows the processor activity to stop completely so that register values can be accessed and modified. No overhead in code size and stack memory size.
- CoreSight technology. Allows memories and peripherals to be accessed from the debugger without halting the processor.
- Supports Serial Wire Debug (SWD) connections. The serial wire debug protocol can handle the same debug features as the JTAG, but it only requires two wires and is already supported by a number of debug solutions from various tools vendors.
- Four (4) hardware breakpoints and two (2) watch points.
- Breakpoint instruction support for an unlimited number of software breakpoints.
- Programmer’s model similar to the ARM7TDMI processor.
- Most existing Thumb code for the ARM7TDMI processor can be reused. This also makes it easy for ARM7TDMI users, as there is no need to learn a new instruction set.

3.11.3 Memory

The DA1458x SoC platform includes the following internal memory blocks. ROM, OTP, System SRAM, and Retention RAM.

3.11.3.1 ROM

This is an 84 kB ROM containing the Bluetooth Low Energy protocol stack as well as the boot code sequence.

3.11.3.2 OTP

This is a 32 kB One-Time Programmable memory array, used to store the application code as well as Bluetooth Low Energy profiles. It also contains the system configuration and calibration data.

3.11.3.3 System SRAM

This is a 42 kB system SRAM (Sys-RAM) which is primarily used for mirroring the program code from the OTP when the system wakes/powers up. It also serves as Data RAM for intermediate variables

and various data that the protocol requires. Optionally, it can be used also as extra memory space for the BLE TX and RX data structures.

3.11.3.4 Retention RAM

These are 4 special low leakage SRAM cells (2 kB + 2 kB + 3 kB + 1 kB) used to store various data of the Bluetooth Low Energy protocol as well as the system's global variables and processor stack when the system goes into Deep Sleep mode. Storage of this data ensures secure and quick configuration of the BLE Core after the system wakes up. Every cell can be powered on or off according to the application needs for retention area when in Deep Sleep mode

3.11.4 BLE Core and Radio Transceiver

The Radio Transceiver implements the RF part of the Bluetooth Low Energy protocol. Together with the Bluetooth 4.1 PHY layer, this provides a 93 dB RF link budget for reliable wireless communication. All RF blocks are supplied by on-chip low-drop out-regulators (LDOs). The bias scheme is programmable per block and optimized for minimum power consumption. The Bluetooth LE radio comprises the Receiver, Transmitter, Synthesizer, Rx/Tx combiner block, and Biasing LDOs.

3.11.4.1 Features

- Single ended RFIO interface, 50 Ω matched.
- Alignment free operation.
- -93 dBm receiver sensitivity.
- 0 dBm transmit output power.
- Ultra-low power consumption.
- Fast frequency tuning minimizes overhead.

3.11.5 Peripheral Interfaces

The platform supports a number of peripheral interfaces, namely UARTs, SPI+, I2C, ADC, Quadrature decoder and Keyboard controller over multiplexed GPIOs.

Drivers are provided for each interface to provide an easier-to-use experience towards the hardware blocks as the developer does not have to interact with the register programming directly.

On top of the these core interface drivers, a number of sample example drivers enabling communication with commonly used Bluetooth Low Energy application components like accelerometers, Flash/EEPROM non-volatile memories, etc. are also provided.

The following sections briefly describe these interfaces.

3.11.5.1 UARTs

The UARTs are compliant to the industry-standard 16550 and can be used for serial communication with a peripheral, modem (data carrier equipment, DCE) or data set. Data is written from a master (CPU) over the APB bus to the UART and it is converted to serial form and transmitted to the destination device. Serial data is also received by the UART and stored for the master (CPU) to read back. There is no DMA support on the UART block as it contains internal FIFOs. Both UARTs support hardware flow control signals (RTS, CTS, DTR, DSR).

3.11.5.2 SPI+

A subset of the Serial Peripheral Interface (SPI) is supported and as serial interface it can transmit and receive 8, 16 or 32 bits in master/slave mode and transmit 9 bits in master mode. The SPI+ interface has enhanced functionality with bidirectional 2x16-bit word FIFOs.

SPI™ is a trademark of Motorola, Inc.

3.11.5.3 I2C

The I2C interface is a programmable control bus that provides support for the communications link between Integrated circuits in a system. It is a simple two-wire bus with a software-defined protocol for system control, which is used in temperature sensors and voltage level translators to EEPROMs, general-purpose I/O, A/D and D/A converters.

3.11.5.4 ADC

The DA1458x platform includes a high-speed ultra-low power 10-bit general purpose Analog-to-Digital Converter (GPADC). It can operate in unipolar (single ended) mode as well as in bipolar (differential) mode. The ADC has its own voltage regulator (LDO) of 1.2 V, which represents the full scale reference voltage.

A conversion has two phases: the sampling phase and the conversion phase. When bit GP_ADC_CTRL_REG[GP_ADC_EN] is set to ‘1’, the ADC continuously tracks (samples) the selected input voltage. Writing a '1' at bit GP_ADC_CTRL_REG[GP_ADC_START] ends the sampling phase and triggers the conversion phase. When the conversion is ready the ADC resets bit GP_ADC_START and returns to the sampling phase. The conversion itself is fast and takes about one clock cycle of 16 MHz, though the data handling will require several additional clock cycles depending on the software code style. The fastest code can handle the data in four clock cycles of 16 MHz, resulting to a highest sampling rate of 16 MHz/5 = 3.3 Msample/s.

3.11.5.5 Quadrature Decoder

This block decodes the pulse trains from a rotary encoder to provide the step and the direction of the movement of an external device. Three axes (X, Y, Z) are supported. The integrated quadrature decoder can automatically decode the signals for the X, Y and Z axes of a HID input device, reporting step count and direction: the channels are expected to provide a pulse train with 90 degrees phase difference; depending on whether the reference channel is leading or lagging, the direction can be determined This block can be used for waking up the chip as soon as there is any kind of movement from the external device connected to it.

3.11.5.6 Keyboard Controller

The Keyboard controller can be used for debouncing the incoming GPIO signals when implementing a keyboard scanning engine. It generates an interrupt to the CPU (KEYBR_IRQ). In parallel, five extra interrupt lines can be triggered by a state change on a number of selectable GPIOs (GPIOx_IRQ) as determined by the device package.

3.11.6 Timers

The platform supports general purpose, wake up and a watchdog timer.

3.11.6.1 General Purpose Timers

The Timer block contains 2 timer modules that are software controlled, programmable and can be used for various tasks.

Timer 0 is a 16-bit general purpose timer with the ability to generate 2 Pulse Width Modulated signals (PWM0 and PWM1) that have common programming).

Timer 2 is a 14-bit general purpose timer, with the ability to generate 3 Pulse Width Modulated signals (PWM2, PWM3 and PWM4).

3.11.6.2 Wake-Up Timer

The Wake-up timer can be programmed to wake up the DA1458x device from power down mode after a preprogrammed number of GPIO events. It features monitoring for GPIO state change, while implementing debouncing time from 0 ms up to 63 ms. It accumulates external events and compares their number to a programmed value, and generates an interrupt to the CPU.

3.11.6.3 Watchdog Timer

The Watchdog timer is an 8-bit timer with sign bit that can be used to detect an unexpected execution sequence caused by a software run-away and can generate a full system reset or a Non-Maskable Interrupt (NMI).

3.11.7 Clock and Reset

The clock is supported by a Digital Controlled Xtal Oscillator (DXCO), a Pierce configured type of oscillator designed for low power consumption and high stability. There are two such crystal oscillators in the system, one at 16 MHz (XTAL16M) and a second at 32.768 kHz (XTAL32K).

Moreover, the DA1458x platform comprises an RST (reset) pad which is active high. It contains an RC filter for spikes suppression and the typical latency of the RST pad is in the range of 2 μs.

3.11.8 Power Management (PMU)

The DA1458x platform has a complete power management function integrated with a Buck DC-DC converter and separate LDOs for the different power domains of the system.

3.11.9 SmartBond™ DA1458x Product Family Devices

The SmartBond™ DA1458x Product Family devices that are based on the DA1458x System on Chip family, are the following: DA14580, DA14581, and DA14583.

3.11.9.1 DA14580

The world's smallest, lowest power and most integrated Bluetooth Low Energy solution. It gives the developer the freedom to develop efficient Bluetooth 4.1 applications with the longest battery lifetimes as it offers world leading power consumption figures drawing just 4.9 mA at transmission and reception, effectively supporting much longer battery lifetime. Moreover, it can run from voltages as low as 0.9 V, making it ideal for running from single-cell or Zinc-air batteries.

3.11.9.2 DA14581

The world's smallest, lowest power and most integrated Bluetooth Low Energy solution for A4WP wireless charging and HCI applications. Offering fast boot time for the Power Receiving Unit (PRU) and eight connections for the Power Transmitting Unit (PTU) make the DA14581 a perfect solution for A4WP applications. The optimized code for HCI, which fits into the DA14581's OTP memory, enables developers to create a preprogrammed HCI device.

3.11.9.3 DA14583

Adding Flash support the DA14583 device is the most flexible and lowest power Bluetooth Low Energy solution, as it combines the benefits of the lowest power, smallest size and lowest system cost Bluetooth Low Energy System-on-Chip with an integrated Flash memory. This offers you to the developer the flexibility of Software Upgrades Over The Air (SUOTA), enabling devices that are up-to-date in the field.

4 DA1458x Software Platform Overview

In this section an overview description of the overall platform including the software architecture that supports the DA1458x SoC Platform is provided. A diagram of the overall software architecture providing a visual overview of the various layers of software involved is given in Figure 5. The following paragraphs of this document provide a high level description of the functionality provided by each layer and the APIs that each layer exposes to the application programmer.

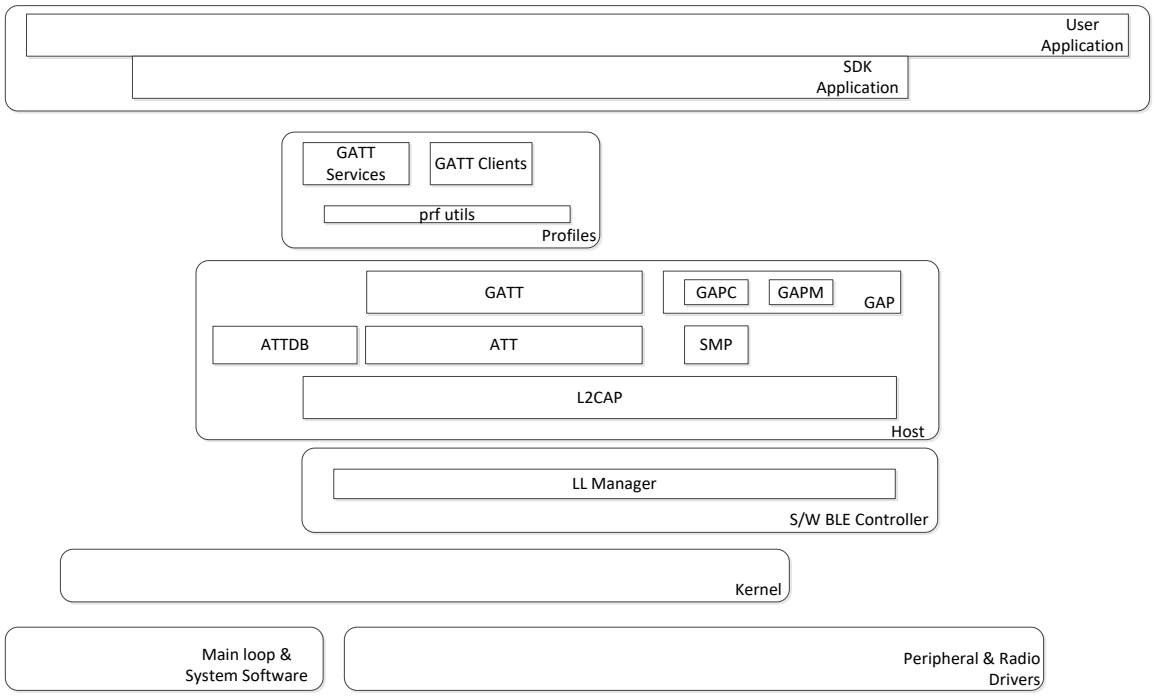


Figure 5: Software Architecture

4.1 System Software and Main Loop

At system start up the main function initializes the system and gets into the main loop. The key concept is simple. Check if the BLE is active and if it is provide CPU time to the kernel scheduler to process all pending messages and events. Inquire then the user application if it has non-message related tasks to perform. If both kernel and user application have nothing more to execute, transition quickly into low power mode and wait for an interrupt to start again. The main loop is not indented to be altered by the user application.

A set of callbacks are provided by which the application gets notified about the state of the main loop. We often use the term asynchronous execution for these callbacks to distinguish them from other events and callbacks that are generated from within the kernel scheduler, which we refer to as synchronous execution. The application can partially control the main loop with the return value of the callbacks or by setting the sleep settings accordingly. It is important to understand here, that the Kernel scheduler is called from within the main loop when the BLE is active. If we do not grant control back to the main loop or if the BLE is inactive, the processing of the kernel messages will be delayed.

4.2 Peripheral and Radio Drivers

The system software incorporates a number of drivers for the peripheral devices, the radio and some hardware specific blocks of the BLE. Application programmers are able to use the peripheral drivers, however the rest of the drivers are to be used only by the kernel and BLE controller software.

4.3 Real Time Kernel

The DA1458x software platform utilizes a small and efficient Real Time Kernel licensed from Riviera Waves. Almost the complete BLE Stack and most of the application makes use of the services

offered by the Real Time Kernel. The kernel offers task, message, events and dynamic memory capabilities. The tasks communicate with messages which are pushed in a queue whenever a task is trying to send a message to another task. Timers and other hardware events also push events in a queue. Whenever the kernel scheduler is called from the main loop it pops messages and events out of the queues according to their priority and invokes the relevant handler, triggering the execution of the different tasks. The execution continues until the queues are empty.

4.4 Bluetooth Low Energy Software

The BLE software implements the Bluetooth® Low Energy protocol as specified in Version 4.1 of the Bluetooth® standard and is fully compliant with this standard. It is a single-mode BLE implementation, therefore there is no support for the Basic Rate / Enhanced Data Rate protocol (BR/EDR).

The basic parts of the BLE software consists of the Host and Controller related software.

Each layer (ATT, GATT, GAP etc.) of the stack is instantiated as multiple different tasks. The radio events are captured by the drivers and fed to the low level software of the BLE controller. The event handlers spawn messages that trigger the protocol layers. If user action or user notification is required this sequence of messages will dispatch messages to the application or profile task. On the reverse direction, if the user application wants to perform a specific operation it dispatches messages to the lower layers, beginning a sequence of messages that may reach, depending to the operation, the BLE controller and the radio.

On top of the Host, a library of ready to use profiles and some profile utilities are provided to simplify application development. Programmers should interface to the profile, GATT and GAP modules to perform all of the functionality required by a BLE application.

4.5 Application Software

Although the message API provided is powerful and complete it poses some restrictions on how fast and how easy one can build an application. To address this issue in the software release 5.0.x we introduce a new layer depicted as SDK Application, which collects part of the common functionality required to build an application in a library exposing a number of APIs and helper functions. Care has been taken to hide the task management and message management as much as possible from the user and provide a function/callback-like API.

4.6 Memory Organization

To achieve minimum power consumption the DA1458x contains different type of memories each suited better for certain operations. The DA1458x SoC platform contains an embedded 32 kB One-Time-Programmable (OTP) memory for storing the application code and the Bluetooth profiles used. In this, the last bytes of the OTP are used to store configuration and calibration data. 1458x also supports storing the application in external non-volatile memories such as serial flashes or eeprom devices. The protocol stack and the kernel are stored in a dedicated ROM. Application runs on a 42kB System RAM. Application code and data are copied from

Also available is a Low leakage Retention RAM used to store sensitive data and connection information while in Deep Sleep mode.

The memory block sizes are listed below:

- 84 kB ROM. Contains Boot ROM code and Bluetooth Low Energy protocol related code.
- 32 kB One-Time-Programmable (OTP). At power up or reset of the DA1458x, the primary boot code (ROM code) checks if the OTP memory is programmed and if it is, it proceeds with mirroring the OTP contents to System RAM and it programs execution.
- 128 kB Flash (DA14583 only). At power up or reset of the DA14583, the primary boot code (ROM code) loads the secondary Bootloader (from OTP memory or FLASH) and the Secondary Bootloader proceeds with copying the FLASH image to System RAM and it programs execution
- 42 kB System SRAM.
- 8 kB Retention SRAM

4.7 Supported Hardware Configurations

4.7.1 Integrated Processor

The straightforward approach to develop application with the DA1458x family is the integrated processor configuration. All SW components, lower layers (controller), higher layers (host), profiles and the complete application run on the DA14580/581/583 as a single chip solution. This configuration is well suited for many low to mid complexity applications. Please check the complete list of reference designs provided by Dialog on the Customer Support site to understand the complexity of the applications that can be supported.

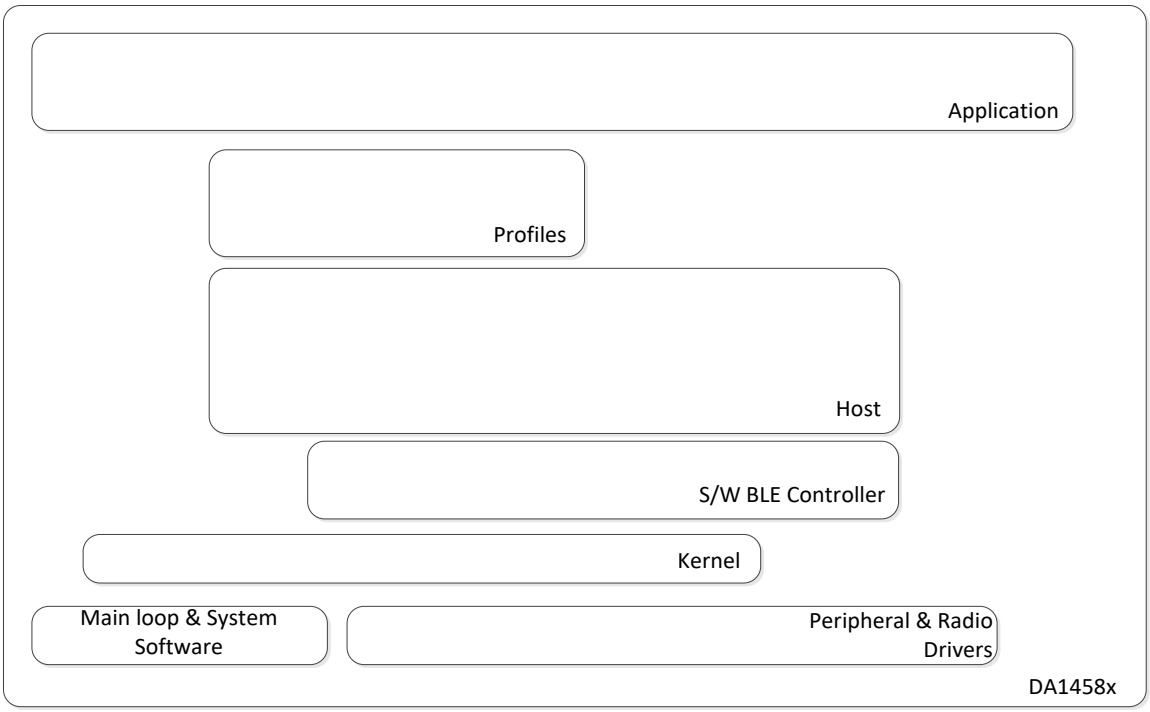


Figure 6: Integrated Processor HW Configuration

4.7.2 External Processor

In cases where an external processor is present or in mid and high complexity applications, the DA1458x can be used as a BLE interface controlled from an external processor via a proprietary protocol called Generic Transport Layer (GTL). The DA1458x can accommodate the link layer, the host protocols and the profiles and the external processor will implement the application functionality. The two components will communicate via GTL over a serial link which can be either UART or SPI. More information on the external processor configuration as well as an example application can be found in Ref. [17].

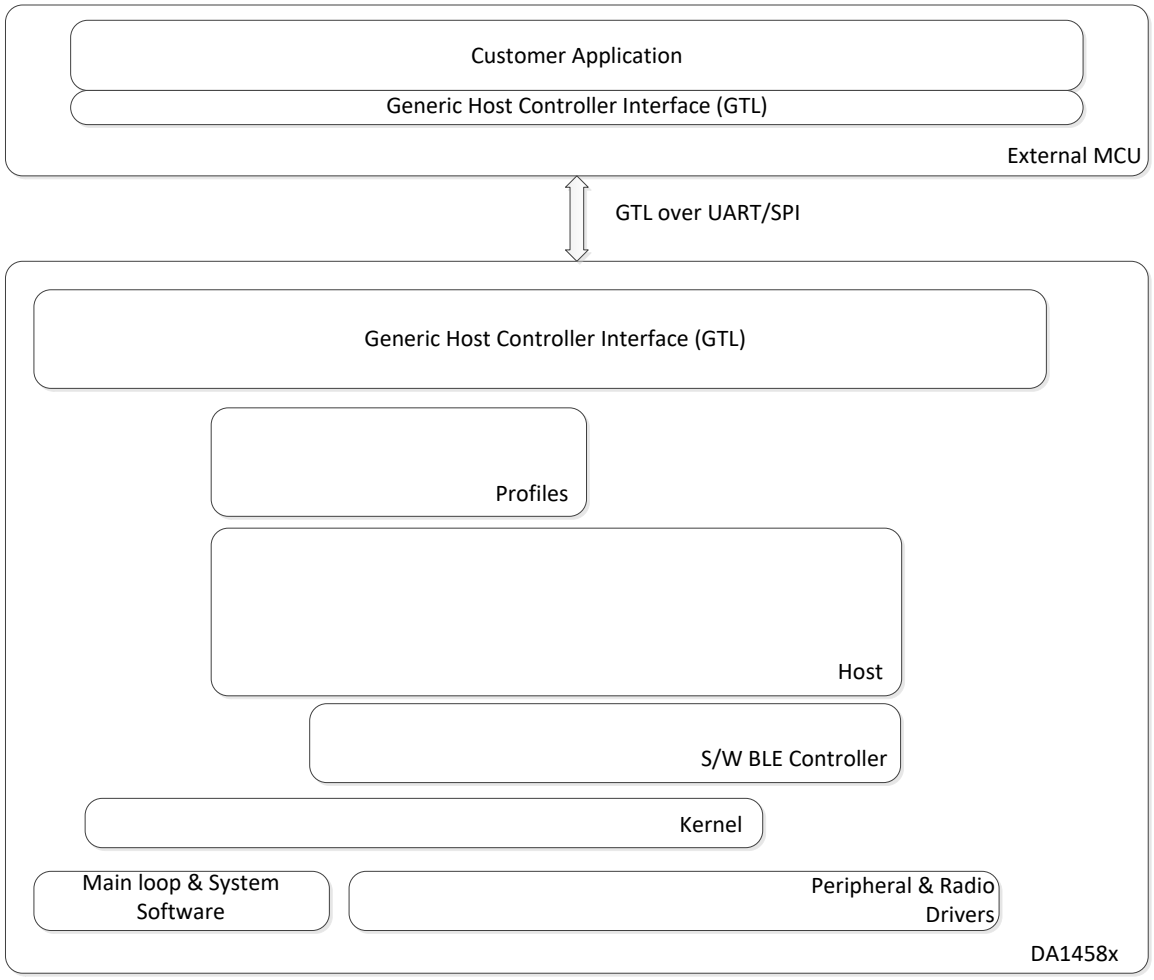


Figure 7: External Processor HW Configuration

4.8 Development Environment

The 5.x series software platform enables development via a Software Development Kit (SDK) that is supported by a number of other tools which aim at assisting the designers and programmers develop new application for the DA1458x family of devices. The projects are based on and supported by the ARM Keil µVision IDE/Debugger, ARM C/C++ Compiler, and essential middleware components, Keil IDE and the Keil build tools. The DA1458x can be interfaced with ARM Segger JTAG cables something that is fully supported by the Keil environment.

To assist with the quick evaluation and programming of the DA1458x chip, Dialog provides a power full graphical tool called SmartSnippets Toolbox. SmartSnippets Toolbox enables connection to the DA1458x devices via UART or JTAG to program the on-chip OTP or even external Flash/EEPROM. A very powerful tool that is included in the SmartSnippets Toolobox to use with the standard Dialog development kits is the Power profiler. The user can track real time the power consumption of the board, set software marker and correlate board consumption with events happening during the execution of the software.

5 Real Time Kernel

5.1 Overview

The DA145x software platform utilizes a small and efficient Real Time Kernel licensed from Riviera Waves. The kernel offers the following features:

- Task creation and state transition.
- Message exchange between tasks.
- Timer management.
- Dynamic memory allocation.
- BLE events scheduling and handling.

5.2 Scheduler

The core of the kernel is a scheduler running in the main loop of the application. The scheduler checks if an event is set and services the pending events by calling the corresponding handler. The event may be a BLE or timer event, a message between two tasks.

Scheduler obtains any timing information from BLE core HW. Main loop code ensures that kernel scheduler is not executed while the HW module of BLE core is not in sleep mode.

The implementation of the kernel resides in ROM memory area; hence the source code files are not included in SDK distribution. The definitions of the API types and the prototypes of API functions can be found in the following header files.

- `ke_task.h` - Kernel task management and creation API.
- `ke_msg.h` - Message handling API.
- `ke_mem.h` - Dynamic memory allocation API.
- `ke_timer.h` - Timer creation and deletion API.

5.3 Tasks

Each standalone software module, i.e. BLE stack layers/GATT profiles/Host application etc., of the DA1458x SDK is instantiates as a kernel's task. The task usually is created at system initialization. The maximum supported number of tasks in a single BLE application is 23.

A task is defined by the task ID number which is a unique identifier per task and a task descriptor structure. The type of the structure is struct `ke_task_desc`, defined in `ke_task.h`. The members of the task descriptor determine the message handlers for each state of the application, the default message handlers, the placeholder of the current task state, the highest valid state of the task and the maximum number of task instances.

The task is created by calling the `ke_task_create()` function. The task ID and descriptor are passed in function parameters.

The state of the task is changed by calling `ke_task_set()`. The current state of the task is returned by `ke_state_set()`.

All the API functions and types of kernel task creation and management are declared in the `ke_task.h` header file.

- Types
  - `ke_msg_handler` – Message handler structure
  - `ke_state_handler` – List of message handlers for a specific or default state.
  - `ke_task_desc` - Task descriptor.
- Functions
  - `ke_task_create()` – Creates a new task.

- ke_state_set() – Sets the state of the task.
- ke_state_get() – Returns the current state of the task.

5.4 Dynamic Memory Allocation

The kernel provides an API to the application code for dynamic memory allocation in the heap memories of the kernel. There are four heap memory areas defined in DA1458x kernel:

- KE_MEM_ENV – Used for environmental variables memory allocation
- KE_MEM_ATT_DB – Used for ATT protocol databases, i.e. services, characteristics, attributes
- KE_MEM_KE_MSG – Used for kernel messages memory allocation
- KE_MEM_NON_RETENTION – General purpose heap memory. If the allocated memory space in this heap is non zero, at a certain point of time then entry in deep sleep mode is not allowed.

The size of the heap memories is determined by the selected sleep mode memory map configuration and the heap memory size configuration parameters in case of deep sleep mode memory map configuration is selected. For further reading regarding SDK configuration reader should refer to [21].

Dynamic memory allocation in any of these heaps is performed by calling the ke_malloc() function. The size and the selection of the heap memory are passed in the parameters of the function. If the memory space in the selected heap memory is insufficient then kernel memory management code will try to allocate the requested memory space in another heap. If memory allocation is failed in all of the heap memories kernel will issue a system software reset.

The API functions provided by the kernel are:

- ke_malloc() – Allocate the requested memory space.
- ke_free() – Free the allocate memory space at the requested memory address.

5.5 Messages

The kernel provides a mechanism for message exchanging between tasks. The messages exchanged by the kernel has specific format. The format is determined by the struct ke_msg type defined in ke_msg.h. The ke_msg structure includes the following members.

- Id: A 16-bit unsigned integer containing the message identifications. The ten least significant bits form a sequential number unique among the messages of the task. In the six most significant bits contain the ID of the task, to ensure the uniqueness of the message identification in the system. The Macro KE_BUILD_ID can be used to build message IDs compliant to this convention.
- dest_id: Task ID of the destination task of the message.
- src_id: Task ID of the source task of the message.
- param_len: Size of the message data contained in param.
- param: The placeholder of the data of message. The type of the structure member is a one position table of 32-bit unsigned integer. However the size of the allocated memory space is determined by the heap memory, allocated by the message memory allocation function and it is equal to param_len.

The transmission of a message is done in three steps:

1. Allocation of a message structure by the sender task. The message allocation is performed by calling one of the following macros:
   - KE_MSG_ALLOC: Allocates space in KE_MEM_KE_MSG heap memory for the message. Message ID, source and destination task IDs and the type of the data of the message are passed in the parameters of the function. Function calculates the memory space to allocate based on the type of the data. Returns a pointer to the start of data of the allocated message.
   - KE_MSG_ALLOC_DYN: Similar to KE_MSG_ALLOC. Additional memory size to the size of data type is passed in an additional parameter.

2. Filling of the message parameters. The code of the source task should fill in the data of the message.
3. Message structure is pushed in the kernel.

The message is sent to the destination task by calling `ke_msg_send()`. The pointer returned by `KE_MSG_ALLOC` or `KE_MSG_ALLOC_DYN` must be passed in the parameter of the function. If the message is allocated but not sent, `ke_msg_free()` must be called to free the allocated memory space.

The reception of a message sent to a task is implemented by defining a message handler function (structure `ke_msg_handler`) in the task descriptor of the message (`ke_task_desc`). A state handler should return `KE_MSG_CONSUMED` when the message is consumed by the destination task and `KE_MSG_NO_FREE` when the message is forwarded to another task. Function `ke_msg_forward()` must be used for this operation.

5.6 Timer

The DA1458x kernel provides timer services to create and delete a timer event. The time reference of kernel timers is the `BLE_GROSS_TIMER` of the BLE HW core. The precision of the `BLE_GROSS_TIMER` timer is 10 ms. The task requested the timer event will be notified for the expiration of the timer by receiving a message. The message ID is equal to the timer ID used for timer creation, hence timer ID must be a valid message ID, as described in section 5.5. A timer handler function must be also defined in the list of task handlers. Kernel timers are one-shot timers.

- API functions
  - `app_timer_set()` – This is a wrapper of `ke_timer_set()`. Timer ID, task ID and timeout in units of 10 ms. The maximum valid timeout is 30000, which corresponds to a 5 min period.
  - `ke_timer_delete()` – Deletes an active kernel timer.

6 Bluetooth Low Energy Software

In this section the Bluetooth Low Energy layers of the software architecture are described.

6.1 Overview

Figure 8 presents a diagram of the Bluetooth Low Energy software architecture.

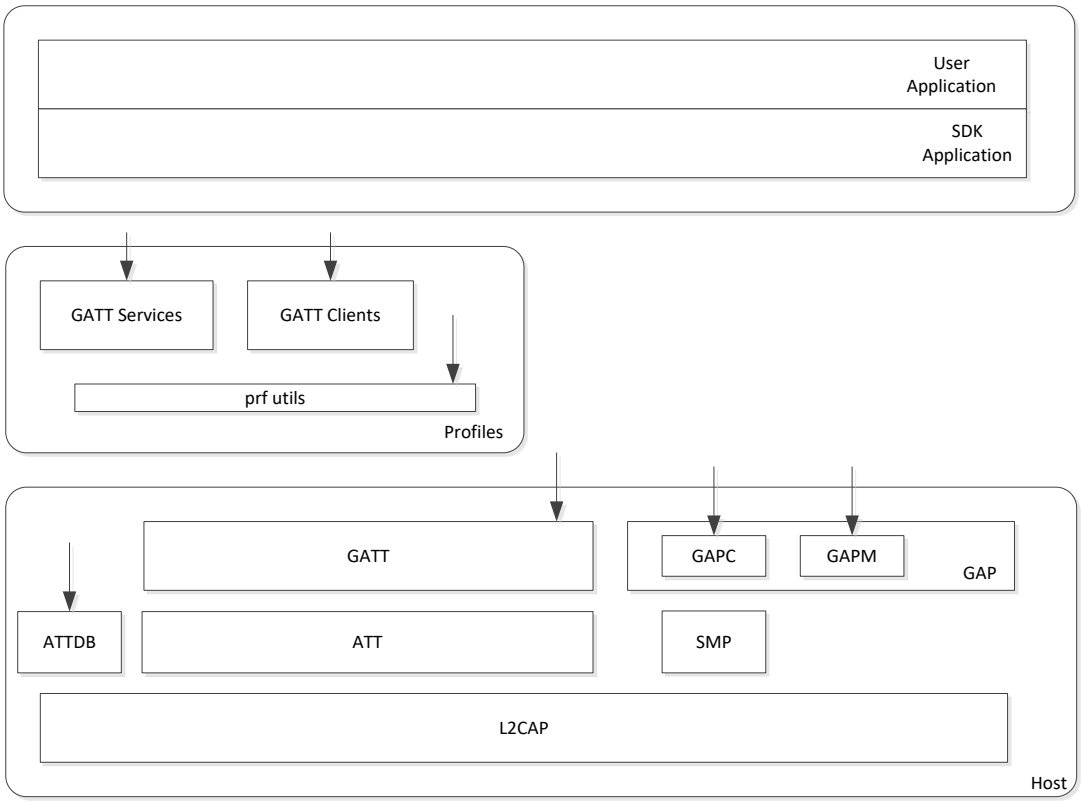


Figure 8: Bluetooth Low Energy Software

The BLE software implements the Bluetooth® Low Energy protocol as specified in Version 4.1 of the Bluetooth standard and is fully compliant with this standard. It is a single-mode BLE implementation, which means that there is no support for the Basic Rate/Enhanced Data Rate protocol (BR/EDR).

Most of the BLE controller of the DA1458x is implemented in hardware. On top of the controller the Host is implemented with all the required layers. The L2CAP and SMP are internal layers within the stack and are not intended for direct access from the application, although this is possible. Most of the ATT layer should not be accessed from the application directly, since its functionality is provided through GATT. The attribute database API may be used to implement profile related functionality.

The GAP and GATT layers are built on top of the ATT and SMP. Those two layers form the borderline of the Host with the application. The majority of the operations can be performed using those two APIs. To further assist the development, DA1458x SDKs provide a library of ready to use profile implementations. Those profiles make use of the GATT and ATT APIs and a set of helper functions gathered within the `prf_utils.c`. The profiles expose a message API and a set of function calls to the application.

The code of the Host is precompiled and burned into ROM while the profile implementation and the `prf_utils.c` are provided in source code, compiled with the application and run in RAM. The symbols of the ROM are exposed into the `rom_symdef.txt` file and linked with the user application during the build process.

The arrows in Figure 8 designate the APIs that can be used by a user application. Moreover these are also listed in Table 1 as a reference.

Table 1. Bluetooth Low Energy Software API

API	API Files	Comments
ATT Database	attm_db.h attm_util.h attm_db_128.h	Please refer to ATTDB Interface Specification (RW-BLE-ATTDB-IS), Riviera Waves.
GAP Manager	gapm.h gapm_task.h gapm_util.h	Please refer to GAP Interface Specification (RW-BLE-GAP-IS), Riviera Waves.
GAP Controller	gapc.h gapc_task.h	Please refer to GAP Interface Specification (RW-BLE-GAP-IS), Riviera Waves.
GATT Manager	gattm.h gattm_task.h	Please refer to GATT Interface Specification (RW-BLE-GATT-IS), Riviera Waves.
GATT Controller	gattc.h gattc_task.h	Please refer to GATT Interface Specification (RW-BLE-GATT-IS), Riviera Waves.
Profile Utilities	prf_utils.h prf_utils_128.h prf_types.h	Please refer to the header files in the code.
<Profiles>	<profile_short>.h < profile_short >_task.h	Please refer to the header files in the code and the documents provided for every profile in the support site. To locate the relevant header file, you need to find the short name for every profile. For example the short name for the battery service client is basc while for the battery server bass. For a complete list of all the profiles please refer to the support site or look under the sdk_profiles in the empty template project.
Custom Profile	custs1_task.h custs1.h custs2_task.h custs2.h	Please refer to the header files in the code.

The following sections describe the role and usage of the various APIs. For more information please refer to the documents that describe each layer.

6.2 GAP

The RW-BLE Generic Access Profile (GAP) defines the basic procedures related to the discovery and link management of Bluetooth devices. Furthermore, it defines procedures related to the use of different LE security modes and levels. For a detailed description of the API refer to [11].

The RW-BLE GAP provides complete and substantial support of the LE GAP:

- Four Roles – central, peripheral, broadcaster and scanner.
- Broadcast and Scan.
- Modes – Discovery, Connectivity, Bonding.
- Security with Authentication, Encryption and Signing.
- Link Establishment and Detachment.
- Random and Static Addresses.
- Privacy Features.
- Pairing and Key Generation.

The RW-BLE GAP is divided into two parts: the GAP Manager (GAPM) and the GAP Controller (GAPC). The GAP Manager manages all application requests that are not related to an established

link. The GAP Controller (called GAPC) on the other hand is created upon a connection to a peer device and deleted when the connection is terminated.

The GAP Manager initialization takes place in the `system_init()` function. When the GAP entity has been initialized and is ready to provide services to the upper layers, it dispatches the `GAPM_DEVICE_READY_IND` message. This message is handled internally in the GAP application module. The GAP module responds by configuring the GAP Manager to a specific role, sending a `GAPM_SET_DEV_CONFIG` message. Upon completion of the GAP Manager configuration the device is ready to initialize the databases, and then according to the selected role perform operations such as scanning and advertising to establish connections with other peers.

For example, in order to start advertising, the `GAPM_START_ADVERTISE_CMD` message is compiled and sent to the GAPM. A message is allocated using the `KE_MSG_ALLOC` macro, e.g.:

```
struct gapm_start_advertise_cmd* cmd = KE_MSG_ALLOC(GAPM_START_ADVERTISE_CMD,
TASK_GAPM, TASK_APP, gapm_start_advertise_cmd);
```

The message is filled with the necessary data:

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

and is then sent to the GAPM task. The result is to start an undirected advertising operation.

Please refer to [11] for a complete list of the supported messages and operations of the GAPM.

When a remote peer tries to connect to the device, the GAP Controller will report the peer's request with the `GAPC_CONNECTION_REQ_IND` to the application. Since the stack is now referring to a specific connection, the messages will now be delivered from/to the related GAP Controller. The application GAP handler will respond to this request with the desired security requirements and it will try to establish a secure or non-secure connection with the peer according to the requirements. The security management protocol messages arrive to the application through the GAP Controller. Upon establishing a secure connection, the device is connected and it can start using the available profile services. Each service may have its own security requirement and establishing a connection with adequate security will expose the services to the peer.

Please refer to [11] for a complete list of the supported messages and operations of the GAPC.

6.3 BLE Data Services

After establishing the connection with the peer device the GAP related operations are only to manage certain aspects of the connection such as change connection parameters, security level or dropping the connection. The data services of a BLE connection are provided through the database. The BLE stack provides to the application numerous APIs to manage and exchange data with local and remote databases. Those are the GATT Manager and GATT Controller API. Alternatively the programmer can also make use of the Attribute database API (ATTDB) directly instead of going through GATT to add and manage services and characteristics in the database. The reason behind this is to produce smaller and efficient code. User can use either approach, through GATT or directly to ATTDB.

6.3.1 GATT

Similar to GAP, GATT consists of two modules: the GATT Manager and the GATT Controller. The GATT Manager is not related to a specific connection and it is instantiated once to provide a message API used to manage the internal attribute database. The GATT Manager is placed between the profile or the application and the ATT Manager and conveys messages between them. It provides service and attribute operations such as adding services and characteristics to the database, setting and getting permission level and setting and getting values to attributes.

The GATT controller is related to a specific BLE connection instance. The interface is used in both client and server roles. In a client role the interface is used to discover, read and write attributes of peer devices and to receive notifications or indications. In a server role, this interface is notified when modification of the database is requested by the peer device, and to send indications or notifications.

For a detailed description of the GATT Controller please refer to [12].

6.3.2 ATTDB

Instead of using the GATT Manager API, the user can directly access the native attribute database API. The ATTDB API provides functions to add elements in the database (services and attributes), choose the position of the element in the database, hide/show attributes as well as manage the permissions and their value. For a detailed description of the ATTDB supported functions please refer to attm_db.h, attm_db_128.h and attm_util.h files and in [13].

6.4 Bluetooth LE Profiles

The SDK provides implementations of various profiles each with its own profile specific API exposed to the application. Please check the dialog support website for the most updated list of the profiles supported from the SDK. There are two distinct roles: the server role and the client role. The server exposes a profile database on the local device. A client reads, writes and manages the database of remote device.

Every profile server implements generic and profile specific functionality. The generic part includes:

- The profile database description, with all the services and attributes.
- The profile initialization function to create the profile task.
- The database creation handler that creates the database when the application task issues the relevant message.
- The profile enable handler that enables the database when the application task issues the relevant message.
- The profile disconnect handler that takes care of profile housekeeping operations when the connection drops.

Depending on the profile, the profile specific part may include update of specific attributes with or without notification and handling and verification of write attempts from peer devices.

Similar principles apply for the client profile as well. The client will include a description of the services expected, functions and message handlers to initialize, enable the client, housekeep the discovery results and handlers for errors and disconnections. Depending on the specific profile special handlers for read and write operation and reception of notifications may exist, tied to specific attributes. For detailed information please refer to the available documents for every profile located in the support site, or browse through the <profile_short>.h and < profile_short >_task.h header files.

The SDK provides a useful set of helper functions to access the GATT and ATTDB services. They are located in the prf_utils.h and are used frequently in the existing profile code. For detailed information please refer to prf_utils.h.

7 System Software

The system software of the DA1458x SDK consists of the following modules:

- Main function and main loop of the application.
- Sleep and power management software.
- BLE events IRQ handlers.
- Patched functions of the ROM code.
- BLE stack and system configuration.
- Finally, the system software exports a number of APIs to application code to facilitate control of sleep mode, execution of application software in main loop and provide development/debug capabilities.

7.1 Main Loop and Sleep Modes

7.1.1 Sleep Modes

This document describes the software architecture of the sleep modes of DA14580/581/583. The various modes of operation of the chip are:

- Active mode.
- Extended Sleep mode.
- Deep Sleep mode.

In Active mode, the system domain (ARM processor, SysRAM, ROM, etc.), the radio (including Radio and Bluetooth Low Energy core) and the peripheral domains (UART1/2, I2C, SPI, etc.) are active.

In Extended Sleep mode, the system domain except the SysRAM, the radio domain and the peripheral domain are powered down and the XTAL16M clock is stopped. The SysRAM is still powered to retain data but is not accessible. The Always ON (AON) power domain is active to keep data in the retention RAMs and to supply power to the blocks that can wake the system up, i.e. wakeup timer, quadrature decoder and the BLE timer.

In Deep Sleep mode, to reduce power consumption even further, the SysRAM is also powered down. The status of the other power domains is the same as in Extended Sleep mode.

7.1.2 Wake-Up Events

When in any of the above mentioned sleep modes, DA14580/581/583 can be woken up in two ways:

1. Synchronously, via the BLE timer which can be programmed to wake up the system in order to serve a BLE event and
2. Asynchronously, via the Wakeup Timer and Quadrature Decoder if triggered by an external event (input).

In a BLE application, DA14580/581/583 could be set to either of the above mentioned sleep modes. For an advertising event, connection event or other wireless communication event, DA14580/581 needs to be woken up and go to Active mode in order to send/receive packets over a BLE wireless link. Since these events are time based, the BLE timer is used to wake up the system, including the BLE core, the radio, the ARM processor and the rest of the blocks. In this case, the following convention is used: “The system is woken up synchronously with the BLE core”.

When in Extended/Deep sleep mode, DA14580/581/583 can also be woken up by an external event and after waking up the ARM processor can perform some functions. However, at that moment it may not yet be the time for a BLE communication event, e.g. a connection event and thus the BLE and the radio can remain in power off state. In this case, the following convention is used: “The system is woken up asynchronously with the BLE core”.

If the system is woken up asynchronously then any requests for transmission of messages to kernel tasks (and, eventually, over the Bluetooth wireless link) cannot be performed immediately and must be synchronized to the BLE core. This is because the stack is built in such a way that the handling of

message events requires the BLE core to be active so that timing information from the BLE core is available. This timing information is not available when the BLE core is powered down.

7.1.3 Main Loop

In the SDK release 5.0.2 (or later) the main loop has be refactored to make it easier to understand for the programmer. Although the application programmer should not alter the main loop since it is considered part of the SDK, most of the programmers will want to go through the main loop execution to understand the application flow.

```
while(1)
{
    do {
        // schedule all pending events
        schedule_while_ble_on();        <- Check BLE core state and run kernel scheduler – [main callback: app_on_ble_powered]
    }
    while ((app_asynch_proc()));    //grant control to the application, try to go to power down
                                    <- [main callback: app_on_system_powered]
    //wait for interrupt and go to sleep if this is allowed
    if (((!BLE_APP_PRESENT) && (check_gtl_state())) || (BLE_APP_PRESENT))
    {
        GLOBAL_INT_STOP(); //Disable the interrupts

        app_asynch_sleep_proc();        <- [main callback: app_before_sleep]

        // get the allowed sleep mode
        sleep_mode = rwip_power_down();     <- Check entry to sleep mode

        if ((sleep_mode == mode_ext_sleep) || (sleep_mode == mode_deep_sleep)) {
            //power down the radio and whatever is allowed
            arch_goto_sleep(sleep_mode);    <- [main callback: app_validate_sleep/app_going_to_sleep]

            WFI();//wait for an interrupt to resume operation
                  <- Wait for interrupt to exit sleep mode
            //resume operation
            arch_resume_from_sleep();       <- [main callback: app_resume_from_sleep]
        }
        else if (sleep_mode == mode_idle)
        {
            if (((!BLE_APP_PRESENT) && check_gtl_state()) || (BLE_APP_PRESENT))
                WFI(); //wait for an interrupt to resume operation
        }           <- Idle state. Wait for interrupt to resume main loop
        // restore interrupts
        GLOBAL_INT_START();
    }

    if (USE_WDOG)
        wdg_reload(WATCHDOG_DEFAULT_PERIOD);
}
```

Figure 9: The Main Loop

The main loop consists of two parts. The first part is executed while the CPU is active and for as long as the kernel or the application wants the CPU to remain active. In the second part of the main loop the program attempts to go into power down mode. It will try to shut down the BLE hardware and then the rest of the peripherals and set the CPU in a low power state while waiting for interrupt (WFI), either from some external pin or a BLE programmed event.

In the active part of the main loop, the kernel will be granted control with a call to `rwip_schedule()` and it will keep the control for as long as messages and events need to be handled. The call to `rwip_schedule()`happens in `schedule_while_ble_on()` since the kernel requires the BLE hardware to be active to process messages. Within `schedule_while_ble_on()` the application is also granted control through the `user_app_main_loop_callbacks.app_on_ble_powered` function pointer. The application can force the main loop to stay within the `schedule_while_ble_on()` given that the BLE remains active according to the return value of `app_on_ble_powered`. If both application and kernel allow `scheduler_while_ble_on()` to return, the control will be granted again to the application via the `app_asynch_proc()` function and the `user_app_main_loop_callbacks.app_on_system_powered` function pointer. This is required when the application wants the main loop to remain in an active state while the BLE is powered off. The return value of `app_on_system_powered` controls in a similar

way to the while loop. If the application decides that nothing else should be done the software will try to go into a power down state.

During this state the application will be granted control three times before going into sleep: once just before starting the power down sequence (`user_app_main_loop_callbacks.app_before_sleep`), once to validate the sleep before closing down the peripherals (`user_app_main_loop_callbacks.app_before_sleep`) and finally just before the `WFI()` is called for the final housekeeping jobs (`user_app_main_loop_callbacks. app_going_to_sleep`).

The application will be called when the main loop resumes via `user_app_main_loop_callbacks.app_resume_from_sleep` and the software will return to the first part of the main loop and the cycle will start again.

7.2 System API

7.2.1 Main Loop Callbacks

The system software API exports a number of application callback functions called in the main function of SDK projects. The application callback functions are defined by `struct arch_main_loop_callbacks` type variable `user_app_main_loop_callbacks` in the `user_callbacks_config.h` header file. In case there is no application task to run in any of the callback functions of the structure, a NULL function should be assigned to the callback member.

Table 2: Callback Functions

#	Function Name	Description	Timing Constraints
1	`app_on_init()`	Called at system start up. Application tasks related to application initialization can be called here.	None
2	`app_on_ble_powered()`	Called if BLE core is active. It is usually used for sending messages to kernel tasks generated from asynchronous events that have been processed in app_on_system_powered(). **Note:** By default the watchdog timer is reloaded and resumed when the system wakes up. The user has to take into account the watchdog timer handling (keep it running, freeze it, reload it, resume it, etc.).	Medium
3	`app_on_system_powered()`	Called if system domain (processor is active) while BLE core can be in sleep mode. Usually used for processing of asynchronous events at “user” level. The corresponding ISRs should be kept as short as possible and the remaining processing should be done at this point. **Note:** By default the watchdog timer is reloaded and resumed when the system wakes up. The user has to take into account the watchdog timer handling (keep it running, freeze it, reload it, resume it, etc.).	Medium
4	`app_before_sleep()`	Used for updating the state of the application based on the latest status just before sleep checking starts.	Medium
5	`app_validate_sleep()`	Used to allow cancelling the entry to Extended or Deep sleep based on the current application state. The BLE and the Radio are still powered off but the other of the power domains stay active.	Hard
6	`app_going_to_sleep()`	Used for application specific tasks just before entering the low power mode.	Hard
7	`app_resume_from_sleep()`	Used for application specific tasks immediately after exiting the low power mode.	Hard

7.2.2 Sleep API

The system software provides a Sleep API to the application to modify the mode of operation (Active, Extended sleep, Deep sleep). The API is defined in the `arch_sleep.h` header file and provides access to the functions listed in Table 3.

Table 3: Sleep API Functions

#	API Function	Description
1	`void arch_disable_sleep(void)`	Disables all sleep modes. The system is either in idle or active state.
2	`void arch_set_extended_sleep(void)`	Activates extended sleep mode.
3	`void arch_set_deep_sleep(void)`	Activates deep sleep mode.

4	`uint8_t arch_get_sleep_mode(void)`	Returns the current mode of operation.
		`0: Sleep is disabled`
		`1: Extended sleep`
		`2: Deep sleep`

5	`void arch_force_active_mode(void)`	If sleep is on then it is disabled. The current sleep mode (before setting it to Active) is stored in order to be able to restore it, if needed.
6	`void arch_restore_sleep_mode(void)`	Restores the previous sleep mode (if any) that was changed with a call to app_force_active_mode(). The application must not have modified the sleep mode in the meantime.
7	`void arch_ble_ext_wakeup_on(void)`	Put BLE into permanent sleep waiting a forced wakeup. After waking up from an external event, if the system has to wake BLE up then it must restore the default mode of operation by calling app_ble_ext_wakeup_off() or the BLE won't be able to wake up in order to serve BLE events!
8	`void arch_ble_ext_wakeup_off(void)`	Restore BLE cores' operation to default mode. In this mode, the BLE core will wake up every 10sec even if no BLE events are scheduled. If an event has been scheduled earlier, then the BLE core will wake up sooner to serve it.

9	`bool arch_ble_ext_wakeup_get(void)`	Returns the current mode of operation of the BLE core:
		`false: default mode`
		`true: permanent sleep, external wake-up is required.`

10	`bool arch_ble_force_wakeup(void)`	If the BLE core is sleeping (permanently or not), this function wakes it up. A call to arch_ble_ext_wakeup_off() should follow in case of permanent sleep.
11	`uint8_t arch_last_rwble_evt_get(void)`	Returns a value that informs about the last BLE or radio interrupt that has occurred. The values returned by this function are defined in last_ble_evt enumeration in arch_sleep.h. It can be used to synchronise asynchronous tasks, which are executed in previously described hooks, with BLE or radio events.

Note that the `arch_sleep.c` module monitors the number of calls to `arch_force_active_mode()` (the counter is incremented) and to `arch_restore_sleep_mode()` (the counter is decremented). In order for `arch_restore_sleep_mode()` to actually enable sleep mode, the counter must be zero! This means that the application must ensure that `arch_restore_sleep_mode()` is called (at least) as many times as `arch_force_active_mode()`. If the application is split into different modules, this rule applies to each module separately.

Finally, note that the Debugger cannot be used in any of the sleep modes because it has to be turned off in order to allow powering down the System power domain.

7.2.3 Serial Logging Interface API

The system software provides a Serial Logging Interface API. The programmer can use this interface for logging purposes or to communicate with external systems via the UART. The API is defined in the `arch_console.h` header file. It provides access to the functions listed in Table 4.

Table 4: Serial Logging API Functions

#	API Function	Description
1	`void arch_puts(const char *s);`	Put string function. Push the string `s` to the UART queue.
2	`int arch_printf(const char *fmt, ...);`	Printf function. Place the formatted output into the UART queue.
3	`void arch_printf_process(void);`	This function is called periodically from the main loop to push the contents of the queue into the UART.

Before using the serial output the user should enable the API by defining the `CFG_PRINTF` flag. Additionally, the user should select the proper UART device by defining the `CFG_PRINTF_UART2` flag or not and including the proper driver in the `sdk_driver` group of the project.

7.2.4 BLE Statistics API

The system software provides a simple API to collect statistics about a given connection. When enabled with the `CFG_BLE_METRICS`, code is added in the receive interrupts that counts the errors during the communication of the device with a remote peer. The data are kept in a global structure of the following type:

```
typedef struct
{
    uint32_t    rx_pkt;
    uint32_t    rx_err;
    uint32_t    rx_err_crc;
    uint32_t    rx_err_sync;
}arch_ble_metrics_t;
```

The user can get the pointer to this structure by calling the `arch_ble_metrics_get()` function and reset the contents by calling the `arch_ble_metrics_reset()` function.

7.2.5 Development Mode API

To track common mistakes and to ease development, the SDK software provides a configuration flag the programmer can use to enable the development debug mode, called `CFG_DEVELOPMENT_DEBUG`. This flag should be disabled for production software. If enabled the following features are available:

- The SysRAM is never powered down in Deep sleep mode, allowing the developers to run applications in Deep sleep without the need to program the OTP.
- A validation is provided that the GPIO pins are not used for more than one function.
- Breakpoints upon Hard Fault, NMI and assert conditions are automatically issued to help the developer attach the Debugger and trace the cause of the error.

7.2.5.1 GPIO Reservation

In development debug mode, before issuing an operation through the GPIO driver, the application should reserve the GPIO pin using the `RESERVE_GPIO(name, port, pin, func)` function. Trying to reserve a pin that has already been reserved or trying to use an unreserved pin will halt the application with a breakpoint.

The GPIO reservation feature can be disabled even in development-debug mode by defining the `GPIO_DRV_PIN_ALLOC_MON_DISABLED` flag to reserve memory resources.

7.2.5.2 Assert, NMI and Hard Fault Handlers

In development debug mode, the `ASSERT_ERROR()` and `ASSERT_WARNING()` macros are defined as breakpoints. Upon detecting erroneous condition the program will halt and the user can attach the debugger and figure out what caused the error. In production mode, (`CFG_DEVELOPMENT_DEBUG` is undefined), the `ASSERT_WARNING()` does nothing while the `ASSERT_ERROR()` will cause the program to stay in a `while(1)` loop waiting for a watchdog reset.

Similar to the ASSERT logic, both Hard Fault and NMI handlers are set as breakpoints in development mode and will cause a reset in production mode.

In the case of development mode, both handlers will keep a copy of the processor state in the retention RAM before issuing the breakpoint. The state information saved consists of the following register values R0-R3, R12, LR, PC, PSR, SP, CFSR, HFSR, AFSR, MMAR, BFAR. The user can look into the state information to try and trace which command has issued the NMI or Hard Fault.

The NMI handler will save the state in address 0x81850 and Hard Fault handler in address 0x81800.

7.2.6 Advanced Features API

For details see Appendix E.

7.2.6.1 Wake-Up and External Processor Configuration

In the external processor applications, the device should be able to wake up the external processor when a message is sent via the GTL, and to be woken up from the external processor, when the host application requires it. This feature is controlled by two configuration flags:

- `CFG_EXTERNAL_WAKEUP`: Defining this flag enables the DA1458x to be woken up by the external processor running the host application, by toggling the state of a pin.
- `CFG_WAKEUP_EXT_PROCESSOR`: Defining this flag enables pulsing a pin during a GTL transmission.

The pin and parameters used to wake up the DA1458x are defined in file `user_periph_setup.h` with the following definitions: `EXTERNAL_WAKEUP_GPIO_PORT`, `EXTERNAL_WAKEUP_GPIO_PIN` and `EXTERNAL_WAKEUP_GPIO_POLARITY`. The pin used to signal to the external host that a GTL message is sent is defined with the `EXT_WAKEUP_PORT` and `EXT_WAKEUP_PIN` definitions in the same header file.

7.2.6.2 True Random Number Generator (TRNG)

The programmer can get a 128-bit random number by calling the function `trng_acquire()`, which is defined as follows:

```
void trng_acquire(uint8_t *trng_bits_ptr)
```

The 128-bit (16 byte) random number is returned in the `trng_bits_ptr` pointer. To enable the True Random Number Generator you need to define the `CFG_TRNG` flag. A random number is generated at system initialization and used to seed the C standard library random number generator.

7.2.6.3 Boost Output Voltage (DCDC_VBAT3V)

In the case of a boost voltage configuration, a function `syscntl_set_dcdc_vbat3v_level()` has been provided to set the output voltage of the boost converter. This function is defined as follows:

```
void syscntl_set_dcdc_vbat3v_level(enum SYSCNTL_DCDC_VBAT3V_LEVEL level)
```

This function will set the output level according to the following enumeration:

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


7.2.6.4 Near Field Control

The output power for all the active connections can be controlled to further minimize power consumption. The following functions are provided:

- `void rf_nfm_enable(void)`: Enables Near Field mode.
- `void rf_nfm_disable(void)`: Disables Near Field mode.
- `bool rf_nfm_is_enabled(void)`: Checks if Near Field mode is enabled (true) or not (false).

7.2.6.5 AES Crypto

The DA1458x SoCs come with an AES encryption hardware engine. The engine is used from the stack but can also be invoked from the user if required. The communication must be done through a synchronous (over message) way to ensure that the use of the API will not conflict with operations invoked from the stack. A software-only implementation of the AES is also provided, which a programmer can use as an alternative. Please refer to Appendix E.5 for more information.

7.2.6.6 Co-Existence

A set of functions has been provided for handling the WLAN co-existence. Two WLAN incoming inputs are supported and a BLE priority output. Functions are provided to handle priority per connection and Bluetooth state. Please refer to Appendix E.6 for more information.

8 Application Software

8.1 Overview

Messages are generated from different layers of the BLE stack and the profile code to signal events to the application task. The application on the other hand generates messages and sends them to the stack to begin numerous operations.

There are two distinct directions for the messages.

- Messages from the stack and the profiles to the application task.
- Messages from the application task to the stack and the profiles.

Although the message API provided is powerful and complete it poses some restrictions on how fast an application can be built and the minimum amount of knowledge about the system required to build even a simple application. To address this issue in the 5.0.x SDK, part of the common functionality required to build an application has been collected in a library, which exposes a number of APIs and helper functions. Care has been taken to hide the task management and message management as much as possible from the user and to provide a function/callback like API.

In addition, most of the parameters required to perform operations have been turned into constants that are defined in the user space. A set of working constants are predefined for the user in the template project.

8.2 API

Figure 10 shows a detailed diagram of the application architecture, consisting of two parts:

- **SDK Application**: This part is defined in the SDK and implements the library functionality.
- **User Application**: This part contains configuration constants and callback definitions and is provided from the template. It configures the operation of the SDK application library. The source code of the actual user application resides here.

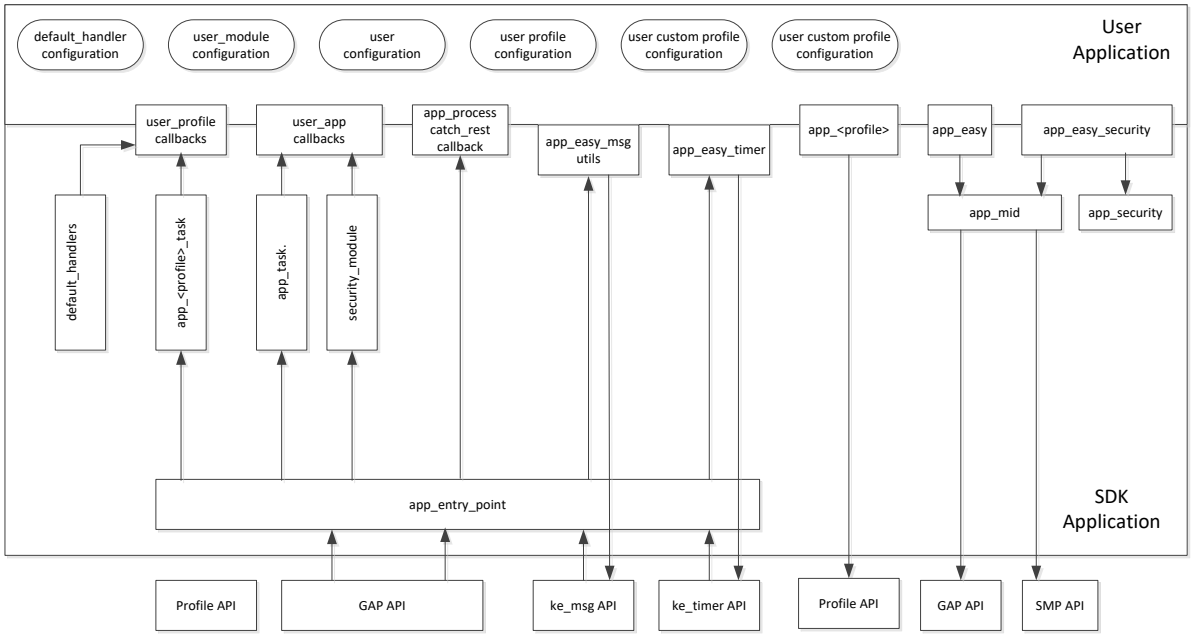


Figure 10: Application Architecture

As mentioned earlier, there are two message directions. The API description starts with the possible ways the User Application can send messages toward the stack and the profiles. Next it describes the available ways the User Application can get messages from other tasks and profiles.

8.2.1 Message API

The standard and most versatile way of executing operations in the stack and profiles is by using the message API. This includes the standard message allocation and message sending functions of the kernel together with the message list supported from every task. The message list and the data structure of each message are usually defined in the file `<destination_task>_task.h`. Please refer to the relevant documents of each task (`gapm`, `gapc`, `gatt`, etc.) for further details.

Whenever a function is not provided from the `app_mid.h` and `app_easy.h` APIs, described later in this document, the user can use the message API.

8.2.2 Mid Layer API

The Mid Layer API is a light stateless set of macros to describe how the most common operations for the gap and security are performed. It describes the generation of a specific message, the filling of the message with data and the dispatch of the message to the required task. The idea behind it is to provide a function API rather than a message API. The following pattern is used:

- `<message pointer*> app_<message_name>_msg_create()`: Allocates a message and fills in the correct destination task.
- `void app_<message_name>_msg_send(<message pointer*>)`: Sends the message.
- `void <operation>_op(parameters)`: Performs a complete operation. The `<operation>_op` function usually creates a message, fills the message with the required data as provided from the caller through the parameters, and dispatches the message.

The complete Mid Layer API is described in the `app_mid.h` file.

8.2.3 Easy API

The Easy API concept tries to reduce the burden of the application programmer concerning message handling, task handling and special sequences that are required to correctly perform some actions in the stack or kernel.

The Easy API also addresses the fact that the majority of the data used in the messages are constants at compile time. Asking the programmer to provide all of the constants when performing a function call is cumbersome and produces larger code. Moreover, there are a lot of these constants that need to be defined even for the simplest of applications, making it challenging for a novice programmer to quickly bring his program into an operational state.

For this reason most if not all of the constants are predefined in the user application space from the template and accessed from the Easy API at compile time. The programmer can quickly bring up his application and then alter the default behavior. Usually altering the behavior only requires changing the proper constant parameters and recompiling the code. As a plus, the small number of parameters used in the function calls of the Easy API provides better readable code and better visibility to the programmer.

The design pattern used for the Easy API is shown in Figure 11.

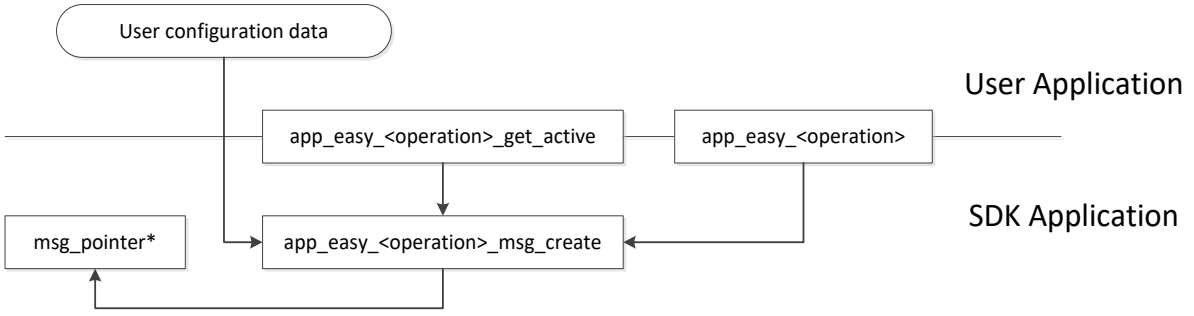


Figure 11: Easy API Design Pattern

Every app_easy_<operation> exposed to the user has a static app_easy_<operation>_msg_create function and a pointer (msg_pointer). When the user calls the app_easy_<operation> function, the relevant msg_create function is called. The message create function will check if the msg_pointer points to a message that has already been created for this operation or not. If a message is already there, it will return the existing message to the app_easy_<operation> function. If no message exists it will create a new one, fill it with constant data provided from the user configuration data files and assign it to the msg_pointer. The app_easy_<operation> will send the message and clear the msg_pointer.

This means that every time the app_easy_<operation> either a new message is created or the existing message is sent and automatically consumed. To support dynamic configuration of the messages the app_easy_<operation>_get_active function is provided. This function will call the message create function and return the pointer to the active message. This way the user application can alter the message dynamically.

To save memory space, there are operations that may share the msg_pointer, for example the different app_easy_<type>_advertise operations.

8.2.4 app_<profile> API

For some of the profiles provided with the SDK an additional application layer exists, which exposes the create database, initialization and profile specific operation functions. Please refer to the relevant header file for the list of functions supported for each profile.

8.2.5 App Entry Point API

In SDK 5.0.2 we have introduced a new piece of software that handles the messages arriving in the application task. This module is called app_entry_point.c. The messages arriving to the app_entry_point_handler will be delivered to all the included application modules of the SDK until a module acknowledges that the message has been handled, in which case the application entry point will return the state of the message as it has been reported, to the kernel scheduler.

In case a message has not been handled by any module, it will be delivered to the application through the app_process_catch_rest_cb function pointer where the user can hook his message handler. The messages arriving at the user space are always considered consumed and cannot be saved or forwarded.

Together with the app_entry_point_handler and the app_process_catch_rest_cb function pointer, the app_entry_point.c module also exposes a set of EXCLUDE_DLG_<MODULE> options defined in user_modules_config.h. These options can be used to disable specific message handlers of the SDK to receive the messages in the user application space via the app_process_catch_rest_cb. This allows the user to override or extend the functionality of the existing SDK modules without having to touch any SDK files.

8.2.6 User Callback API

The SDK provides numerous modules that are hooked on the entry point to handle messages sent to the application task. The primary function of the modules is to transform the received messages into events, which are exposed to the user application space as function pointers with meaningful names and parameters. There are also modules that implement a lot of application functionality, such as the app_spotar_task.c. The modules following this pattern are the gap, security, and all the existing app_<profile>_task.c modules.

The library looks for the value of the pointers in the user_callback_config.h. There the user can assign a new function of his application to be called upon a specific event. The gap and security function pointers are grouped in the user_app_callbacks structure, while the profile related pointers (for profiles with an app_<profile>_task.c) are grouped in the user_profile_callbacks structure.


```
struct app_callbacks{
  void (*app_on_connection) (const uint8_t, struct gapc_connection_req_ind const *);
  void (*app_on_disconnect) (struct gapc_disconnect_ind const *); //app disconnect
  void (*app_on_update_params_rejected) (const uint8_t);
  void (*app_on_update_params_complete)(void);
  void (*app_on_set_dev_config_complete)(void);
  void (*app_on_adv_undirect_complete) (const uint8_t);
  void (*app_on_adv_direct_complete) (const uint8_t);
  void (*app_on_db_init_complete)(void);
  void (*app_on_scanning_completed)(void);
  void (*app_on_adv_report_ind)(struct gapm_adv_report_ind const *);
  void (*app_on_connect_failed)(void);
  void (*app_on_pairing_request) (uint8_t const, struct gapc_bond_req_ind const *);
  void (*app_on_tk_exch_nomitm) (uint8_t const, struct gapc_bond_req_ind const *);
  void (*app_on_irk_exch)(struct gapc_bond_req_ind const *);
  void (*app_on_csrk_exch) (uint8_t const, struct gapc_bond_req_ind const *);
  void (*app_on_ltk_exch) (uint8_t const, struct gapc_bond_req_ind const *);
  void (*app_on_pairing_succeded)(void);
  void (*app_on_encrypt_ind) (const uint8_t);
  void (*app_on_mitm_passcode_req) (const uint8_t);
  void (*app_on_encrypt_req_ind) (uint8_t const, struct gapc_encrypt_req_ind const *);
};
```

8.2.7 Default Handlers

Simple BLE peripheral applications to a large extent share common functionality. They should advertise and respond to specific requests from the central device to establish a connection. To minimize the amount of code that is required from an application programmer to start an application, a library of Default Handler functions has been created. These functions are already hooked from the template project in the user callbacks, giving a working peripheral device without writing a single line of code. The programmer can then alter the functionality, overriding the default functionality.

The default handlers have their own configuration options and function hooks as well. The user can configure the advertise operation scenario or the security request scenario of the peripheral as implemented by the default handlers, in the `user_default_hnd_conf file`. He can also override the advertise operation totally in the `user_default_app_operations` structure.

9 Memory Organization

9.1 Overview

The DA1458x contains an embedded One Time Programmable (OTP) memory for storing Bluetooth profiles as well as custom application code. The qualified Bluetooth® Low Energy protocol stack is stored in a dedicated ROM. Low leakage Retention RAM is used to store sensitive data and connection information while in Deep Sleep mode. The memory block sizes are as follows:

- 84 kB ROM. Contains Boot ROM code and Bluetooth Low Energy protocol related code.
- 32 kB One-Time-Programmable (OTP). At power up or reset of the DA1458x, the primary boot code (ROM code) checks if the OTP memory is programmed and if it is, it proceeds with mirroring the OTP contents to System RAM and it programs execution.
- 128 kB Flash (DA14583 only). At power up or reset of the DA14583, the primary boot code (ROM code) loads the secondary Bootloader (from OTP memory or FLASH) and the Secondary Bootloader proceeds with copying the FLASH image to System RAM and it programs execution.
- 42 kB System SRAM.
- 8 kB Retention SRAM.

9.2 Memory Map

The BLE Core requires access to a memory space named “Exchange Memory” to store control structures and frame buffers. The mapping of the BLE Core address space to the System Bus address space is controlled via the register field GP_CONTROL_REG[EM_MAPPING] (see Ref. [1]). In the SDK application examples, Case 23 is selected and the programming of the register is in the file `sdk\platform\arch\boot\rvds\system_ARMCM0.c`:

```
SetBits32(GP_CONTROL_REG, EM_MAP, 23);
```

Note that the memory mapping choice needs to be passed in to the Keil (J-Link) debugger via the initialization file `sdk\common_project_files\misc\sysram_case23.ini` so that the debugger can use the correct memory mapping. In this file the following line provides this information:

```
E long 0x50003308 = 0x2e
```

Figure 12 illustrates the address mapping for Case 23 which is used in all example applications of the SDK. See Appendix A.

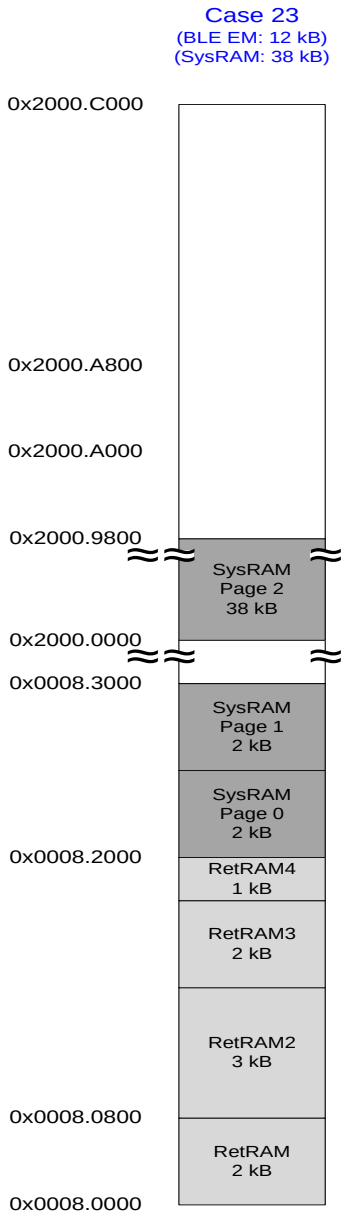


Figure 12: Case 23 RAM Memory Map

As shown in Figure 12, areas 0x0 to 0x7FFFF and 0x83000 to 0x1FFFFFFF are reserved and cannot be used. In 0x80000 to 0x81FFF the Retention RAM is located. Above it, at 0x82000 to 0x82FFF there is 4 kB of RAM space that is not retained when the chip goes into Deep Sleep mode.

From 0x20000000 to 0x200097FF there is 38 kB of RAM space (SysRAM). Some areas in this memory space are reserved and cannot be used by the user. These are:

- Vector table, placed at 0x20000000.
- Jump table, placed at 0x20000160.
- Timeout table, placed at 0x200002C0.
- NVDS storage, placed at 0x20000340.
- ROM code data, placed at 0x20009000.

In the remaining memory space (~37 kB) the application's code and data can be placed. Note that the size of the code and the initialized data is limited by the OTP size of 32 kB. There is also space left in the area 0x80000 to 0x82FFF for data storage. For more details refer to [19].

9.3 ARM Scatter File

The scatter file instructs the Linker where to place code and data. It is comprised of Load Region descriptors and Execution Region descriptors.

A Load Region instructs the linker where to place code and data and the initialization code for loading the code and data. Before the code reaches `main_func()` in `arch_main.c`, the initialization code is executed. During initialization code and data will be copied, if necessary, from the Load Regions to the Execution Regions. When for example a Flash memory is used to store code and data, this architecture allows for the seamless transfer of data and code areas to the system RAM.

For DA1458x this functionality is not required since copying from OTP, during system power-up or when exiting Deep Sleep mode, is done by the BootROM code and this procedure is completely transparent to the application code. Therefore the Load Regions in the DA1458x's scatter files match their respective Execution Regions, at least for the memory range from 0x20000000 to 0x200097FF.

The rest of the Load Regions should contain only zero initialized data. The area 0x80000 to 0x82FFF should be declared as containing data that do not have to be initialized (UNINIT). Since this area is zeroed when the chip boots, zero initialized data can be placed in here. Data compression must be disabled for all regions.

Scatter files for the example applications can be found at `sdk\common_project_files\scatterfile`. For a detailed explanation of the Scatter file and the various sections of the code and how the memory map is changing depending on the sleep mode, refer to [19].

10 Peripheral Drivers

10.1 Overview

The DA1458x Bluetooth® Low Energy SoC Platform supports several peripherals on different interfaces. To support them the DA1458x Software Platform provides the following driver architecture.

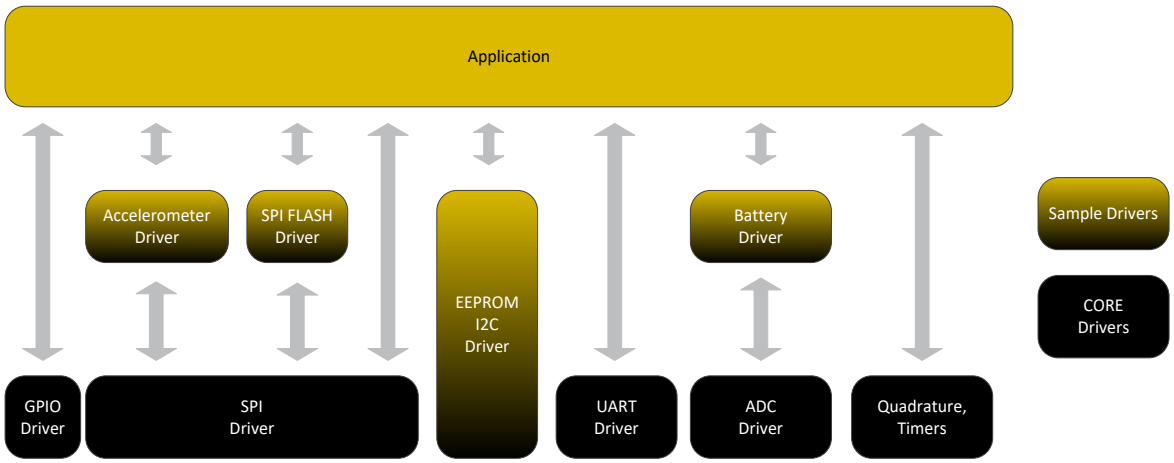


Figure 13: Peripherals Driver Architecture

The DA1458x SDK comes with a core driver for each interface (GPIO, SPI, UART, ADC, Quadrature, and Timers) together with several sample drivers (Accelerometer, SPI Flash, EEPROM I2C, Battery Level). Note that even though the source code for all drivers is provided in the SDK to aid debugging, modifying the core drivers is not recommended.

Note: Upon system wakeup from Extended Sleep or Deep Sleep mode, the device initialization and configuration functions have to be called again. The dedicated location to implement these calls is the `periph_init()` function in `periph_setup.c`.

The following sections first describe the core driver for each interface (GPIO, SPI, UART, ADC, Quadrature and Timers) and then the sample drivers, listing the various functions that support them.

10.2 UART

This driver is used to provide the necessary abstraction to the applications when access to the UART is required. It is used in the provided example Bluetooth Low Energy applications. You can use the various functions of the UART driver library's API as is, or compile additional layers to wrap the provided functionality.

The source code for this driver is located in: `sdk\platform\driver\uart`.

10.2.1 How to Use this Driver

- Enable the UART peripheral clock, setting the appropriate bit in `CLK_PER_REG`.
- Initialize the UART, using `uart_init()`.
- Set the RTS signal to Active (LOW), using `uart_flow_on()`.
- Set the RTS signal to Inactive (HIGH), using `uart_flow_off()`.
- Wait until all transfers are finished, using `uart_finish_transfers()`.
- Read from the UART, using `uart_read()`.
- Write to the UART, using `uart_write()`.

> **CAUTION**
>
> Do not call any UART functions to initiate UART transactions until after the system has booted up completely (system execution has reached the main loop). The XTAL16M clock has to be stabilized before any UART transaction takes place, either after power-up or after wake-up from sleep.

10.2.2 Initialization and Configuration

- `uart_init()`
- `uart_flow_on()`
- `uart_flow_off()`
- `uart_finish_transfers()`
- `uart_read()`
- `uart_write()`

10.2.3 Function Reference

The following sections provide the function reference for the UART driver.

10.2.3.1 uart_init

Function name	**void uart_init(uint8_t baud_rate, uint8_t mode)**
Function description	Initializes the UART to default values.
Parameters	baud_rate UART_BAUDRATE_115K2 mode Bit 7: Set always to zero. Bit 6: UART Break Control Bit. Setting this bit to 1, causes a break condition to be transmitted to the receiving device: the serial output is forced to the spacing (logic 0) state. Bit 5: Reserved (must be 0). Bit 4: Even Parity Select. This is used to select between even and odd parity, when parity is enabled (Parity Enable = 1). If set to 1, an even number of logic 1s is transmitted or checked. If set to 0, an odd number of logic 1s is transmitted or checked. Bit 3: Parity Enable. This bit is used to enable and disable parity generation and detection in transmitted and received serial character respectively: 0: parity disabled 1: parity enabled Bit 2: Number of Stop Bits. This is used to select the number of stop bits per character that the peripheral transmits and receives. 0: 1 stop bit 1: 1.5 stop bits when DLS (LCR[1:0]) is zero, else 2 stop bits Bits 1, 0: Data Length Select. This is used to select the number of data bits per character that the peripheral transmits and receives: 00: 5 bits 01: 6 bits 10: 7 bits 11: 8 bits Example. mode = 0x3: {no parity, 1 stop bit, 8 bits data length} settings are applied.
Return values	None
Notes	

10.2.3.2 uart_flow_on

Function name	**void uart_flow_on(void)**
Function description	Enables the UART RTS signal (active LOW).
Parameters	None
Return values	None
Notes	The RTS pad, if configured, is set to active (LOW). Please, note that with Auto Flow Control Enabled, the RTS signal is also gated with the receiver FIFO threshold trigger (RTS is inactive-high when above the threshold).

10.2.3.3 uart_flow_off

Function name	**`void uart_flow_off(void)`**
Function description	Disables the UART RTS signal (active LOW).
Parameters	None
Return values	None
Notes	The RTS pad, if configured, will be driven LOW (active).

10.2.3.4 uart_finish_transfers

Function name	**`void uart_finish_transfers(void)`**
Function description	Waits until all UART transfers have finished.
Parameters	None
Return values	None
Notes	Waits while any of the bits Transmitter Empty and Transmit Holding Register Empty of the UART_LSR_REG register is set.

10.2.3.5 uart_write

Function name	**`void uart_write(uint8_t *bufptr, uint32_t size, void (*callback) (uint8_t))`**
Function description	Writes one or more bytes of data to the UART.
Parameters	`bufptr` **Pointer to the buffer.** `size` **Count of bytes to send.** `callback` **Set to NULL.** Example: `uart_write( buffer1[], 12, NULL)` will **send 12 bytes from buffer1 to the UART.**
Return values	None
Notes	

10.2.3.6 uart_read

Function name	**`void uart_read(uint8_t *bufptr, uint32_t size, void (*callback) (uint8_t))`**
Function description	Reads one or more bytes of data from the UART.
Parameters	`bufptr` Pointer to the buffer. `size` Count of bytes to read. `callback` Set to NULL. Example: `uart_write( buffer1[], 12, NULL)` will read 12 bytes from the UART to buffer1.
Return values	None
Notes	

10.2.4 Definitions

```
#define UART_BAUDRATE_115K2         9
/// Baudrate used on the UART
#ifndef CFG_ROM
      #define UART_BAUDRATE  UART_BAUDRATE_115K2
#else //CFG_ROM
      #define UART_BAUDRATE  UART_BAUDRATE_460K8
#endif //CFG_ROM
#if (UART_BAUDRATE == UART_BAUDRATE_921K6)
      #define UART_CHAR_DURATION        11
#else
#define UART_CHAR_DURATION       (UART_BAUDRATE * 22)
#endif // (UART_BAUDRATE == UART_BAUDRATE_921K6)

/// Generic enable/disable enum for UART driver
enum
{
    /// uart disable
    UART_DISABLE = 0,
    /// uart enable
    UART_ENABLE  = 1
};

/// Character format
enum
{
    /// char format 5
    UART_CHARFORMAT_5 = 0,
    /// char format 6
    UART_CHARFORMAT_6 = 1,
    /// char format 7
    UART_CHARFORMAT_7 = 2,
    /// char format 8
    UART_CHARFORMAT_8 = 3
};

/// Stop bit
enum
{
    /// stop bit 1
    UART_STOPBITS_1 = 0,
    /* Note: The number of stop bits is 1.5 if a
     * character format with 5 bit is chosen */
    /// stop bit 2
    UART_STOPBITS_2 = 1
};

/// Parity bit
enum
{
    /// even parity
    UART_PARITYBIT_EVEN  = 0,
    /// odd parity
    UART_PARITYBIT_ODD   = 1,
    /// space parity
    UART_PARITYBIT_SPACE = 2, // The parity bit is always 0.
    /// mark parity
    UART_PARITYBIT_MARK  = 3  // The parity bit is always 1.
};
```



```
/// Error detection
enum
{
    /// error detection disabled
    UART_ERROR_DETECT_DISABLED = 0,
    /// error detection enabled
    UART_ERROR_DETECT_ENABLED  = 1
};

/// status values
enum
{
    /// status ok
    UART_STATUS_OK,
    /// status not ok
    UART_STATUS_ERROR
};
```

10.3 GPIO

The various functions of the GPIO driver library are described in this section. This driver is used to provide the necessary abstraction to the applications when access to the GPIOs is required. Furthermore, it guarantees that each GPIO is used only from one module (or place) at a time.

For the monitoring of the GPIO assignment, the assumption is made that this functionality is required only during the Development phase. Thus, any variables used for this purpose are not required in the final version that will be burned in the OTP and no valuable memory space will be consumed.

Based on the above, a 64-bit variable is used for the monitoring of the GPIO assignment. The first 16-bits of this variable are assigned to Port 0, the next 16-bits to Port 1 and so on. Each bit represents one GPIO pin of a port. This variable is placed in the retention memory and preserved during Deep Sleep mode.

Each module that needs to use a GPIO pin must first reserve it. The reservations are made inside the source file periph_setup.h (function GPIO_reservations()) using the macro RESERVE_GPIO(). This macro is defined in *gpio.h* as:

```
#define RESERVE_GPIO( name, port, pin, func )   /
     { GPIO[##port##][##pin##] = (GPIO[##port##][##pin##] != 0) ? (-1) : 1; };
```

The parameters name and func are used only to provide a readable declaration. This macro will set a member of the GPIO[] array (that corresponds to this GPIO pin) to 1 when it is free, or to -1 when it has already been reserved.

Upon initialization the function GPIO_init() (gpio.c) is called. This function will first check for multiple reservations of the same GPIO pin (halting at a breakpoint when one is found) and then set the 64-bit GPIO_status variable, according to the reservations that have been made in gpio_pindefs.h.

This GPIO_status variable will then be checked at the entry of any of the API functions to find out whether the GPIO, for which the function is being called, has been previously reserved. When it was not reserved, a breakpoint will be asserted.

The functionality described so far is available only when the DEVELOPMENT_DEBUG flag is set to 1. Since breakpoints result in Hard Fault interrupts when no debugger is attached, the code snippet MUST be left out in the final version by setting the flag DEVELOPMENT_DEBUG to 0.

Of course, direct access to GPIOs without using this API must be avoided. There is no way to prevent such a coding approach, but one should keep in mind that any visibility offered by this driver will be lost and there will be no guarantee that the same GPIO is not used in multiple places and, possibly, for not the same purpose.

The source code for this driver is located in: sdk\platform\driver\gpio.

10.3.1 How to Use this Driver

Typical Use

- Populate function `GPIO_reservations()` in `periph_setup.h`: Add a `RESERVE_GPIO()` macro instruction with the proper arguments, for each GPIO pin you wish to use.
- Populate function `set_pad_functions()` in `periph_setup.h`: Add a call to function `GPIO_ConfigurePin()` with the proper arguments, for each GPIO pin you wish to use. After verifying that the pin has been previously reserved, the desired functionality and direction/electrical configuration is set up.
- Set the logic state of a properly configured pin to HIGH, using `GPIO_SetActive()`.
- Set the logic state of a properly configured pin to LOW, using `GPIO_SetInactive()`.
- Get the logic state of a properly configured pin, using `GPIO_GetPinStatus()`.
- Configure any of the five available `GPIOx_IRQ` to trigger by a logic level or edge on a selectable GPIO, using `GPIO_EnableIRQ()`.
- Reset a `GPIOx_IRQ` interrupt, using `GPIO_ResetIRQ()`.

- Register a custom callback function to be called when the interrupt is triggered, using `GPIO_RegisterCallback()`.

Other Functionality

- Initialize the GPIO driver, using `GPIO_init()`. This function is called during system startup.
- Set the desired direction of a pin (input/output), its electrical configuration (pull-up/pull-down/high-z) and its functionality (GPIO/various peripherals' pin), using `GPIO_SetPinFunction()`. This function is called from `GPIO_ConfigurePin()`.
- Select the power rail from which a pin is powered, using `GPIO_ConfigurePinPower()`.

DA14583 Specific Functionality

When the `__DA14583__` preprocessor flag is defined the GPIO driver initialization function `GPIO_init()` shall automatically reserve the pins which are assigned to the internal SPI Flash memory (provided that the `DEVELOPMENT_DEBUG` flag is also defined to 1). However the GPIO driver will not configure these pins automatically and it is the application's responsibility to configure them. An application will typically add the following code in function set_pad_functions() in order to configure the pins assigned to the internal SPI Flash memory:

```
GPIO_ConfigurePin(GPIO_PORT_2, GPIO_PIN_3, OUTPUT, PID_SPI_EN,  true );
GPIO_ConfigurePin(GPIO_PORT_2, GPIO_PIN_0, OUTPUT, PID_SPI_CLK, false );
GPIO_ConfigurePin(GPIO_PORT_2, GPIO_PIN_9, OUTPUT, PID_SPI_DO,  false );
GPIO_ConfigurePin(GPIO_PORT_2, GPIO_PIN_4, INPUT,  PID_SPI_DI,  false );
```

10.3.2 Initialization and Configuration

- `GPIO_init()`
- `GPIO_SetPinFunction()`
- `GPIO_ConfigurePin()`
- `GPIO_SetActive()`
- `GPIO_SetInactive()`
- `GPIO_GetPinStatus()`
- `GPIO_ConfigurePinPower()`

10.3.3 Interrupt Handling

- `GPIO_EnableIRQ()`
- `GPIO_ResetIRQ()`
- `GPIO_RegisterCallback()`

10.3.4 Function Reference: Initialization and Configuration Functions

10.3.4.1 GPIO_init

Function name	**void GPIO_init(void)**
Function description	Checks for multiple reservations of the same GPIO pin. Initializes the GPIO_status variable. Called at system start-up.
Parameters	None
Return values	None (a breakpoint is triggered in case of duplicate assignment of a pin)
Notes	Active only during development (DEVELOPMENT_DEBUG = 1). Deactivate for release, to preserve memory space.

10.3.4.2 GPIO_SetPinFunction

Function name	**void GPIO_SetPinFunction(int port, int pin, GPIO_PUPD mode, GPIO_FUNCTION function)**
Function description	Sets the pin type (input, input pull-up or pull-down, output) and the pin function (GPIO, UART1_RX, etc.).
Parameters	port The GPIO port (GPIO_PORT_n). pin The GPIO pin (GPIO_PIN_n). mode The GPIO pin direction/electrical configuration: INPUT: input high-z INPUT_PULLUP: input with pull-up resistor enabled INPUT_PULLDOWN: input with pull-down resistor enabled OUTPUT: output function The function of the pin (assignment to internal peripherals): PID_GPIO, PID_UART1_RX, PID_UART1_TX, PID_UART2_RX, PID_UART2_TX, PID_SPI_DI, PID_SPI_DO, PID_SPI_CLK, PID_SPI_EN, PID_I2C_SCL, PID_I2C_SDA, PID_UART1_IRDA_RX, PID_UART1_IRDA_TX, PID_UART2_IRDA_RX, PID_UART2_IRDA_TX, PID_ADC, PID_PWM0, PID_PWM1, PID_BLE_DIAG, PID_UART1_CTSN, PID_UART1_RTSN, PID_UART2_CTSN, PID_UART2_RTSN, PID_PWM2, PID_PWM3, PID_PWM4
Return values	None
Notes	PID_ADC is available only on P0_0 to P0_3.

10.3.4.3 GPIO_ConfigurePin

Function name	**`void GPIO_ConfigurePin( int port, int pin, GPIO_PUPD mode, GPIO_FUNCTION function, const bool high )`**
Function description	Combined function to set the pin type and function (like `GPIO_SetPinFunction()` does) and the state of the pin. Can be used for output pins to first set them to the desired state and then configure them as outputs.
Parameters	`port` The GPIO port (GPIO_PORT_n). `pin` The GPIO pin (GPIO_PIN_n). `mode` The GPIO pin direction/electrical configuration: `INPUT: input high-z` `INPUT_PULLUP: input with pull-up resistor enabled` `INPUT_PULLDOWN: input with pull-down resistor enabled` `OUTPUT : output` `function` The function of the pin (assignment to internal peripherals): `PID_GPIO, PID_UART1_RX, PID_UART1_TX, PID_UART2_RX, PID_UART2_TX, PID_SPI_DI, PID_SPI_DO, PID_SPI_CLK, PID_SPI_EN, PID_I2C_SCL, PID_I2C_SDA, PID_UART1_IRDA_RX, PID_UART1_IRDA_TX, PID_UART2_IRDA_RX, PID_UART2_IRDA_TX, PID_ADC, PID_PWM0, PID_PWM1, PID_BLE_DIAG, PID_UART1_CTSN, PID_UART1_RTSN, PID_UART2_CTSN, PID_UART2_RTSN, PID_PWM2, PID_PWM3, PID_PWM4` `high` The desired logic level of the pin.
Return values	None
Notes	

10.3.4.4 GPIO_SetActive

Function name	**`void GPIO_SetActive( int port, int pin )`**
Function description	Sets the GPIO as logic HIGH.
Parameters	`port` The GPIO port (GPIO_PORT_n). `pin` The GPIO pin (GPIO_PIN_n).
Return values	None
Notes	The GPIO must have been previously configured as output. No check of the configuration of the pin is done in this function.

10.3.4.5 GPIO_SetInactive

Function name	**`void GPIO_SetInactive( int port, int pin )`**
Function description	Sets the GPIO as logic LOW.
Parameters	`port` The GPIO port (GPIO_PORT_n). `pin` The GPIO pin (GPIO_PIN_n).
Return values	None
Notes	The GPIO must have been previously configured as output. No check of the configuration of the pin is done in this function.

10.3.4.6 GPIO_GetPinStatus

Function name	**`bool GPIO_GetPinStatus( int port, int pin )`**
Function description	Gets the logic state of this GPIO pin.
Parameters	`port` The GPIO port (GPIO_PORT_n). `pin` The GPIO pin (GPIO_PIN_n).
Return values	TRUE when the pin is logic HIGH, else FALSE.
Notes	The GPIO must have been previously configured as input. No check of the configuration of the pin is done in this function.

10.3.4.7 GPIO_ConfigurePinPower

Function name	**`void GPIO_ConfigurePinPower( GPIO_PORT port, GPIO_PIN pin, GPIO_POWER_RAIL power_rail )`**
Function description	Selects the power rail from which a pin is powered.
Parameters	`port` The GPIO port (GPIO_PORT_n). `pin` The GPIO pin (GPIO_PIN_n). `power_rail` The desired power rail: (GPIO_POWER_RAIL_3V, GPIO_POWER_RAIL_1V)
Return values	None
Notes	

10.3.5 Function Reference: Interrupt Handling Functions

10.3.5.1 GPIO_EnableIRQ

Function name	**`void GPIO_EnableIRQ( GPIO_PORT port, GPIO_PIN pin, IRQn_Type irq, bool low_input, bool release_wait, uint8_t debounce_ms )`**
Function description	Configures the GPIO interrupt generator to trigger by a logic level or edge on one of the selectable GPIOs.
Parameters	`port` The GPIO port (GPIO_PORT_n). `pin` The GPIO pin (GPIO_PIN_n). `irq` The GPIO IRQ to configure (`IRQn_Type`). `low_input` TRUE: generates an IRQ when the input is logic LOW FALSE: generates an IRQ when the input is logic HIGH. `release_wait` TRUE: waits for key release after interrupt was reset (edge). `debounce_ms` Duration of a de-bounce phase before the IRQ is generated.
Return values	None
Notes	

10.3.5.2 GPIO_ResetIRQ

Function name	**`void GPIO_ResetIRQ( IRQn_Type irq )`**
Function description	Resets a GPIOn_IRQ interrupt.
Parameters	`irq` The GPIO IRQ to reset (`IRQn_Type`).
Return values	None
Notes	

10.3.5.3 GPIO_RegisterCallback

Function name	**void GPIO_RegisterCallback(IRQn_Type irq, GPIO_handler_function_t callback)**
Function description	Registers a custom callback function to be called when the interrupt is triggered.
Parameters	irq — The GPIO IRQ to configure (IRQn_Type). callback — The custom callback function to be called when the interrupt is triggered (GPIO_handler_function_t).
Return values	None

Notes	The GPIOn_IRQ is cleared before entering the callback function, by the driver's common GPIOn handler (GPIOn_Handler).
	Sample callback function:
	`void my_callback_function(void)`
	`{`
	`  // user code here`
	`}`

10.3.6 Definitions

```
typedef enum {
    INPUT = 0,
    INPUT_PULLUP = 0x100,
    INPUT_PULLDOWN = 0x200,
    OUTPUT = 0x300,
} GPIO_PUPD;

typedef enum {
    GPIO_PORT_0 = 0,
    GPIO_PORT_1 = 1,
    GPIO_PORT_2 = 2,
    GPIO_PORT_3 = 3,
    GPIO_PORT_3_REMAP = 4,
} GPIO_PORT;

typedef enum {
    GPIO_PIN_0 = 0,
    GPIO_PIN_1 = 1,
    GPIO_PIN_2 = 2,
    GPIO_PIN_3 = 3,
    GPIO_PIN_4 = 4,
    GPIO_PIN_5 = 5,
    GPIO_PIN_6 = 6,
    GPIO_PIN_7 = 7,
    GPIO_PIN_8 = 8,
    GPIO_PIN_9 = 9,
} GPIO_PIN;

typedef enum {
    PID_GPIO = 0,
    PID_UART1_RX,
    PID_UART1_TX,
    PID_UART2_RX,
    PI_UART2_TX,
    PID_SPI_DI,
    PID_SPI_DO,
    PID_SPI_CLK,
    PID_SPI_EN,
```



```
    PID_I2C_SCL,
    PID_I2C_SDA,
    PID_UART1_IRDA_RX,
    PID_UART1_IRDA_TX,
    PID_UART2_IRDA_RX,
    PID_UART2_IRDA_TX,
    PID_ADC,
    PID_PWM0,
    PID_PWM1,
    PID_BLE_DIAG,
    PID_UART1_CTSN,
    PID_UART1_RTSN,
    PID_UART2_CTSN,
    PID_UART2_RTSN,
    PID_PWM2,
    PID_PWM3,
    PID_PWM4,
} GPIO_FUNCTION;

//
// Macro for pin definition structure
//      name: usage and/or module using it
//      func: GPIO, UART1_RX, UART1_TX, etc.
//
#if DEVELOPMENT_DEBUG
#define RESERVE_GPIO( name, port, pin, func )   { GPIO[##port##][##pin##] =
(GPIO[##port##][##pin##] != 0) ? (-1) : 1;GPIO_status |=
((uint64_t)GPIO[##port##][##pin##] << ##pin##) << (##port## * 16);}
#else
#define RESERVE_GPIO( name, port, pin, func )   {}
#endif
```


10.4 Analog to Digital Converter

The following section lists the various functions of the ADC driver library. The source code for this driver is located in: `sdk\platform\driver\adc`.

10.4.1 How to Use this Ddriver

- Calibrate the ADC module using `adc_calibrate()`.
- Enable the ADC module and configure it, using `adc_init()`.
- Enable the desired ADC channel, using `adc_enable_channel()`.
- Get a sample from the enabled ADC channel, using `adc_get_sample()`.
- Get a sample from one of the battery ADC channels, using `adc_get_vbat_sample()`.
- Upon completion, if desired, disable the ADC, using `adc_disable()`.

10.4.2 Initialization and Configuration

- `adc_calibrate()`
- `adc_init()`
- `adc_enable_channel()`
- `adc_get_sample()`
- `adc_disable()`

10.4.3 Function Reference: Initialization and Configuration Functions

10.4.3.1 adc_calibrate

Function name	**`void adc_calibrate(void)`**
Function description	Calibrates the ADC module and stores the calibration offsets in registers GP_ADC_OFFP_REG and GP_ADC_OFFN_REG. When the system uses Sleep mode, this function always has to be called before using the ADC, since the calibration registers are not retained. When the system is not using Sleep mode this function can be called once at system start-up.
Parameters	None
Return values	None
Notes	

10.4.3.2 adc_init

Function name	**`void adc_init (uint16_t mode, uint16_t sign, uint16_t attn)`**
Function description	Initializes the ADC peripheral according to the parameters.

Parameters	`mode`
	`0: differential mode`
	`GP_ADC_SE (0x800): single ended mode`
	`sign`
	`0: Default`
	`GP_ADC_SIGN (0x400): conversion with opposite sign at input and output to cancel internal offset of the ADC and low frequencies`
	`attn`
	`0: No attenuation`
	`GP_ADC_ATTN3X: 3x attenuation`

Return values	None
Notes	

10.4.3.3 adc_enable_channel

Function name	**`void adc_enable_channel (uint16_t input_selection)`**
Function description	Enables the ADC channel specified in the parameter.
Parameters	`input_selection` Inpt channel. Must pass one of the definitions starting with ADC_CHANNEL_ in adc.h. See section 10.4.5.
Return values	None
Notes	The device must have been initialized, using `adc_init()`.

10.4.3.4 adc_disable

Function name	**`void adc_disable(void)`**
Function description	Disables the ADC module.
Parameters	None
Return values	None
Notes	

10.4.4 Function Reference: ADC Sampling Functions

10.4.4.1 adc_get_sample

Function name	**int adc_get_sample(void)**
Function description	Reads an ADC sample. The ADC must have been initialized using `adc_init()` and a valid channel must have been set using `adc_enable_channel()`.
Parameters	None
Return values	None
Notes	The developer is responsible for handling this value with respect to the attenuation (`attn`) selected in `adc_init()`.

10.4.4.2 adc_get_vbat_sample

Function name	**int adc_get_vbat_sample(bool sample_vbat1v)**
Function description	Reads an ADC sample of the battery voltage from ADC_CHANNEL_VBAT1V or ADC_CHANNEL_VBAT3V. The ADC must have been initialized using `adc_init()`.

Parameters	`sample_vbat1v`
	`TRUE: use channel ADC_CHANNEL_VBAT1V`
	`FALSE: use channel ADC_CHANNEL_VBAT3V`

Return values	The ADC sample for the selected battery channel.
Notes	

10.4.5 Definitions

ADC_channels

```
#define ADC_CHANNEL_P00         0
#define ADC_CHANNEL_P01         1
#define ADC_CHANNEL_P02         2
#define ADC_CHANNEL_P03         3
#define ADC_CHANNEL_AVS         4
#define ADC_CHANNEL_VDD_REF     5
#define ADC_CHANNEL_VDD_RTT     6
#define ADC_CHANNEL_VBAT3V      7
#define ADC_CHANNEL_VDCDC       8
#define ADC_CHANNEL_VBAT1V      9
```


10.5 Serial Peripheral Interface (SPI) driver

The following section lists the various functions of the SPI driver library that handle the initialization, configuration and release of the SPI module, the control of the chip select (CS) line and the data transfer over the SPI.

The source code for this driver is located in: `sdk\platform\driver\spi`.

10.5.1 How to Use this Driver

- Enable the SPI block and configure its parameters, using `spi_init()`.
- Activate the Chip Select line, using `spi_set_cs_low()`.
- Make a sequence of SPI transfers (send and receive), using `spi_access()` for each transfer.
- Select the desired SPI bit mode using `setSpiBitmode()`.
- Deactivate the Chip Select line, using `spi_set_cs_high()`.
- For a simple SPI transaction (1 read-write cycle for the selected bit mode) you can use `spi_transaction()`. This function includes driving the Chip Select line.
- Disable the SPI module, using `spi_release()`.

10.5.2 Initialization and Configuration

- `spi_init()`
- `spi_release()`
- `setSpiBitmode()`
- `spi_set_cs_low()`
- `spi_set_cs_high()`

10.5.3 Function Reference: Initialization and Configuration Functions

10.5.3.1 spi_init

Function name	**`void spi_init(SPI_Pad_t *cs_pad_param, SPI_Word_Mode_t bitmode, SPI_Role_t role, SPI_Polarity_Mode_t clk_pol, SPI_PHA_Mode_t pha_mode, SPI_MINT_Mode_t irq, SPI_XTAL_Freq_t freq)`**
Function description	Initializes the SPI block and configures the driver according to the parameters. First the SPI module is disabled, then the status register is updated with the selected parameters and finally the module is enabled again. When the SPI block is disabled, the RX/TX buffers are reset.

Parameters	`cs_pad_param` Port and pin of the Chip Select (/CS) pad for the target SPI slave.
	`bitmode` Selects the transfer word length.
	`SPI_MODE_8BIT: 8-bit mode`
	`SPI_MODE_16BIT: 16-bit mode`
	`SPI_MODE_32BIT: 32-bit mode`
	`SPI_MODE_9BIT: 9-bit mode`
	`role` Selects the master/slave role.
	`SPI_ROLE_MASTER: Master mode`
	`SPI_ROLE_SLAVE: Slave mode`
	`clk_pol` Clock phase.
	`SPI_CLK_IDLE_POL_LOW: SPI_CLK is initially LOW`
	`SPI_CLK_IDLE_POL_HIGH: SPI_CLK is initially HIGH`
	`pha_mode` Clock polarity.
	`SPI_PHA_MODE_0`
	`SPI_PHA_MODE_1`
	Note: When the clock phase equals the clock polarity (`pha_mode = clk_pol`), data is captured on the clock's rising edge, otherwise it is captured on the clock's falling edge.
	`irq` Interrupt request enable/disable.
	`SPI_MINT_DISABLE: Disable SPI interrupt (SPI_INT_BIT) to the ICU`
	`SPI_MINT_ENABLE: Enable SPI interrupt (SPI_INT_BIT) to the ICU`
	`freq` Select the SPI_CLK clock frequency in master mode.
	`SPI_XTAL_DIV_8 = (XTAL)/ (CLK_PER_REG * 8)`
	`SPI_XTAL_DIV_4 = (XTAL) / (CLK_PER_REG * 4)`
	`SPI_XTAL_DIV_2 = (XTAL) / (CLK_PER_REG * 2)`
	`SPI_XTAL_DIV_14 = (XTAL) / (CLK_PER_REG * 14)`

Return values	None
Notes	

10.5.3.2 SPI modes

SPI mode	Clock polarity (clk_pol)	Clock phase (pha_mode)	Clock edge for data sampling
0	SPI_CLK_IDLE_POL_LOW	SPI_PHA_MODE_0	rising edge
1	SPI_CLK_IDLE_POL_LOW	SPI_PHA_MODE_1	falling edge
2	SPI_CLK_IDLE_POL_HIGH	SPI_PHA_MODE_0	falling edge
3	SPI_CLK_IDLE_POL_HIGH	SPI_PHA_MODE_1	rising edge

10.5.3.3 setSpiBitmode

Function name	**void setSpiBitmode (SPI_Word_Mode_t spiBitMode)**
Function description	Selects the SPI word length. First the SPI module is disabled, then the SPI control register (SPI_CTRL_REG) is updated to set the selected bit mode and finally the module is enabled again. When the SPI block is disabled, the RX/TX buffers are reset.
Parameters	`spiBitMode` Selects the word length. `SPI_MODE_8BIT` = 8-bit mode `SPI_MODE_16BIT` = 16-bit mode `SPI_MODE_32BIT` = 32-bit mode `SPI_MODE_9BIT` = 9-bit mode
Return values	None
Notes	

10.5.3.4 spi_release

Function name	**void spi_release(void)**
Function description	Disables the SPI block by resetting the SPI_ON bit of the SPI Control Register (SPI_CTRL_REG) and resetting the SPI_ENABLE bit of the Peripheral divider register (CLK_PER_REG).
Parameters	None
Return values	None
Notes	

10.5.4 Function Reference: Sending and Receiving Functions

10.5.4.1 spi_access

Function name	**uint32_t spi_access(uint32_t dataToSend)**
Function description	Writes `dataToSend` to the SPI and reads the received data value. Prior to a transfer, the SPI module has to be initialized using `spi_init()`. The function first extracts the selected word length for the current SPI configuration. Then it writes the SPI Rx/Tx register(s) (SPI_RX_TX_REG0 and also SPI_RX_TX_REG1 in case of 32-bit and 9-bit modes). Next, the function polls the SPI Control Register, waiting for the transfer to complete. Upon completion of the transfer (SPI Control Register: SPI_INT_BIT = 1), the function reads the received data from the SPI Rx/Tx register(s), clears the interrupt bit and returns the received data.
Parameters	`dataToSend` Data to be written.
Return values	Received data.
Notes	The function reads the value of the status register to determine the word length (8/16/32/9 bits) of the configuration. The state of the /CS line is not altered by this function, so it can be called multiple times in conjunction with `spi_cs_low()` and `spi_cs_high()` to form a complex SPI transaction. For a simple complete SPI transaction, see section 10.5.4.2.

10.5.4.2 spi_transaction

Function name	**`uint32_t spi_transaction(uint32_t dataToSend)`**
Function description	Writes `dataToSend` to the SPI in a simple complete transaction and reads the received data value. Prior to a transaction, the SPI module has to be initialized using `spi_init()`. The function first sets the /CS line to LOW (active), then calls spi_access to perform the data transfer and finally sets the /CS line to HIGH (inactive).
Parameters	`dataToSend` Data to be written.
Return values	Received data
Notes	See section 10.5.4.1 (`spi_access`).

10.5.4.3 spi_cs_low

Function name	**`inline void spi_cs_low(void)`**
Function description	Sets the chip select line (/CS) to LOW (active). This signals the beginning of an SPI transaction. Prior to using this function, the SPI module has to be initialized using `spi_init()`.
Parameters	None
Return values	None
Notes	Uses the `spi_driver_cs_pad` structure which is initialized with the /CS port and pin numbers in the `spi_init()` function.

10.5.4.4 spi_cs_high

Function name	**`inline void spi_cs_high(void)`**
Function description	Sets the chip select line (/CS) to HIGH (inactive). This signals the end of an SPI transaction. Prior to using this function, the SPI module has to be initialized using `spi_init()`.
Parameters	None
Return values	None
Notes	Uses the `spi_driver_cs_pad` structure which is initialized with the /CS port and the pin numbers in the `spi_init()` function.

10.5.5 Definitions

```
SPI block configuration
typedef enum SPI_WORD_MODES{
      SPI_MODE_8BIT,
      SPI_MODE_16BIT,
      SPI_MODE_32BIT,
      SPI_MODE_9BIT,
}SPI_Word_Mode_t;

typedef enum SPI_ROLES{
      SPI_ROLE_MASTER,
      SPI_ROLE_SLAVE,
}SPI_Role_t;

typedef enum SPI_POL_MODES{
      SPI_CLK_IDLE_POL_LOW,
      SPI_CLK_IDLE_POL_HIGH,
}SPI_Polarity_Mode_t;

typedef enum SPI_PHA_MODES{
      SPI_PHA_MODE_0,
      SPI_PHA_MODE_1,
}SPI_PHA_Mode_t;

typedef enum SPI_MINT_MODES{
      SPI_MINT_DISABLE,
      SPI_MINT_ENABLE,
}SPI_MINT_Mode_t;

typedef enum SPI_FREQ_MODES{
  SPI_XTAL_DIV_8,
  SPI_XTAL_DIV_4,
      SPI_XTAL_DIV_2,
      SPI_XTAL_DIV_14,
}SPI_XTAL_Freq_t;

typedef struct
{
      GPIO_PORT port;
      GPIO_PIN pin;
}SPI_Pad_t;
```

10.6 Quadrature Decoder

The following section lists the various functions of the QUADRATURE DECODER driver library. These functions implement the various operations to support the interfacing to a rotary encoder of up to three axes (X, Y, Z) plus the initialization, configuration and release of the driver interface.

The source code for this driver is located in: `sdk\platform\driver\wkupct_quadec`.

10.6.1 How to Use this Driver

Important Notes:

1. When the wakeup timer, the quadrature decoder or both are used in the application, the preprocessor directives `WKUP_ENABLED` and/or `QUADEC_ENABLED` respectively must be defined in the application, to allow for the inclusion of essential parts of the code.
2. When upon reception of an interrupt from the wakeup timer or the quadrature decoder, the system wakes up from Sleep mode and should resume the functionality of the peripherals, the following lines must be included in the wakeup handler function(s) that were registered using `wkupct_register_callback()` and/or `quad_decoder_register_callback()`:

```
// Init System Power Domain blocks: GPIO, WD Timer, Sys Timer, etc.
// Power up and init Peripheral Power Domain blocks,
// and finally release the pad latches.
if(GetBits16(SYS_STAT_REG, PER_IS_DOWN))
        periph_init();
```

The QUADRATURE DECODER driver provides one function for the initialization and configuration of the quadrature decoder, `quad_decoder_init()`, and one to disable it, `quad_decoder_release()`.

For working with interrupts, the driver provides a function to register a callback function, `quad_decoder_register_callback()`, a function to enable the IRQ, `quad_decoder_enable_irq()` and a function to disable the IRQ, `quad_decoder_disable_irq()`.

10.6.1.1 Usage with Polling

- Enable and initialize the quadrature block using `quad_decoder_init()`.
- Poll quadrature decoder counter values using `quad_decoder_get_x_counter()`, `quad_decoder_get_y_counter()` and `quad_decoder_get_z_counter()`.
- Release the quadrature decoder driver using `quad_decoder_release()`.

10.6.1.2 Usage with Interrupts

- Register a callback function to be called from within the `WKUP_QUADEC_Handler`, when interrupt is sourced from the quadrature decoder, using `quad_decoder_register_callback()`.
- In the callback function, placed in the application code, the quadrature decoder counter values for x, y, z are passed as parameters for further processing.
- Set up and enable the interrupts for the quadrature decoder, using `quad_decoder_enable_irq()`.
- After the setup, optionally disable the quadrature decoder IRQ, using `disable_quadec_irq()`.
  Caution: The IRQ will be disabled only if it has not also been enabled by the Wakeup Timer.
- Release the quadrature decoder driver, using `quad_decoder_release()`.
  Note: This function also calls `disable_quadec_irq()`.

10.6.1.3 Initialization and Configuration

- `void quad_decoder_init()`
- `quad_decoder_register_callback()`
- `quad_decoder_release()`
- `quad_decoder_enable_irq()`

- quad_decoder_disable_irq()

10.6.1.4 Reading Quadrature Decoder Counters

- quad_decoder_get_x_counter()
- quad_decoder_get_y_counter()
- quad_decoder_get_z_counter()

10.6.2 Function Reference: Initialization and Configuration Functions

10.6.2.1 quad_decoder_init

Function name	**void quad_decoder_init(QUAD_DEC_INIT_PARAMS_t *quad_dec_init_params)**
Function description	Initializes the quadrature decoder according to the specified parameters.
Parameters	quad_dec_init_params Pointer to parameter structure. chx_port_sel Selection of port X pads (CHX_PORT_SEL_t struct). chy_port_sel Selection of port Y pads (CHY_PORT_SEL_t struct). chz_port_sel Selects port Z pads (CHZ_PORT_SEL_t struct). qdec_clockdiv The number of system clock cycles per which the decoding logic samples the data input on the channel lines.
Return values	None
Notes	

10.6.2.2 quad_decoder_release

Function name	**void quad_decoder_release(void)**
Function description	Disables the quadrature decoder. This function resets the pin assignment of the quadrature decoder to QUAD_DEC_CHXA_NONE_AND_CHXB_NONE, QUAD_DEC_CHYA_NONE_AND_CHYB_NONE and QUAD_DEC_CHZA_NONE_AND_CHZB_NONE. Finally, it sets bit QUAD_ENABLE of the CLK_PER_REG register to 0, to disable the quadrature decoder. Also, it calls the function disable_quadec_irq().
Parameters	None
Return values	None
Notes	

10.6.2.3 quad_decoder_register_callback

Function name	**void quad_decoder_register_callback(uint32_t* callback)**
Function description	Registers a callback function to be called by the interrupt handler of the WKUP_QUADEC_IRQn, when the interrupt source is the quadrature decoder.
Parameters	callback Pointer to the callback function.
Return values	None
Notes	The callback function must be of the following type: void my_quad_decoder_user_callback_function(int16_t qdec_xcnt_reg, int16_t qdec_ycnt_reg, int16_t qdec_zcnt_reg)

10.6.2.4 quad_decoder_enable_irq

Function name	**void quad_decoder_enable_irq(uint8_t event_count)**
Function description	Setup and enable the interrupts for the quadrature decoder. Any pending WKUP_QUADEC_IRQn interrupt is cleared, the count of quadrature decoder events to trigger an interrupt is set, the QD_IRQ_MASK is reset and the WKUP_QUADEC_IRQn is enabled.
Parameters	event_count The count of quadrature decoder events to trigger an interrupt.
Return values	None
Notes	

10.6.2.5 quad_decoder_disable_irq

Function name	**wkupct_quadec_error_t quad_decoder_disable_irq(void)**
Function description	Unregisters the quadrature controller from the use of the WKUP_QUADEC_IRQn interrupt requests and calls the wkupct_quad_disable_IRQ() function to disable the interrupts for the quadrature decoder and the wakeup timer, only when no registration from the wakeup timer is active.
Parameters	None
Return values	WKUPCT_QUADEC_ERR_OK: No error. WKUPCT_QUADEC_ERR_RESERVED: The wakeup timer was previously registered for WKUP_QUADEC_IRQn.
Notes	

10.6.3 Function Reference: Quadrature Decoder Counter Reading Functions

10.6.3.1 quad_decoder_get_x_counter

Function name	**inline int16_t quad_decoder_get_x_counter(void)**
Function description	Retrieves the current value of the QDEC_XCNT_REG register, which holds the counter for the X channel of the quadrature decoder.
Parameters	None
Return values	The current value of the QDEC_XCNT_REG register.
Notes	

10.6.3.2 quad_decoder_get_y_counter

Function name	`inline int16_t quad_decoder_get_y_counter(void)`
Function description	Retrieves the current value of the QDEC_YCNT_REG register, which holds the counter for the Y channel of the quadrature decoder.
Parameters	None
Return values	The current value of the QDEC_YCNT_REG register.
Notes	

10.6.3.3 quad_decoder_get_z_counter

Function name	`inline int16_t quad_decoder_get_z_counter(void)`
Function description	Retrieves the current value of the QDEC_ZCNT_REG register, which holds the counter for the Z channel of the quadrature decoder.
Parameters	None
Return values	The current value of the QDEC_ZCNT_REG register.
Notes	

10.6.4 Definitions

```
typedef void (*quad_encoder_handler_function_t)(int16_t qdec_xcnt_reg, int16_t
qdec_ycnt_reg, int16_t qdec_zcnt_reg);
typedef enum
{
    QUAD_DEC_CHXA_NONE_AND_CHXB_NONE = 0,
    QUAD_DEC_CHXA_P00_AND_CHXB_P01 = 1,
    QUAD_DEC_CHXA_P02_AND_CHXB_P03 = 2,
    QUAD_DEC_CHXA_P04_AND_CHXB_P05 = 3,
    QUAD_DEC_CHXA_P06_AND_CHXB_P07 = 4,
    QUAD_DEC_CHXA_P10_AND_CHXB_P11 = 5,
    QUAD_DEC_CHXA_P12_AND_CHXB_P13 = 6,
    QUAD_DEC_CHXA_P23_AND_CHXB_P24 = 7,
    QUAD_DEC_CHXA_P25_AND_CHXB_P26 = 8,
    QUAD_DEC_CHXA_P27_AND_CHXB_P28 = 9,
    QUAD_DEC_CHXA_P29_AND_CHXB_P20 = 10
}CHX_PORT_SEL_t;

typedef enum
{
    QUAD_DEC_CHYA_NONE_AND_CHYB_NONE = 0<<4,
    QUAD_DEC_CHYA_P00_AND_CHYB_P01 = 1<<4,
    QUAD_DEC_CHYA_P02_AND_CHYB_P03 = 2<<4,
    QUAD_DEC_CHYA_P04_AND_CHYB_P05 = 3<<4,
    QUAD_DEC_CHYA_P06_AND_CHYB_P07 = 4<<4,
    QUAD_DEC_CHYA_P10_AND_CHYB_P11 = 5<<4,
    QUAD_DEC_CHYA_P12_AND_CHYB_P13 = 6<<4,
    QUAD_DEC_CHYA_P23_AND_CHYB_P24 = 7<<4,
    QUAD_DEC_CHYA_P25_AND_CHYB_P26 = 8<<4,
    QUAD_DEC_CHYA_P27_AND_CHYB_P28 = 9<<4,
    QUAD_DEC_CHYA_P29_AND_CHYB_P20 = 10<<4
}CHY_PORT_SEL_t;
```

```
typedef enum
{
    QUAD_DEC_CHZA_NONE_AND_CHZB_NONE = 0<<8,
    QUAD_DEC_CHZA_P00_AND_CHZB_P01 = 1<<8,
    QUAD_DEC_CHZA_P02_AND_CHZB_P03 = 2<<8,
    QUAD_DEC_CHZA_P04_AND_CHZB_P05 = 3<<8,
    QUAD_DEC_CHZA_P06_AND_CHZB_P07 = 4<<8,
    QUAD_DEC_CHZA_P10_AND_CHZB_P11 = 5<<8,
    QUAD_DEC_CHZA_P12_AND_CHZB_P13 = 6<<8,
    QUAD_DEC_CHZA_P23_AND_CHZB_P24 = 7<<8,
    QUAD_DEC_CHZA_P25_AND_CHZB_P26 = 8<<8,
    QUAD_DEC_CHZA_P27_AND_CHZB_P28 = 9<<8,
    QUAD_DEC_CHZA_P29_AND_CHZB_P20 = 10<<8
}CHZ_PORT_SEL_t;

typedef struct
{
    CHX_PORT_SEL_t chx_port_sel;
    CHY_PORT_SEL_t chy_port_sel;
    CHZ_PORT_SEL_t chz_port_sel;
    uint16_t qdec_clockdiv;
    uint8_t qdec_events_count_to_trigger_interrupt;
}QUAD_DEC_INIT_PARAMS_t;

typedef enum
{
    WKUPCT_QUADEC_ERR_RESERVED = -1,
    WKUPCT_QUADEC_ERR_OK = 0,
} wkupct_quadec_error_t;

typedef enum
{
    RESERVATION_STATUS_FREE = 0,
    RESERVATION_STATUS_RESERVED,
} reservation_status_t;

typedef void (*quad_encoder_handler_function_t)(int16_t qdec_xcnt_reg, int16_t
qdec_ycnt_reg, int16_t qdec_zcnt_reg);
```

10.6.5 Defines in the Application for the QUADRATURE DECODER Driver

The following preprocessor directive must be defined in the application, in order to include the necessary parts of the code:

```
QUADEC_ENABLED
```

See section 10.6.1, important note 1).

10.7 Wake-Up Timer

The following sections list the various functions of the WAKEUP TIMER driver library. These functions support the configuration of the Wakeup Interrupt Controller (WIC).

The source code for this driver is located in: `sdk\platform\driver\wkupct_quadec`.

10.7.1 How to Use this Driver

Important Notes:

1. When the wakeup timer, the quadrature controller or both are used in an application, the preprocessor directives: WKUP_ENABLED and/or QUADEC_ENABLED respectively must be defined in the application, to allow for the inclusion of essential parts of the code.
2. When upon reception of an interrupt from the wakeup timer or the quadrature decoder, the system wakes up from Sleep mode and should resume the functionality of the peripherals, the following lines must be included in the wakeup handler function(s) that were registered using `wkupct_register_callback()` and/or `quad_decoder_register_callback()`:

```
// Init System Power Domain blocks: GPIO, WD Timer, Sys Timer, etc.
// Power up and init Peripheral Power Domain blocks,
// and finally release the pad latches.
  if(GetBits16(SYS_STAT_REG, PER_IS_DOWN))
        periph_init();
```

Register a callback function that is called from the driver's WKUP_QUADEC_IRQn interrupt handler, using `wkupct_register_callback()`.

Enable the WKUP_QUADEC_IRQn interrupt request with the wakeup parameters, using `wkupct_enable_irq()`.

10.7.2 Available Functions

- `wkupct_register_callback()`
- `wkupct_enable_irq()`
- `wkupct_disable_irq()`

10.7.3 Function Summary

`wkupct_register_callback()`: Registers a callback function that is called from the driver's WKUP_QUADEC_IRQn interrupt handler.

`wkupct_enable_irq()`: Registers the wakeup timer for use of WKUP_QUADEC_IRQn interrupt requests and enables the WKUP_QUADEC_IRQn with the desired wakeup parameters.

`wkupct_disable_irq()`: Unregisters the wakeup timer from the use of WKUP_QUADEC_IRQn interrupt requests and calls the `wkupct_quad_disable_IRQ()` function to disable the interrupts for the quadrature decoder and the wakeup timer.
Caution: The IRQ will be disabled only if it has not also been enabled by the quadrature decoder.

10.7.4 Function Reference

10.7.4.1 wkupct_register_callback

Function name	**void wkupct_register_callback(uint32_t* callback)**
Function description	Registers a callback function that is called from the driver's WKUP_QUADEC_IRQn interrupt handler.
Parameters	callback The user-defined callback function.
Return values	None
Notes	A local pointer to this function is stored in the retention memory area.

10.7.4.2 wkupct_enable_irq

Function name	**void wkupct_enable_irq(uint32_t sel_pins, uint32_t pol_pins, uint16_t events_num, uint16_t deb_time)**
Function description	Enables and configures the wakeup timer and enables the WKUP_QUADEC_IRQn.
Parameters	sel_pins Select enabled inputs (0: disabled, 1: enabled): Bits 0 to 7: port 0 Bits 8 to 15: port 1 Bits 16 to 23: port 2 Bits 24 to 31: port 3 pol_pins Input pin polarity (0: active HIGH, 1: active LOW). Bits 0 to 7: port 0 Bits 8 to 15: port 1 Bits 16 to 23: port 2 Bits 24 to 31: port 3 events_num Number of events before wakeup interrupt. (max. value: 255). deb_time Debouncing time (max. value: 0x3F).
Return values	None
Notes	

10.7.4.3 wkupct_disable_irq

Function name	**wkupct_quadec_error_t wkupct_disable_irq(void)**
Function description	Unregisters the wakeup timer from the use of the WKUP_QUADEC_IRQn interrupt requests and calls the wkupct_quad_disable_IRQ() function to disable the interrupts for the quadrature decoder and the wakeup timer, only when no registration from the quadrature decoder is active.
Parameters	None
Return values	WKUPCT_QUADEC_ERR_OK: No error. WKUPCT_QUADEC_ERR_RESERVED: The wakeup timer was previously registered for WKUP_QUADEC_IRQn.
Notes	

10.7.5 Definitions

```
enum
{
    SRC_WKUP_IRQ = 0x01,
    SRC_QUAD_IRQ,
};

typedef enum
{
    WKUPCT_QUADEC_ERR_RESERVED = -1,
    WKUPCT_QUADEC_ERR_OK = 0,
} wkupct_quadec_error_t;

typedef enum
{
    RESERVATION_STATUS_FREE = 0,
    RESERVATION_STATUS_RESERVED,
} reservation_status_t;

typedef void (*wakeup_handler_function_t)(void);
```

10.7.6 Defines in the Application for the WAKEUP TIMER Driver

The following preprocessor directive must be defined in the application, in order to include the necessary parts of the code:

```
WKUP_ENABLED
```

See section 10.7.1, important note 1.

10.8 PWM Timers

The following sections list the various functions of the PWM TIMERS driver library. These functions implement the various operations to support the configuration and operation of the TIMER0 and TIMER2 drivers:

TIMER0

- Controls the PWM signals PWM0 and PWM1 which is always the complementary of PWM0.
- If needed, the interrupt SWTIM_IRQn can be configured to be triggered in intervals configured separately.

TIMER2

- Controls the PWM signals PWM2, PWM3 and PWM4 which all use the same frequency with individually configured duty cycles.
- If needed, the interrupt SWTIM_IRQn can be enable to be triggered, in intervals that are separately configurable.

The source code for this driver is located in: `sdk\platform\driver\pwm`.

10.8.1 How to Use this Driver

This section contains instructions on the initialization, use and release of the PWM TIMERS library.

Important notes:

1. The user application is responsible for the correct configuration of any pads that are to be driven by the PWM0, PWM1 (TIMER0) and PWM2, PWM3, PWM4 (TIMER2) signals.
   For example, a line of the format:
   ```
   GPIO_ConfigurePin(GPIO_PORT_0, GPIO_PIN_1, OUTPUT, PID_PWM0, true);
   ```
   will configure pin P0_1 to be driven by the PWM0 signal.
2. The TIMER0/TIMER2 peripheral clock must be enabled for both TIMER0 and TIMER2, using `set_tmr_enable()`.

TIMER0

- Enable the TIMER0/TIMER2 peripheral clock, using `set_tmr_enable()`.
- To use the 16 MHz clock source, select the TIMER0/TIMER2 clock division factor, using `set_tmr_div()`. This setting does not apply in the case of the 32 kHz clock source.
- Initialize the PWM with the desired PWM mode, TIMER0 “on” time division option (**Note:** This only affects the “on” time) and clock source selection settings, using `timer0_init()`.
- Set the TIMER0 “on”, “high” and “low“ times, using `timer0_set()`.

To use interrupts:

- In the application, define a (callback) function of the form:
  `void pwm_user_callback_function(void)`
  Important note: Always keep the code size in this function to the bare minimum, in order to keep the application responsive.
- Register this callback function (will be called by the interrupt handler of SWTIM_IRQn), using `timer0_register_callback()`.
- Enable SWTIM_IRQn, using `timer0_enable_irq()`.
- Start TIMER0, using `timer0_start()`.
- Stop TIMER0, using `timer0_stop()`.
- Optionally, disable the TIMER0/TIMER2 peripheral clock, using `set_tmr_enable()`.
  Caution: When disabling the common peripheral clock, TIMER2 will also cease to run.

- Optionally, disable SWTIM_IRQn, using `timer0_disable_irq()`.

TIMER2

- Enable the TIMER0/TIMER2 peripheral clock, using `set_tmr_enable()`.
- Set the TIMER0/TIMER2 clock division factor, using `set_tmr_div()`.
  Note: This setting is common to both TIMER0 and TIMER2.
- Initialize PWM with the desired `hw_pause` behavior, `sw_pause` setting, using `timer2_init()`.
- Set the duty cycle for the desired PWM signal(s), using `timer2_set_pwm2_duty_cycle()`, `timer2_set_pwm3_duty_cycle()` and `timer2_set_pwm4_duty_cycle()`.
- When initialized with `sw_pause` enabled, release the `sw_pause`, using `timer2_set_sw_pause()`.
  In any case, the timer starts.
- Stop the timer, enabling `sw_pause` again when desired, using `timer2_set_sw_pause()`.
- Stop and disable the timer, using `timer2_stop()`.
- Optionally, disable the TIMER0/TIMER2 peripheral clock, using `set_tmr_enable()`.
  Caution: When disabling the common peripheral clock, TIMER0 will also cease to run.

10.8.2 Common Functions (TIMER0, TIMER2)

- `set_tmr_enable()`
- `set_tmr_div()`

10.8.3 TIMER0 functions (PWM0, PWM1)

- `timer0_init()`
- `timer0_start()`
- `timer0_stop()`
- `timer0_release()`
- `timer0_set_pwm_on_counter()`
- `timer0_set_pwm_high_counter()`
- `timer0_set_pwm_low_counter()`
- `timer0_set()`
- `timer0_enable_irq()`
- `timer0_disable_irq()`
- `timer0_register_callback()`

10.8.4 TIMER2 functions (PWM2, PWM3, PWM4)

- `timer2_enable()`
- `timer2_set_hw_pause()`
- `timer2_set_sw_pause()`
- `timer2_set_pwm_frequency()`
- `timer2_init()`
- `timer2_stop()`
- `timer2_set_pwm2_duty_cycle()`
- `timer2_set_pwm3_duty_cycle()`
- `timer2_set_pwm4_duty_cycle()`

10.8.5 Function Summary

10.8.5.1 Common Functions (TIMER0, TIMER2)

`set_tmr_enable()`: Enables the peripheral clock to both TIMER0 and TIMER2.

`set_tmr_div()`: Sets the division factor for the peripheral clock (not applicable for 32 kHz clock source).

10.8.5.2 TIMER0 Functions

`timer0_init()`: Initializes TIMER0. The PWM mode of operation, the TIMER0 “on” time clock division option and the clock source are selected here.

`timer0_start()`: Starts TIMER0, if it has been initialized using `timer0_init()`.

`timer0_stop()`: Stops TIMER0.

`timer0_release()`: Same as `timer0_stop()`.

`timer0_set_pwm_on_counter()`: Sets the value of TIMER0 “on(ON)” counter. This is the counter that controls the intervals between SWTIM_IRQn interrupts.

`timer0_set_pwm_high_counter()`: Sets the value of TIMER0 “high(M)” counter.

`timer0_set_pwm_low_counter()`: Sets the value of TIMER0 “low(N)” counter.

`timer0_set`: Sets the values of the “on(ON)”, “high(M)” and “low(N)” counters, in a single function call.

`timer0_enable_irq()`: Enables the SWTIM_IRQn IRQ.

`timer0_disable_irq()`: Disables the SWTIM_IRQn IRQ.

`timer0_register_callback()`: Registers a user defined callback function that is called from the SWTIM_IRQn interrupt handler of the driver.

Important note: Always keep the code size in this function to the bare minimum, in order to keep the application responsive.

10.8.5.3 TIMER2 Functions

`timer2_enable()`: Enables/disables TIMER2.

`timer2_set_hw_pause()`: Enables/disables the hw pause feature of TIMER2.

`timer2_set_sw_pause()`: Enables/disables the sw pause feature of TIMER2.

`timer2_set_pwm_frequency()`: Sets the pwm frequency of TIMER2.

`timer2_init()`: Enables/disables the `hw_pause` and the `sw_pause` features of TIMER2 and sets the PWM frequency of TIMER2, in a single function call.

`timer2_stop()`: Stops TIMER2.

`timer2_set_pwm2_duty_cycle()`: Sets the duty cycle of PWM2.

`timer2_set_pwm3_duty_cycle()`: Sets the duty cycle of PWM3.

`timer2_set_pwm4_duty_cycle()`: Sets the duty cycle of PWM4.

10.8.6 Function Reference: Common Functions (TIMER0, TIMER2)

10.8.6.1 set_tmr_enable

Function name	**void set_tmr_enable(CLK_PER_REG_TMR_ENABLE_t clk_per_reg_tmr_enable)**
Function description	Enables the peripheral clock to both TIMER0 and TIMER2.
Parameters	clk_per_reg_tmr_enable CLK_PER_REG_TMR_DISABLED Disables the peripheral clock. CLK_PER_REG_TMR_ENABLED Enables the peripheral clock.
Return values	None
Notes	

10.8.6.2 set_tmr_div

Function name	**void set_tmr_div(CLK_PER_REG_TMR_DIV_t per_tmr_div)**
Function Description	Sets the division factor for the peripheral clock.
Parameters	per_tmr_div CLK_PER_REG_TMR_DIV_1 Clock peripheral division factor is 1. CLK_PER_REG_TMR_DIV_2 Clock peripheral division factor is 2. CLK_PER_REG_TMR_DIV_4 Clock peripheral division factor is 4. CLK_PER_REG_TMR_DIV_8 Clock peripheral division factor is 8.
Return values	None
Notes	Not applicable for 32 kHz clock source. Affects TIMER0 when clocked from 16 MHz clock and TIMER2 always.

10.8.7 Function Reference: TIMER0 Functions

10.8.7.1 timer0_init

Function name	**void timer0_init(TIM0_CLK_SEL_t tim0_clk_sel , PWM_MODE_t pwm_mode, TIM0_CLK_DIV_t tim0_clk_div)**
Function description	Initializes TIMER0.
Parameters	pwm_mode PWM_MODE_ONE: The PWM signal will be always HIGH during the “high time”. PWM_MODE_CLOCK_DIV_BY_TWO: The PWM signals are not HIGH during the “high time” but output a clock in that stage. The frequency is based on the 16 MHz peripheral clock (also when 32 kHz clock is used as timer clock source), divided by the value set with timer0_init, but divided by two to get a 50 % duty cycle. For example, when a factor of 8 has been selected, the clock that will be observed during the “high times” of PWM0 and PWM1, will have a frequency of: (16 MHz / 8) / 2 = 1 MHz, irrespective of the selected clock source for TIMER0. tim0_clk_div TIMER0 “on” time (on duration) division factor. **Note:** This parameter affects only the PWM “on” time. It also affects the intervals between SWTIM_IRQn interrupts. TIM0_CLK_NO_DIV The internal “on” counter is clocked by the same clock as TIMER0. TIM0_CLK_DIV_BY_10 The internal “on” counter is clocked by 1/10 of the clock used for TIMER0. tim0_clk_sel Selects the clock source user for TIMER0. TIM0_CLK_32K The 32 kHz clock source is used. TIM0_CLK_FAST The 16 MHz clock source is used.
Return values	None
Notes	

10.8.7.2 timer0_start

Function name	**void timer0_start(void)**
Function description	Starts TIMER0, when it has been previously initialized.
Parameters	None
Return values	None
Notes	The timer must have been initialized using timer0_init().

10.8.7.3 timer0_stop

Function name	**void timer0_stop(void)**
Function description	Stops TIMER0.
Parameters	None
Return values	None
Notes	

10.8.7.4 timer0_release

Function name	**void timer0_release(void)**
Function description	Stops TIMER0. Same function as timer0_stop().
Parameters	None
Return values	None
Notes	Exists for compliance to the driver architecture nomenclature.

10.8.7.5 timer0_set_pwm_on_counter

Function name	**void timer0_set_pwm_on_counter(uint16_t pwm_on)**
Function description	Sets the PWM “ON” counter value.
Parameters	pwm_on The PWM “ON” counter value to set.
Return values	None
Notes	Is directly related to the value of the tim0_clk_div parameter in timer0_init().

10.8.7.6 timer0_set_pwm_high_counter

Function name	**void timer0_set_pwm_high_counter(uint16_t pwm_high)**
Function description	Sets the PWM “HIGH” counter value.
Parameters	pwm_high The PWM “HIGH” counter value to set.
Return values	None
Notes	

10.8.7.7 timer0_set_pwm_low_counter

Function name	**void timer0_set_pwm_low_counter(uint16_t pwm_low)**
Function description	Sets the PWM “LOW” counter value.
Parameters	pwm_low The PWM “LOW” counter value to set.
Return values	None
Notes	

10.8.7.8 timer0_set

Function name	**void timer0_set(uint16_t pwm_on, uint16_t pwm_high, uint16_t pwm_low)**
Function description	Sets the PWM “ON”, “HIGH” and “LOW” counter values in a single function call.
Parameters	pwm_on The PWM “ON” counter value to set. pwm_high The PWM “HIGH” counter value to set. pwm_low The PWM “LOW” counter value to set.
Return values	None
Notes	Please refer to the sections describing functions timer0_set_pwm_on_counter(), timer0_set_pwm_high_counter() and timer0_set_pwm_low_counter().

10.8.7.9 timer0_enable_irq

Function name	**void timer0_enable_irq(void)**
Function description	Enables the SWTIM_IRQn interrupt request.
Parameters	None
Return values	None
Notes	

10.8.7.10 timer0_disable_irq

Function name	**void timer0_disable_irq(void)**
Function description	Disables the SWTIM_IRQn interrupt request.
Parameters	None
Return values	None
Notes	

10.8.7.11 timer0_register_callback

Function name	**void timer0_register_callback(timer0_handler_function_t* callback)**
Function description	Registers a callback function that is called from the body of the SWTIM_IRQn IRQ handler in the driver.
Parameters	`callback` The user callback function.
Return values	None
Notes	The user callback function must be of type `timer0_handler_function_t`.

10.8.8 Function Reference: TIMER2 Functions

10.8.8.1 timer2_enable

Function name	**void timer2_enable(TRIPLE_PWM_ENABLE_t triple_pwm_enable)**
Function description	Enables/disables TIMER2.
Parameters	`triple_pwm_enable` `TRIPLE_PWM_DISABLED` TIMER2 is disabled. `TRIPLE_PWM_ENABLED` TIMER2 is enabled.
Return values	None
Notes	Disabling TIMER2 does not disable the TIM clock, as this is shared with TIMER0.

10.8.8.2 timer2_set_hw_pause

Function name	**void timer2_set_hw_pause(TRIPLE_PWM_HW_PAUSE_EN_t hw_pause_en)**
Function description	Enables/disables the “pause_by_hw” TIMER2 feature, that allows for the hardware to disable the TIMER2 PWM, for instance during radio transmission and reception, to reduce interference (bit `HW_PAUSE_EN` of register `TRIPLE_PWM_CTRL_REG`).
Parameters	`hw_pause_en` `HW_CAN_NOT_PAUSE_PWM_2_3_4` Hardware cannot pause TIMER2. `HW_CAN_PAUSE_PWM_2_3_4` Hardware can pause TIMER2.
Return values	None
Notes	

10.8.8.3 timer2_set_sw_pause

Function name	**void timer2_set_sw_pause(TRIPLE_PWM_SW_PAUSE_EN_t sw_pause_en)**
Function description	Pauses/resumes TIMER2 operation (bit SW_PAUSE_EN of register TRIPLE_PWM_CTRL_REG).
Parameters	sw_pause_en PWM_2_3_4_SW_PAUSE_DISABLED — TIMER2 is paused by software. PWM_2_3_4_SW_PAUSE_ENABLED — TIMER2 operation resumes.
Return values	None
Notes	

10.8.8.4 timer2_set_pwm_frequency

Function name	**void timer2_set_pwm_frequency(uint16_t triple_pwm_frequency)**
Function description	Sets the internal counter reload value that controls the TIMER2 frequency.
Parameters	triple_pwm_frequency — The value that will be automatically reloaded to the counter that determines the TIMER2 frequency. Example: When this value is 500d and the per_tmr_div parameter in function set_tmr_div() is CLK_PER_REG_TMR_DIV_8, the TIMER2 frequency (PWM2, PWM3, PWM4) will be: (16 MHz / 8) / 500d = 4 kHz.
Return values	None
Notes	

10.8.8.5 timer2_init

Function name	**void timer2_init(TRIPLE_PWM_HW_PAUSE_EN_t hw_pause_en, TRIPLE_PWM_SW_PAUSE_EN_t sw_pause_en, uint16_t triple_pwm_frequency)**
Function description	Initializes TIMER2 parameters in a single function call.
Parameters	hw_pause_en HW_CAN_NOT_PAUSE_PWM_2_3_4 — Hardware cannot pause TIMER2. HW_CAN_PAUSE_PWM_2_3_4 — Hardware can pause TIMER2. sw_pause_en PWM_2_3_4_SW_PAUSE_DISABLED — TIMER2 is paused by software. PWM_2_3_4_SW_PAUSE_ENABLED — TIMER2 operation resumes. triple_pwm_frequency — The value that will be automatically reloaded to the counter that determines the TIMER2 frequency. Example: When this value is 500d and the per_tmr_div parameter in function set_tmr_div() is CLK_PER_REG_TMR_DIV_8, the TIMER2 frequency (PWM2, PWM3, PWM4) will be: (16 MHz / 8) / 500d = 4 kHz.
Return values	None
Notes	The parameters can also be set individually, using the dedicated driver functions.

10.8.8.6 timer2_stop

Function name	**void timer2_stop(void)**
Function description	Stops TIMER2. Same function as timer2_enable(TRIPLE_PWM_DISABLED).
Parameters	None
Return values	None
Notes	Disabling TIMER2 does not disable the TIM clock, as this is shared with TIMER0.

10.8.8.7 timer2_set_pwm2_duty_cycle

Function name	**void timer2_set_pwm2_duty_cycle(uint16_t pwm2_duty_cycle)**
Function description	Sets the internal counter reload value that controls the TIMER2 PWM2 duty cycle.
Parameters	pwm2_duty_cycle — The reload value used by the internal counter that determines the TIMER2 PWM2 duty cycle. Example: When the pwm_frequency parameter of the function timer2_set_pwm_frequency() is 500d and the pwm2_duty_cycle parameter is 100 d, the duty cycle will be 100 / 500 = 20 %.
Return values	None
Notes	

10.8.8.8 timer2_set_pwm3_duty_cycle

Function name	**void timer2_set_pwm3_duty_cycle(uint16_t pwm3_duty_cycle)**
Function description	Sets the internal counter reload value that controls the TIMER2 PWM3 duty cycle.
Parameters	pwm3_duty_cycle — The reload value used by the internal counter that determines the TIMER2 PWM3 duty cycle.
Return values	None
Notes	Related to the pwm_frequency parameter. See example in function timer2_set_pwm2_duty_cycle().

10.8.8.9 timer2_set_pwm4_duty_cycle

Function name	**void timer2_set_pwm4_duty_cycle(uint16_t pwm4_duty_cycle)**
Function description	Sets the internal counter reload value that controls the TIMER2 PWM4 duty cycle.
Parameters	pwm4_duty_cycle — The reload value used by the internal counter that determines the TIMER2 PWM4 duty cycle.
Return values	None
Notes	Related to the pwm_frequency parameter. See example in function timer2_set_pwm2_duty_cycle().

10.8.9 Definitions

```
typedef enum
{
    PWM_MODE_ONE,
    PWM_MODE_CLOCK_DIV_BY_TWO
} PWM_MODE_t;

typedef enum
{
    TIM0_CLK_DIV_BY_10,
    TIM0_CLK_NO_DIV
} TIM0_CLK_DIV_t;

typedef enum
{
    TIM0_CLK_32K,
    TIM0_CLK_FAST
} TIM0_CLK_SEL_t;

typedef enum
{
    TIM0_CTRL_OFF_RESET,
    TIM0_CTRL_RUNNING
} TIM0_CTRL_t;

typedef enum
{
    CLK_PER_REG_TMR_DISABLED,
    CLK_PER_REG_TMR_ENABLED,
} CLK_PER_REG_TMR_ENABLE_t;

typedef enum
{
    CLK_PER_REG_TMR_DIV_1,
    CLK_PER_REG_TMR_DIV_2,
    CLK_PER_REG_TMR_DIV_4,
    CLK_PER_REG_TMR_DIV_8
} CLK_PER_REG_TMR_DIV_t;

typedef enum
{
    HW_CAN_NOT_PAUSE_PWM_2_3_4,
    HW_CAN_PAUSE_PWM_2_3_4
} TRIPLE_PWM_HW_PAUSE_EN_t;

typedef enum
{
    PWM_2_3_4_SW_PAUSE_DISABLED,
    PWM_2_3_4_SW_PAUSE_ENABLED
} TRIPLE_PWM_SW_PAUSE_EN_t;

typedef enum
{
    TRIPLE_PWM_DISABLED,
    TRIPLE_PWM_ENABLED
} TRIPLE_PWM_ENABLE_t;

typedef void (timer0_handler_function_t)(void);
```


10.9 SysTick Timer

The following sections list the various functions of the SysTick TIMER driver library. The source code for this driver is located in: `sdk\platform\driver\systick`.

10.9.1 How to Use this Driver

- Register a callback function that is called from the driver's `SysTick_Handler()` interrupt handler, using `systick_register_callback()`.
- Enable the timer and select whether an interrupt will be generated when the timer counts down to 0, using `systick_start()`.
- Read the current value of the timer, using `systick_value()`.
- Disable the timer and clear its value, using `systick_stop()`.
- Generate an accurate inline delay with a single function call, using `systick_wait()`.

10.9.2 Available Functions

- `systick_register_callback()`
- `systick_start()`
- `systick_stop()`
- `systick_value()`
- `systick_wait()`

10.9.3 Function Summary

`systick_register_callback()`: Registers a callback function that is called from the driver's `SysTick_Handler()` interrupt handler.

`systick_start()`: Enables the timer and selects whether an interrupt will be generated when the timer counts down to 0.

`systick_stop()`: Disables the timer and clears its value.

`systick_value()`: Reads the current value of the timer.

`systick_wait()`: Creates a delay without having to manually start the timer, create a callback function or poll for a value of 0 and then stop the timer.

10.9.4 Function Reference

10.9.4.1 systick_register_callback

Function name	**`void systick_register_callback(systick_callback_function_t callback)`**
Function description	Registers a callback function that is called from the driver's `SysTick_Handler()` exception handler.
Parameters	`callback` The user-defined callback function.
Return values	None
Notes	A pointer to this function is stored in the `systick_callback_function` global variable.

10.9.4.2 systick_start()

Function name	**`void systick_start(uint32_t ticks, bool exception)`**
Function description	Enables the SysTick timer and selects whether an exception request will be generated when the timer counts down to 0.
Parameters	`ticks` The duration of the countdown. `exception` Enables generating an exception request when the SysTick timer counts down to 0. `TRUE` SysTick exception request is enabled. `FALSE` SysTick exception request is disabled.
Return values	None
Notes	

10.9.4.3 systick_stop()

Function name	**`void systick_stop(void)`**
Function description	Disables the SysTick timer and clears its Value register.
Parameters	None
Return values	None
Notes	

10.9.4.4 systick_value()

Function name	**`uint32_t systick_value(void)`**
Function description	Reads the current value of the SysTick timer.
Parameters	None
Return values	The current value of the timer.
Notes	

10.9.4.5 systick_wait()

Function name	**`void systick_wait(uint32_t ticks)`**
Function description	Creates a delay without having to manually start the timer, create a callback function or poll for a value of 0 and then stop the timer.
Parameters	`ticks` The duration of the delay.
Return values	None
Notes	

10.9.5 Definitions

```
typedef void (*systick_callback_function_t)(void);
```

10.9.6 Global Variables and Constants

```
systick_callback_function_t systick_callback_function = NULL;
bool systick_core_clock = true;
```

10.10 SPI Flash Driver

The following sections list the various functions and data structures of the SPI Flash driver library, which enables the use of various SPI Flash memory devices through a uniform API.

Although a number of SPI Flash devices is directly supported by the driver (see Appendix B), support for other devices can be added by the user (for instructions see Appendix B.2). This driver uses the SPI functions included in the SPI Driver libraries (`spi.c`).

The source code for this driver is located in: `sdk\platform\driver\spi_flash`.

10.10.1 How to Use this Driver

In order to work with an SPI Flash memory device, the following parameters must be known:

- The size of the memory array in bytes
- The page size that the memory device uses. SPI Flash memory devices are internally organized in pages of a fixed size.

For the directly supported devices (see Appendix B) these parameters can automatically be retrieved upon automatic detection of the device (see function `spi_flash_auto_detect()`).

10.10.2 Initialization and Configuration

Before using the SPI Flash driver, after the initial power up or wakeup of the application, the SPI Flash driver has to be (re-)initialized in the following sequence:

1. Ensure that the application has configured the SPI related pads correctly (normally using the `periph_init()` function).
2. Initialize the SPI interface using `spi_flash_init()`.
3. Attempt automatic detection of the SPI Flash device, using `spi_flash_auto_detect()`. The device parameters (`spi_flash_size`, `spi_flash_page_size`) are retrieved automatically.
4. Initialize the Flash memory with the `spi_flash_size` and `spi_flash_page_size` parameters, using `spi_flash_init()`.

10.10.3 Controlling Write Access

- Set the Write Enable Latch (WEL) bit of the SPI Flash status register to 1, using `spi_flash_set_write_enable()`.
- Enable the write operation for the volatile bits of the SPI Flash status register, using `spi_flash_write_enable_volatile()`.
- Reset the Write Enable Latch (WEL) bit of the SPI Flash status register to 0, using `spi_flash_write_disable()`.

10.10.4 Status Register Access

- Read the SPI Flash status register, using `spi_flash_read_status_reg()`.
- Write the SPI Flash status register, using `spi_flash_write_status_reg()`.

10.10.5 Reading

- Read data from the SPI Flash memory array, using `spi_flash_read_data()`.
- Read the Manufacturer/Device ID of the SPI Flash device, using `spi_read_flash_memory_man_and_dev_id()`.
- Read the Unique ID Number of the SPI Flash device, using `spi_read_flash_unique_id()`.
- Read the JEDEC ID of the SPI Flash device, using `spi_read_flash_jedec_id()`.

10.10.6 Writing

- Write within a page of the SPI Flash, using `spi_flash_page_program()`.
- Write any amount of data to the SPI Flash, `using spi_flash_write_data()`.

10.10.7 Erasing

- Erase either a sector (4 kB), a 32 kB block or a 64 kB block of the SPI Flash, using `spi_flash_block_erase()`.
- Try to erase the whole SPI Flash memory array, using `spi_flash_chip_erase()`. Any protected blocks of the memory will not be erased by this function.
- Erase the whole SPI Flash memory array, using `spi_flash_chip_erase_forced()`. Before the erasure of the SPI Flash memory array, all protection schemes are removed, thus ensuring that the whole memory array gets erased.

10.10.8 Data protection

Set the desired protection scheme of the device (which blocks of the memory become read-only, if any), using spi_flash_configure_memory_protection().

10.10.9 Function Reference: Initialization and Configuration Functions

10.10.9.1 spi_flash_auto_detect

Function name	**int8_t spi_flash_auto_detect(void)**
Function description	Allows the automatic detection of the SPI Flash device based on its JEDEC ID, provided it is one of the directly supported devices.
Parameters	None
Return values	The index of the detected SPI Flash device: 1: W25X10 2: W25X20 3: AT25DF011/ AT25DS011) 4: MX25V1006E_JEDEC_ID SPI_FLASH_AUTO_DETECT_NOT_DETECTED
Notes	For a list of the currently supported devices, refer to Appendix B.1.

10.10.9.2 spi_flash_init

Function name	**int8_t spi_flash_init(void)**
Function description	Initializes the SPI Flash driver and stores the parameters of the SPI Flash device in variables of the SPI Flash driver.
Parameters	None
Return values	ERR_OK, ERR_TIMEOUT
Notes	This function does not initialize the SPI interface. This has to be done using spi_init().

10.10.9.3 spi_flash_set_write_enable

Function name	**int8_t spi_flash_set_write_enable(void)**
Function description	Sets the Write Enable bit (WEL) in the SPI Flash Status Register. This function drives the /CS line LOW, writes the Write Enable instruction code and then drives the /CS line HIGH. Before returning, the function polls the SPI Flash Status Register to ensure that the WEL bit has been set.
Parameters	None
Return values	ERR_OK, ERR_TIMEOUT
Notes	The WEL bit must be set prior to every write or erase instruction. This provision is embedded in the user functions provided by the SPI Flash driver. Before any write operation takes place, these functions call spi_flash_set_write_enable(). In case write access cannot be achieved, a timeout occurs.

10.10.9.4 spi_flash_write_enable_volatile

Function name	**void spi_flash_write_enable_volatile(void)**
Function description	Enables writing of the non-volatile bits of the SPI Flash Status Register. This function must be called prior to a Write Status Register instruction.
Parameters	None
Return values	None
Notes	This function does not set the Write Enable Latch (WEL) bit.

10.10.9.5 spi_flash_write_disable

Function name	`void spi_flash_write_disable(void)`
Function description	Resets the Write Enable bit (WEL) in SPI Flash Status Register. This function drives the /CS line LOW, writes the Write Disable instruction code and then drives the /CS line HIGH. Before returning, the function polls the SPI Flash Status Register to ensure that the WEL bit has been reset.
Parameters	None
Return values	None
Notes	The WEL bit is automatically reset after power-up and upon completion of a write or erase instruction. This function can be used to reset the WEL bit before calling `spi_flash_write_status_reg()`.

10.10.9.6 spi_flash_read_status_reg

Function name	`uint8_t spi_flash_read_status_reg(void)`
Function description	Reads the value of the SPI Flash Status Register. This function drives the /CS line LOW, writes the Read Status Register instruction code, reads the value of the Status Register and then drives the /CS line HIGH.
Parameters	None
Return values	The SPI Flash Status Register value
Notes	This function may be called at any time, even while a write or erase cycle is in progress. This allows the BUSY status bit to be checked to determine when the cycle is complete and whether the device can accept another instruction. The SPI Flash Status Register can be read continuously.

10.10.9.7 spi_flash_write_status_reg

Function name	`void spi_flash_write_status_reg(uint8_t status)`
Function description	Writes an 8-bit value to the SPI Flash Status Register. This function issues a Write Enable instruction, then writes the Write Status Register instruction code and finally inputs the value to be written into the SPI Flash Status Register.
Parameters	`status` 8-bit value to be written to the status register.
Return values	None
Notes	Only non-volatile SPI Flash Status Register bits (7, 5, 3 and 2) can be written.

10.10.10 Function Reference: Flash Read Functions

10.10.10.1 spi_flash_read_data

Function name	**uint32_t spi_flash_read_data(uint8_t *rd_data_ptr, uint32_t address, uint32_t size)**
Function description	Reads a specified amount of data from the SPI Flash memory.
Parameters	`*rd_data_ptr` Pointer to the memory position where the read data will be stored. `address` Address of the first element to be read. `size` Size of the data block to be read.
Return values	Bytes actually read or `ERR_TIMEOUT` in case of failure.
Notes	If the size parameter exceeds the SPI Flash memory size available after the address, this function will read only the available size.

10.10.11 Function Reference: Flash Write Functions

10.10.11.1 spi_flash_page_program

Function name	**int32_t spi_flash_page_program(uint8_t *wr_data_ptr, uint32_t address, uint16_t size)**
Function description	Writes a specified amount of data to a page of the SPI Flash memory. The memory locations must have been erased (0xFF). This function sends a Write Enable instruction and then the Page Program instruction. After the transmission of all data to be written, the function polls the SPI Flash Status Register to determine the completion of the Page Program instruction.
Parameters	`*wr_data_ptr` Pointer to the memory position where the data to be written resides. `address` Address where the first element will be written. `size` Size of the data block to be written (1 to `spi_flash_page_size` bytes) as set during initialization.
Return values	Bytes actually written or `ERR_TIMEOUT` in case of failure.
Notes	When an entire page has to be programmed, the address must be a multiple of the page size. Otherwise, the addressing will wrap to the beginning of the page and the respective data will be overwritten. A partial page (fewer bytes than the page size) can be programmed without having any effect on other bytes within the same page. When the size parameter exceeds the SPI Flash memory page size available after the address, this function will only write the available size.

10.10.11.2 spi_flash_write_data

Function name	**int32 spi_flash_write_data(uint8_t *wr_data_ptr, uint32_t address, uint32_t size)**
Function description	Writes a specified amount of data to the SPI Flash memory. This function uses `spi_flash_page_program()`.
Parameters	`*wr_data_ptr` Pointer to the memory position where the data to be written resides. `address` Address where the first element will be written. `size` Size of the data block to be written.
Return values	Bytes actually written.
Notes	When the size parameter exceeds the SPI Flash memory size available after the address, this function will only write the available size.

10.10.11.3 spi_flash_page_fill

Function name	**int32_t spi_flash_page_fill(uint8_t value, uint32_t address, uint16_t size)**
Function description	Fills a range within a page of the SPI Flash memory with a 1-byte value. The memory locations must have been erased (0xFF). This function sends a Write Enable instruction and then the Page Program instruction. After the transmission of all data to be written, the function polls the SPI Flash Status Register to determine the completion of the Page Program instruction.
Parameters	value — The 1-byte value with which the memory range will be filled. address: — Starting address of the range to fill. size — Size of the range to be filled (1 to spi_flash_page_size bytes) as set during initialization.
Return values	Bytes actually written or ERR_TIMEOUT in case of failure.
Notes	When an entire page has to be programmed, the address must be a multiple of the page size. Otherwise, the addressing will wrap to the beginning of the page and the respective data will be overwritten. A partial page (fewer bytes than the page size) can be programmed without having any effect on other bytes within the same page. When the size parameter exceeds the SPI Flash memory page size available after the address, this function will write only the available size.

10.10.11.4 spi_flash_fill

Function name	**int32 spi_flash_fill(uint8_t value, uint32_t address, uint32_t size)**
Function description	Fills a range of the SPI Flash memory with a 1-byte value. This function uses spi_flash_page_fill().
Parameters	value — The 1-byte value to which the memory range will be filled. address — Starting address of the range to fill. size — Size of the range to be filled.
Return values	Bytes actually written.
Notes	When the size parameter exceeds the SPI Flash memory size available after the address, this function will only write up to the available size.

10.10.12 Function Reference: Flash Erase Functions

10.10.12.1 spi_flash_block_erase

Function name	**int8_t spi_flash_block_erase(uint32_t address, SPI_erase_module_t spiEraseModule)**
Function description	Erases the sector (4 kB), the 32 kB block or the 64 kB block to which the specified address belongs.
Parameters	address — Address belonging to the sector or block to be erased. spiEraseModule — Memory block size to be erased. SECTOR_ERASE — Erase 4 kB sector to which address belongs. BLOCK_ERASE_32 — Erase 32 kB block to which address belongs. BLOCK_ERASE_64 — Erase 64 kB block to which address belongs.
Return values	ERR_OK, ERR_TIMEOUT
Notes	This function does not verify the erasure.

10.10.12.2 spi_flash_chip_erase

Function name	**int8_t spi_flash_chip_erase (void)**
Function description	Erases the whole SPI Flash memory, except any protected blocks.
Parameters	None
Return values	ERR_OK, ERR_TIMEOUT
Notes	

10.10.12.3 spi_flash_chip_erase_forced

Function name	**int8_t spi_flash_chip_erase_forced (void)**
Function description	Erases the whole SPI Flash memory, after disabling all protection schemes.
Parameters	None
Return values	ERR_OK, ERR_TIMEOUT, ERR_UNKNOWN_FLASH_TYPE
Notes	Only for the supported types of Flash memory modules.

10.10.13 Function Reference: Power Management Functions

10.10.13.1 spi_flash_power_down

Function name	**int32_t spi_flash_power_down(void)**
Function description	Sends the Power-Down instruction to the SPI Flash memory. In power-down mode, all function calls except `spi_flash_release_from_power_down()` are ignored.
Parameters	None
Return values	ERR_OK, ERR_TIMEOUT
Notes	Only for the supported types of SPI Flash memory modules.

10.10.13.2 spi_flash_release_from_power_down

Function name	**int32_t spi_flash_release_from_power_down(void)**
Function description	Sends the Release from Power-Down instruction to the SPI Flash memory.
Parameters	None
Return values	ERR_OK, ERR_TIMEOUT
Notes	Only for the supported types of SPI Flash memory modules.

10.10.14 Function Reference: Data Protection Functions

10.10.14.1 spi_flash_configure_memory_protection

Function name	**int32_t spi_flash_configure_memory_protection(uint8_t spi_flash_memory_protection_setting)**
Function description	Configures the memory protection scheme to be applied to the SPI Flash memory.
Parameters	`spi_flash_memory_protection_setting` The memory protection to be applied. For the W25X10 memory device: `W25x10_MEM_PROT_NONE` Whole memory array is unprotected. `W25x10_MEM_PROT_UPPER_HALF` Upper half of the memory is protected. `W25x10_MEM_PROT_LOWER_HALF` Lower half of the memory is protected. `W25x10_MEM_PROT_ALL` Whole memory array is protected. For the W25X20 memory device: `W25x20_MEM_PROT_NONE` Whole memory array is unprotected. `W25x20_MEM_PROT_UPPER_QUARTER` Upper quarter of the memory is protected. `W25x20_MEM_PROT_UPPER_HALF` Upper half of the memory is protected. `W25x20_MEM_PROT_LOWER_QUARTER` Lower quarter of the memory is protected. `W25x20_MEM_PROT_LOWER_HALF` Lower half of the memory is protected. `W25x20_MEM_PROT_ALL` Whole memory array is protected. For the AT25DF011/ AT25DS011 memory device: `AT25Dx011_MEM_PROT_NONE` Whole memory array is unprotected. `AT25Dx011_MEM_PROT_ENTIRE_MEMORY_PROTECTED` Whole memory array is protected.
Return values	`ERR_OK`, `ERR_TIMEOUT`, `ERR_UNKNOWN_FLASH_VENDOR`
Notes	Only for the supported types of Flash memory modules. When the device is not recognized, `ERR_UNKNOWN_FLASH_VENDOR` is returned.

10.10.15 Function Reference: Miscellaneous Functions

10.10.15.1 spi_read_flash_memory_man_and_dev_id

Function name	`int16_t spi_read_flash_memory_man_and_dev_id(void)`
Function description	Reads the Manufacturer/Device ID of the SPI Flash memory.
Parameters	None
Return values	The Manufacturer/Device ID or 0 (in case of a time out).
Notes	

10.10.15.2 spi_read_flash_unique_id

Function name	`uint64_t spi_read_flash_unique_id(void)`
Function description	Reads the Unique ID number of the SPI Flash memory.
Parameters	None
Return values	The Unique ID number or 0 (in case of a time out).
Notes	

10.10.15.3 spi_read_flash_jedec_id

Function name	`int32_t spi_read_flash_jedec_id(void)`
Function description	Reads the JEDEC ID of the SPI Flash memory.
Parameters	None
Return values	The JEDEC ID or 0 (in case of a time out).
Notes	

10.10.16 Definitions

```
#define SPI_FLASH_DEVICES_SUPPORTED_COUNT (3)

// 1. W25X10CL
#define SPI_FLASH_DEVICE_INDEX_W25X10 0
#define W25X10_MAN_DEV_ID 0xEF10
#define       W25X10_JEDEC_ID 0xEF3011
#define W25X10_JEDEC_ID_MATCHING_BITMASK 0xFFFFFF
#define W25X10_TOTAL_FLASH_SIZE 0x20000
#define W25X10_PAGE_SIZE 0x100
#define W25x10_MEM_PROT_NONE 0
#define W25x10_MEM_PROT_UPPER_HALF 4
#define W25x10_MEM_PROT_LOWER_HALF 36
#define W25x10_MEM_PROT_ALL 8

// 2. W25X20CL
#define SPI_FLASH_DEVICE_INDEX_W25X20 1
#define W25X20_MAN_DEV_ID 0xEF11
#define       W25X20_JEDEC_ID 0xEF3012
#define W25X20_JEDEC_ID_MATCHING_BITMASK 0xFFFFFF
#define W25X20_TOTAL_FLASH_SIZE 0x40000
#define W25X20_PAGE_SIZE 0x100
#define W25x20_MEM_PROT_NONE 0
#define W25x20_MEM_PROT_UPPER_QUARTER 4
#define W25x20_MEM_PROT_UPPER_HALF 8
#define W25x20_MEM_PROT_LOWER_QUARTER 36
#define W25x20_MEM_PROT_LOWER_HALF 40
#define W25x20_MEM_PROT_ALL 12

// Parameters common to both W25X10 and W25X20
#define W25x_MEM_PROT_BITMASK 0x2C

// 3. AT25DN011, AT25DF011
#define SPI_FLASH_DEVICE_INDEX_AT25Dx011 2
#define AT25Dx011_JEDEC_ID 0x1F4200
#define AT25Dx011_JEDEC_ID_MATCHING_BITMASK 0xFFFF00
#define AT25Dx011_TOTAL_FLASH_SIZE 0x20000
#define AT25Dx011_PAGE_SIZE 0x100
#define AT25Dx011_MEM_PROT_BITMASK 4
#define AT25Dx011_MEM_PROT_NONE 0
#define AT25Dx011_MEM_PROT_ENTIRE_MEMORY_PROTECTED 4

// 4. MX25V1006E
#define SPI_FLASH_DEVICE_INDEX_MX25V1006E 3
#define MX25V1006E_MAN_DEV_ID 0xC210
#define MX25V1006E_JEDEC_ID 0xC22011
#define MX25V1006E_JEDEC_ID_MATCHING_BITMASK 0xFFFFFF
#define MX25V1006E_TOTAL_FLASH_SIZE 0x20000
#define MX25V1006E_PAGE_SIZE 0x100
#define MX25V1006E_MEM_PROT_BITMASK 0x0C
#define MX25V1006E_MEM_PROT_NONE 0
#define MX25V1006E_MEM_PROT_ENTIRE_MEMORY_PROTECTED 0x0C
typedef struct
{
uint32_t jedec_id;                      // JEDEC ID (3 bytes)
uint32_t jedec_id_matching_bitmask;     // bitmask of the JEDEC ID to derive //
matching
     uint32_t flash_size;                       // the total size in bytes
     uint32_t page_size;                        // the page size in bytes
```



```
    uint8_t memory_protection_bitmask;        // the memory protection-related bits
                                              // of the status register
uint8_t memory_protection_unprotected;   // the 'entire memory unprotected'
// status register value
} SPI_FLASH_DEVICE_PARAMETERS_BY_JEDEC_ID_t;

typedef enum SPI_ERASE_MODULE
{
    BLOCK_ERASE_64  = 0xd8,
    BLOCK_ERASE_32  = 0x52,
    SECTOR_ERASE    = 0x20,
} SPI_erase_module_t;

#define      MAX_READY_WAIT_COUNT 200000
#define      MAX_COMMAND_SEND_COUNT 50

/* Status Register Bits */
#define STATUS_BUSY        0x01
#define      STATUS_WEL          0x02
#define      STATUS_BP0          0x04
#define      STATUS_BP1          0x08
#define      STATUS_TB     0x20
#define      STATUS_SRP          0x80

#define ERR_OK                     0
#define ERR_TIMEOUT               -1
#define ERR_NOT_ERASED            -2
#define ERR_PROTECTED             -3
#define ERR_INVAL                 -4
#define ERR_ALIGN                 -5
#define ERR_UNKNOWN_FLASH_VENDOR  -6
#define ERR_UNKNOWN_FLASH_TYPE    -7
#define ERR_PROG_ERROR            -8

/* commands */
#define WRITE_ENABLE      0x06
#define WRITE_ENABLE_VOL  0x50
#define WRITE_DISABLE     0x04

#define READ_STATUS_REG   0x05
#define WRITE_STATUS_REG  0x01
#define PAGE_PROGRAM      0x02
#define QUAD_PAGE_PROGRAM 0x32
#define CHIP_ERASE        0xC7
#define ERASE_SUSPEND     0x75
#define ERASE_RESUME      0x7a
#define POWER_DOWN        0xb9
#define HIGH_PERF_MODE    0xa3
#define MODE_BIT_RESET    0xff
#define REL_POWER_DOWN    0xab
#define MAN_DEV_ID        0x90
#define READ_UNIQUE_ID    0x4b
#define JEDEC_ID          0x9f
#define READ_DATA         0x03
#define FAST_READ         0x0b

#define SPI_FLASH_AUTO_DETECT_NOT_DETECTED (-1)
```

10.10.17 Global Variables and Constants

```
// local copy of FLASH setup parameters
int16_t spi_flash_device_index;
const SPI_FLASH_DEVICE_PARAMETERS_BY_JEDEC_ID_t *spi_flash_detected_device;
uint32_t spi_flash_size;
uint32_t spi_flash_page_size;

const SPI_FLASH_DEVICE_PARAMETERS_BY_JEDEC_ID_t
SPI_FLASH_KNOWN_DEVICES_PARAMETERS_LIST[] =
{
{W25X10_JEDEC_ID, W25X10_JEDEC_ID_MATCHING_BITMASK, W25X10_TOTAL_FLASH_SIZE,
W25X10_PAGE_SIZE, W25x_MEM_PROT_BITMASK, W25x10_MEM_PROT_NONE},
{W25X20_JEDEC_ID, W25X20_JEDEC_ID_MATCHING_BITMASK, W25X20_TOTAL_FLASH_SIZE,
W25X20_PAGE_SIZE, W25x_MEM_PROT_BITMASK, W25x20_MEM_PROT_NONE},
{AT25Dx011_JEDEC_ID, AT25Dx011_JEDEC_ID_MATCHING_BITMASK, AT25Dx011_TOTAL_FLASH_SIZE,
AT25Dx011_PAGE_SIZE, AT25Dx011_MEM_PROT_BITMASK, AT25Dx011_MEM_PROT_NONE},
     {MX25V1006E_JEDEC_ID, MX25V1006E_JEDEC_ID_MATCHING_BITMASK,
MX25V1006E_TOTAL_FLASH_SIZE, MX25V1006E_PAGE_SIZE, MX25V1006E_MEM_PROT_BITMASK,
MX25V1006E_MEM_PROT_NONE},
};
```

10.11 I2C EEPROM Driver

The following sections list the various functions of the I2C EEPROM driver library. These functions implement the various operations of an I2C EEPROM (e.g. Microchip 24AA02, ST M24M01-R), such as initialization, configuration and release of the I2C controller. For guidelines on using a new I2C EEPROM device, see Appendix B.3.

The source code for this driver is located in: `sdk\platform\driver\i2c_eeprom`.

10.11.1 How to Use this Driver

- Enable the I2C module and configure it, using `i2c_eeprom_init()`.
- Read a random byte from the I2C EEPROM, using `i2c_eeprom_read_byte()`.
- Read a desired amount of data from the I2C EEPROM, using `i2c_eeprom_read_data()`.
- Write a random byte to the I2C EEPROM, using `i2c_eeprom_write_byte()`.
- Write a page to the I2C EEPROM, using `i2c_eeprom_write_page()`.
- Write a specified amount of data to the I2C EEPROM, using `i2c_eeprom_write_data()`.
- Upon completion, if desired, disable the I2C, using `i2c_eeprom_release()`.

10.11.2 Initialization and Configuration

- `i2c_eeprom_init()`
- `i2c_eeprom_release()`

10.11.3 Reading

- `i2c_eeprom_read_byte()`
- `i2c_eeprom_read_data()`

10.11.4 Writing

- `i2c_eeprom_write_byte()`
- `i2c_eeprom_write_page()`
- `i2c_eeprom_write_data()`

10.11.5 Function Reference: Initialization and Configuration Functions

The I2C EEPROM driver provides one function for the initialization and configuration of the I2C module, `i2c_eeprom_init()`, and one function to disable the I2C module, `i2c_eeprom_release()`.

10.11.5.1 i2c_eeprom_init

Function name	**`void i2c_eeprom_init(uint16_t dev_address, uint8_t speed, uint8_t address_mode, uint8_t address_size)`**
Function description	Initializes the I2C EEPROM according to the specified parameters. The I2C module is first disabled, then the control register is updated with the selected parameters and finally the module is enabled again.
Parameters	`dev_address` I2C slave address (device specific). `speed` I2C interface speed. `I2C_STANDARD` Standard (100 kbit/s) `I2C_FAST` Fast (400 kbit/s) `address_mode` I2C addressing mode. I2C_7BIT_ADDR 7-bit addressing I2C_10BIT_ADDR 10-bit addressing `address_size` Size of the I2C EEPROM address. I2C_1BYTE_ADDR 1-byte address I2C_2BYTES_ADDR 2-byte address I2C_3BYTES_ADDR 3-byte address
Return values	None
Notes	The I2C module is configured as bus master, with ‘send restart conditions’ enabled (by default).

10.11.5.2 i2c_eeprom_release

Function name	**`void i2c_eeprom_release(void)`**
Function description	Disables the I2C module. This function resets the `I2C_ENABLE` register and resets the `I2C_ENABLE` bit of the Peripheral divider register (`CLK_PER_REG`).
Parameters	None
Return values	None
Notes	

10.11.6 Function Reference: EEPROM Read Functions

10.11.6.1 i2c_eeprom_read_byte

Function name	**`i2c_error_code i2c_eeprom_read_byte(uint16_t address, uint8_t *byte)`**
Function description	Reads the byte that is stored at a specific address in the I2C EEPROM. This function first repeatedly makes a dummy access to poll the I2C Transmit Abort Source Register, until an acknowledgement (ACK) indicates that the I2C EEPROM is not busy executing another operation. Then the function writes the address to the I2C Rx/Tx Data Buffer, followed by a read command. Next, the function polls the I2C Receive FIFO Level Register, waiting for the read byte. As soon as the level on the I2C Receive FIFO Level Register is greater than 0, the function reads the byte that resides in the I2C Rx/Tx Data Buffer.
Parameters	`address` The memory location of the byte to be read. `*byte` The byte to be read.
Return values	The error code.
Notes	

10.11.6.2 i2c_eeprom_read_data

Function name	**`i2c_error_code i2c_eeprom_read_data(uint8_t *rd_data_ptr,uint16_t address, uint16_t size, uint32_t *bytes_read)`**
Function description	Reads a specified amount of data from a starting address of the I2C EEPROM. This function first repeatedly makes a dummy access to polls the I2C Transmit Abort Source Register, until an acknowledgement (ACK) indicates that the I2C EEPROM is not busy executing another operation. Then the function writes the starting address to the I2C Rx/Tx Data Buffer, followed by as many read commands as the given size. Next, the function polls the I2C Receive FIFO Level Register, waiting for the read bytes. As soon as the level on the I2C Receive FIFO Level Register is greater than 0, the function reads the data received from the I2C Rx/Tx Data Buffer.
Parameters	`*rd_data_ptr` Pointer to the memory position where the read data will be stored. `address` Address of the first element to be read. `size` Size of the data block to be read. `*bytes_read` Number of the bytes actually read.
Return values	The error code.
Notes	When the size parameter exceeds the EEPROM size available after the given address, this function will read only the available size. The read process is done in chunks of 64 bytes, due to the Rx FIFO limitation.

10.11.7 Function Reference: EEPROM Write Functions

10.11.7.1 i2c_eeprom_write_byte

Function name	**`i2c_error_code i2c_eeprom_write_byte(uint16_t address, uint8_t byte)`**
Function description	Writes a byte to a specific address in the I2C EEPROM. This function first repeatedly makes a dummy access to polls the I2C Transmit Abort Source Register, until an acknowledgement (ACK) indicating that the I2C EEPROM is not busy executing another operation. Then the function writes the specified address to the I2C Rx/Tx Data Buffer, followed by the byte to be written.
Parameters	`address` Address where the element will be written. `byte` Byte to be written.
Return values	The error code.
Notes	

10.11.7.2 i2c_eeprom_write_page

Function name	**`i2c_eeprom_write i2c_eeprom_write_page(uint8_t* wr_data_ptr , uint32_t address , uint16_t size, uint32_t *bytes_written)`**
Function description	Writes a specified amount of data to a page of the I2C EEPROM.
Parameters	`*wr_data_ptr` Pointer to the memory location of the data to be written. `address` Address where the first element will be written. `size` Size of the data block to be written (1 to I2C_EEPROM_PAGE). `*bytes_writen` Number of the bytes actually written.
Return values	The error code.
Notes	When an entire page has to be programmed, the address must be a multiple of the page size. Otherwise, the addressing will wrap to the beginning of the page and the respective data will be overwritten. A partial page (fewer bytes than the page size) can be programmed without having any effect on other bytes within the same page. When the size parameter exceeds the EEPROM page size available after the address, this function will only write the available size.

10.11.7.3 i2c_eeprom_write_data

Function name	**`uint32_t i2c_eeprom_write_data(uint8_t *wr_data_ptr,uint32_t address, uint16_t size)`**
Function description	Writes a specified amount of data to the I2C EEPROM. This function uses the `i2c_eeprom_write_page()` function. This function first checks whether the size parameter exceeds the available size from the address to the end of the EEPROM. Then it calculates and writes the amount of the bytes from address to the end of the corresponding EEPROM page. Next, it performs as many page writes as needed until the size has been written.
Parameters	`*wr_data_ptr` Pointer to the memory position of the data to be written. `address` Address where the first element will be written. `size` Size of the data block to be written. `*bytes_written` Number of bytes actually written.
Return values	The error code.
Notes	The address does not need to be a multiple of `I2C_EEPROM_PAGE`. When the size parameter exceeds the EEPROM size available after the address, this function will only write the available size.

10.11.8 Definitions

```
enum I2C_SPEED_MODES{
  I2C_STANDARD,
  I2C_FAST,
};
enum I2C_ADDRESS_MODES{
  I2C_7BIT_ADDR,
  I2C_10BIT_ADDR,
};
enum I2C_ADRESS_BYTES_COUNT{
  I2C_1BYTE_ADDR,
  I2C_2BYTES_ADDR,
  I2C_3BYTES_ADDR,
};

typedef enum
{
    I2C_NO_ERROR,
    I2C_7B_ADDR_NOACK_ERROR,
    I2C_INVALID_EEPROM_ADDRESS
} i2c_error_code;
```

10.11.9 Preprocessor definitions in the application for the I2C EEPROM driver

The following preprocessor directives must be defined to their corresponding values in the application, in order for the I2C EEPROM driver to handle the various requests.

`I2C_EEPROM_SIZE`: the size of the EEPROM in bytes.

`I2C_EEPROM_PAGE`: the EEPROM page size in bytes.

`I2C_SPEED_MODE`: the I2C interface speed (`I2C_STANDARD` or `I2C_FAST`.

`I2C_ADDRESS_MODE`: the I2C bus addressing mode (`I2C_7BIT_ADDR` or `I2C_10BIT_ADDR`).

`I2C_ADDRESS_SIZE`: the I2C EEPROM address size (`I2C_1BYTE_ADDR`, `I2C_2BYTES_ADDR` or `I2C_3BYTES_ADDR`).

10.12 Battery Level

The following sections list the various functions of the BATTERY driver library. These functions support the measurement and translation of the battery level.

The source code for this driver is located in: `sdk\platform\driver\battery`.

10.12.1 How to use this driver

Measure the battery level, using battery_get_lvl().

10.12.2 Function reference

10.12.2.1 battery_get_lvl

Function name	**`uint8_t battery_get_lvl(uint8_t batt_type)`**
Function description	Gets the voltage level of the battery.
Parameters	`batt_type` The code of the battery (ies). Used for the correct translation of the measured battery level, based on the battery's characteristics.
Return values	The battery level that is measured.
Notes	The power configuration (buck/boost mode) is detected automatically. When buck mode is detected for an AAA type battery, it is assumed that two AAA batteries in series are used. Stores a copy of the battery level measurement in the retention memory.

10.12.3 Definitions

```
// Battery types definitions
#define BATT_CR2032    1        //CR2032 coin cell battery
#define BATT_CR1225    2        //CR1225 coin cell battery
#define BATT_AAA       3        //AAA alkaline battery (boost: 1 cell, buck: 2 cells)

#define BATTERY_MEASUREMENT_BOOST_AT_1V5 (0x340)
#define BATTERY_MEASUREMENT_BOOST_AT_1V0 (0x230)
#define BATTERY_MEASUREMENT_BOOST_AT_0V9 (0x1F0)
#define BATTERY_MEASUREMENT_BOOST_AT_0V8 (0x1B0)

#define BATTERY_MEASUREMENT_BUCK_AT_3V0 (0x6B0)
#define BATTERY_MEASUREMENT_BUCK_AT_2V8 (0x640)
#define BATTERY_MEASUREMENT_BUCK_AT_2V6 (0x5D0)
#define BATTERY_MEASUREMENT_BUCK_AT_2V4 (0x550)
```

11 Development Environment

11.1 Overview

This section fist provides an overview for the DA1458x development environment consisting of the:

- **ARM Keil µVision IDE/Debugger, ARM C/C++ Compiler**, and its essential middleware components, **Keil IDE** and the **Keil build** tools
- ARM **Segger JTAG** cables and software that is fully supported by the Keil environment
- DA1458x **Software Development Kit (SDK)**

The development environment is also supported by a number of other utilities and tools such as:

- **SmartSnippets Toolbox**, which is a framework of PC based tools to control the DA14580/581/583 development kit, involving an:
  - OTP Programmer: a tool for OTP memory programming
  - UART booter: a tool for downloading hex files to DA14580/581 SRAM over UART
  - SPI and I2C memory programmer: Tool for SPI flash and I2C EEPROM programming
  - Power Profiler: a tool with which the user can track in real time the power consumption of the board, set software marker and correlate board consumption with events happening during the execution of software
- **Connection Manager**, which is a PC based software tool to control the link layer of the DA14580/581/583, with the following capabilities:
  - Functional in Peripheral and Central role
  - Set advertising parameters
  - Set connection parameters
  - Reading from Attribute database
  - Perform production test commands

In the following sections the Software Development Kit (SDK) contents and directory structure are presented.

11.2 Software Development Kit (SDK) Structure

The SDK structure is defined in the following sections.

11.2.1 root Directory

The SDK contains the following main directories.

```
.
├── binaries
├── config
├── doc
├── projects
├── sdk
└── utilities
```

Figure 14: root Directory Structure

11.2.2 binaries Directory

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

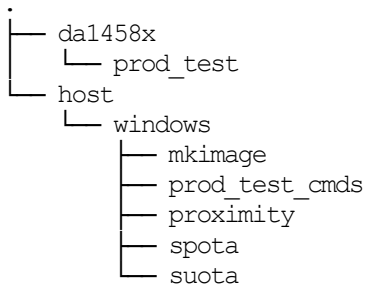


Figure 15: binaries Directory Structure

11.2.3 config Directory

This directory contains the DA1458x configuration file for the SmartSnippets tool.

11.2.4 doc Directory

This directory contains the SDK license files.

11.2.5 projects Directory

This directory contains the various SDK example projects. The projects directory is divided into two main directories:

- host_apps
- target_apps

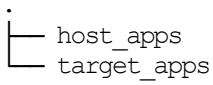


Figure 16. projects Directory Structure

11.2.5.1 host_apps Directory

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

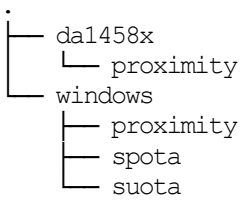


Figure 17: host_apps Directory Structure

11.2.5.2 target_apps Directory

This directory holds example applications that run on the DA14580/581/583 SoC. Each project directory contains Keil project files, along with the specific project's source code and configuration files.

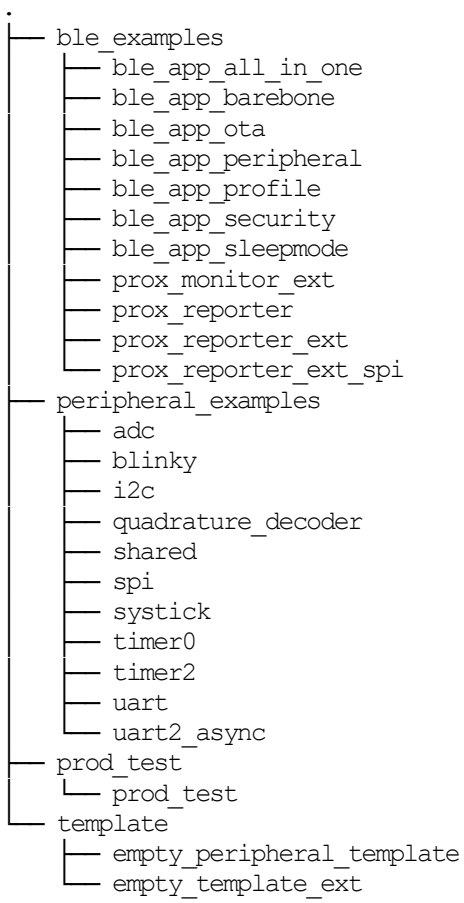


Figure 18: target_apps Directory Structure

The `ble_examples` directory contains DA14580/581/583 SoC BLE application examples for "Integrated processor" or "External processor" configuration. The ble_examples demonstrate the BLE functionality of the DA14580/581/583 SoC. The following list describes briefly each BLE example.

- `ble_app_barebone`, BLE example that demonstrates basic BLE procedures such as advertising, connection, connection parameters update and implementation of the device information service. It is based on the "Integrated processor" configuration.
- `ble_app_profile`, BLE example that demonstrates the same as the `ble_app_barebone` project, plus the implementation of a custom service (128-bit UUID) defined by the user. The application demonstrates only the custom database creation. It is based on the "Integrated processor" configuration.
- `ble_app_peripheral`, BLE example that demonstrates the same as the ble_app_profile. The application also adds some basic interaction over the provided custom service (read/write/notify values). It is based on the "Integrated processor" configuration.
- `ble_app_sleepmode`, BLE example that demonstrates the same as the ble_app_profile. The application adds the use of the sleep mode API, making use of the two available sleep modes – Extended Sleep and Deep Sleep. It is based on the "Integrated processor" configuration.
- `ble_app_security`, BLE example that demonstrates the same as the ble_app_profile. The application adds the various security/privacy features. It is based on the "Integrated processor" configuration.

- ble_app_ota, BLE example that demonstrates the same as the ble_app_profile. The application adds the over the air programming feature. It is based on the “Integrated processor” configuration.
- ble_app_all_in_one, BLE example that combine in one example the features of the previous ble_app_<example> projects. It is based on the “Integrated processor” configuration.
- prox_monitor_ext, BLE example that demonstrates the proximity monitor service. It also includes the device information client service. It uses the “External processor” configuration.
- prox_reporter, BLE example that demonstrates the proximity reporter service. It also includes the device information server service, the battery server service and the software patching over the air receiver (SPOTAR) service. It uses the “Integrated processor” configuration.
- prox_reporter_ext, BLE example that demonstrates the proximity reporter service. It also includes the device information server service and the software patching over the air receiver (SPOTAR) service. It uses the “External processor” configuration.
- prox_reporter_ext_spi, BLE example that demonstrates the proximity reporter service. It also includes the device information server service and the software patching over the air receiver (SPOTAR) service. It uses the “External processor” configuration over SPI interface.

The peripheral_examples directory contains DA14580/581/583 SoC peripheral examples. The examples demonstrate some of the non-BLE functionality of the DA14580/581/583 SoC. The following list describes briefly each peripheral example.

- adc, analog to digital conversion example.
- blinky, blinks a led.
- i2c, i2c interface example.
- quadrature_decoder, quadrature decoder example.
- shared, shared library for peripheral examples.
- spi, spi interface example.
- systick, SysTick timer control example.
- timer0, timer0 control example.
- timer2, timer2 control example.
- uart, uart communication example.

The prod_test directory contains the DA14580/581/583 SoC production tests examples.

The template directory contains DA14580/581/583 SoC BLE template examples, for “Integrated processor” or “External processor” configuration. The examples demonstrate the BLE functionality of the DA14580/581/583 SoC and act as a project template for the user. The following list describes briefly each BLE example.

- empty_peripheral_template, BLE example that demonstrates basic BLE functionality. It uses the “Integrated processor” configuration.
- empty_template_ext, BLE example that demonstrates basic BLE functionality. It uses the “External processor” configuration.

The Keil projects contained in the ble_examples, prod_test and template SDK directories are based on the following structure.

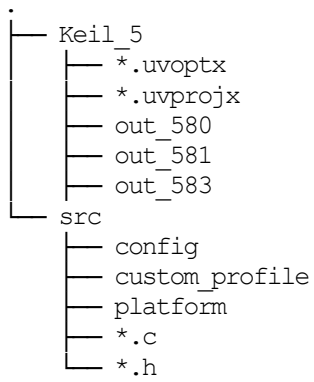


Figure 19: Project Directory Example Layout

The Keil_5 directory contains the Keil project files, `*.uvprojx` and `*.uvoptx`. The `out_580`, `out_581` and `out_583` directories are generated by the Keil program each time a project is compiled for the respective SoC (DA14580/581/583), and contain the compilation output files.

The `src` directory contains the user's application source code and header files (e.g. `user_<example>.c`, `user_<example>.h` or more) and three directories: `config`, `custom_profile` and `platform`. These three directories contain files that must be included in a user project structure. The names of these files must not be altered by the user.

Note: The `custom_profile` directory can be omitted when the user does not use a Custom profile in his application.

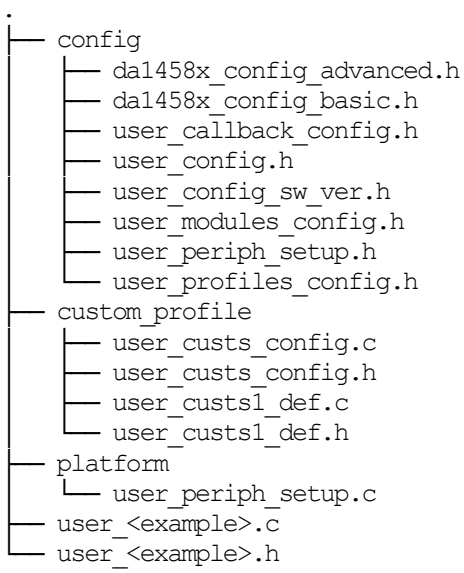


Figure 20: src Directory Example Layout

- `da1458x_config_advanced.h`, holds DA14580/581/583 advanced configuration settings.
- `da1458x_config_basic.h`, holds DA14580/581/583 basic configuration settings.
- `user_callback_config.h`, callback functions that handle various events or operations.
- `user_config.h`, holds advertising parameters, connection parameters, etc.
- `user_config_sw_ver.h`, holds user specific information about software version.
- `user_custs1_def.c`, defines the structure of the Custom 1 profile database structure.

- `user_custs_config.c`, defines the `cust_prf_funcs[] array`, which contains the Custom profiles API functions calls.
- `user_modules_config.h`, defines which application modules are included or excluded from the user's application.
- `user_periph_setup.h`, holds hardware related settings.
- `user_profiles_config.h`, defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user's application.
- `user_periph_setup.c`, source code file that handles peripheral configuration and initialization.

A detailed description of the aforementioned configuration files is included in Ref. [15].

11.2.6 sdk Directory

This directory holds the core files of the SDK. The directory structure of the core SDK modules is depicted below.

```
.
├── app_modules
│   ├── api
│   └── src
├── ble_stack
│   ├── controller
│   ├── hcic
│   ├── hcih
│   ├── host
│   ├── profiles
│   ├── rwble
│   └── rwble_hl
├── common_project_files
│   ├── misc
│   └── scatterfiles
└── platform
    ├── arch
    ├── core_modules
    ├── driver
    ├── include
    └── patch_code
```

Figure 21: sdk Directory Structure

11.2.6.1 app_modules Directory

This directory holds the application source and header files.

- `api`, contains the application header files.
- `src`, contains applications project specific code for some BLE profiles and handling functions for BLE operations, like advertising, connection, security/encryption, etc.

11.2.6.2 ble_stack Directory

This directory contains BLE stack related files.

11.2.6.3 common_project_files Directory

This directory contains the following folders plus three configuration header files.

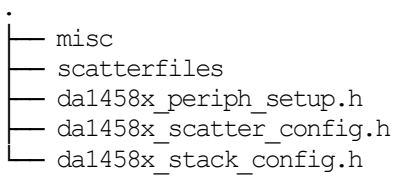


Figure 22: common_project_files Directory Structure

- misc, contains the ROM symbol definition files. This file will be used as input into the linker to create the final executable. The executable files, as well as the compilation outputs, are saved in a newly created directory named out_580, out_581 or out_583, depending on the selected SoC.
- scatterfiles, Keil scatter files for the DA14580/581/583 SoC.

The three configuration files are:

- da1458x_periph_setup.h, definitions of the used hardware platform.
- da1458x_scatter_config.h, definitions of the memory layout.
- da1458x_stack_config.h, definitions of the stack.

11.2.6.4 platform Directory

This directory contains the platform specific files for the ARM Cortex-M0 processor and its supported peripherals (BLE, serial interfaces, GPIOs, etc.).

- arch, contains the system files and the main() application function.
- core_modules, contains core system modules, like the kernel that implements the message handling, the GTL implementation, the non-volatile data storage manipulation, RF drivers, etc.
- driver, contains all the supported drivers for the ARM Cortex-M0 peripherals.
- include, header files of the core source files.
- patch_code, contains the object files of the patched ROM functions. More information for the patched functions is given the Release Notes of the SDK distribution.

11.2.7 utilities Directory

This directory holds utilities and tools that supplement the SDK.

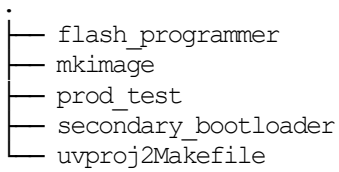


Figure 23: utilities Directory Structure

- flash_programmer, Flash programmer for DA14580/581/583 SoC.
- prod_test, production test utility.
- secondary_bootloader, secondary bootlader utility.
- uvproj2Makefile, utility that converts Keil4 project to Makefile.

A detailed description for some of the utilities is included in Ref. [20], Ref. [16] and the relevant source code.

Appendix A Memory Mapping and Non-Volatile Data Storage

A.1 Exchange Memory Mapping Possibilities

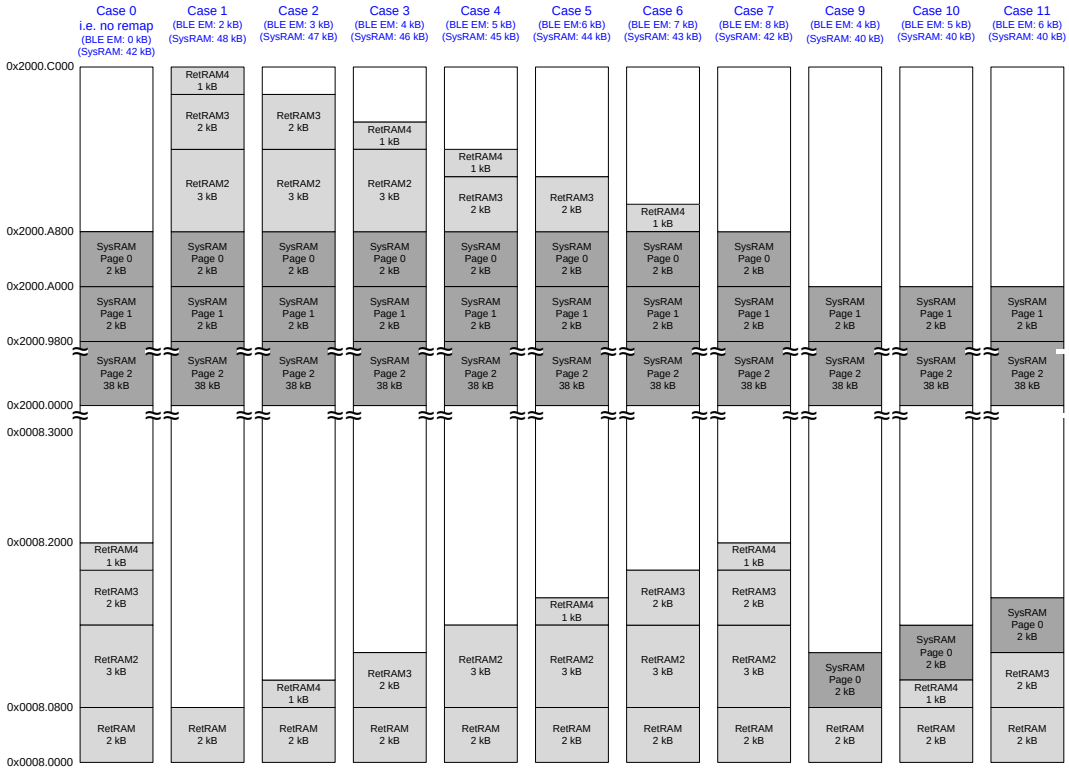


Figure 24: Memory Configurations (0..11)

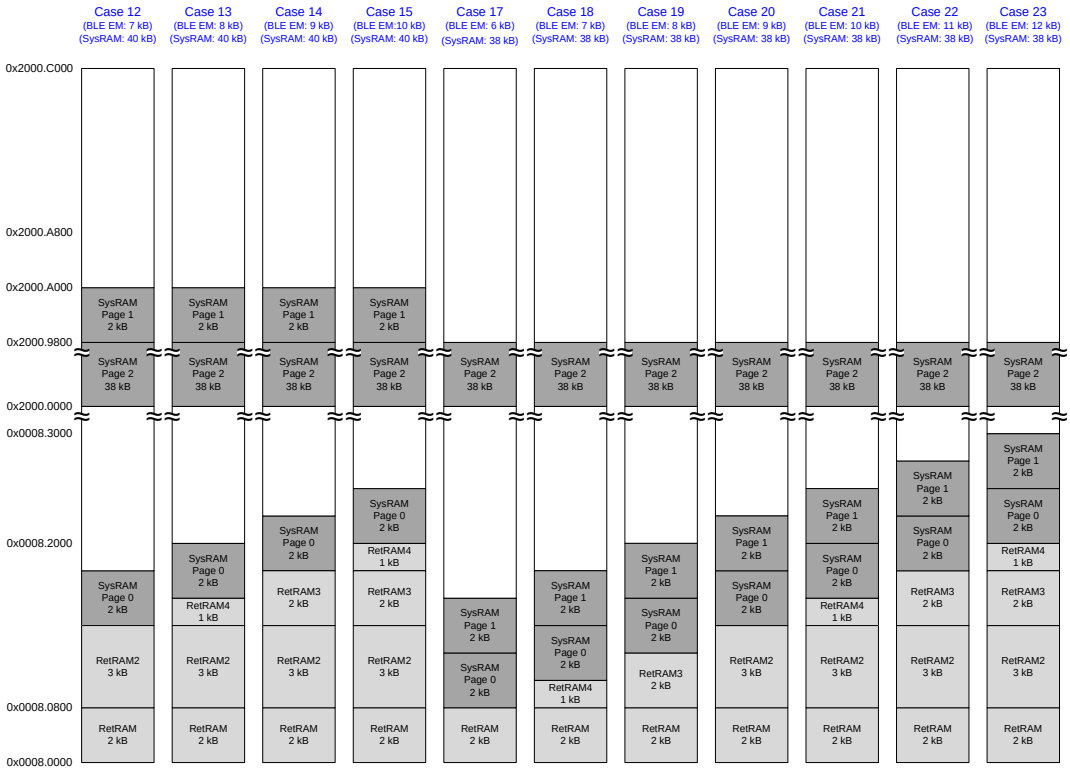


Figure 25: Memory Configurations (12..23)

A.2 Non-Volatile Data Storage

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

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

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

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

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

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

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

.

Appendix B Interfacing to SPI Flash and I2C EEPROM Devices

B.1 Supported SPI Flash Memory Devices

The Peripheral drivers API directly support the SPI Flash memory devices listed in Table 5.

Table 5: SPI Flash Memory Devices Directly Supported by the SPI Flash Driver

Part Number	Manufacturer	Total Ssize	Page Size
W25X10CL	Winbond	1 Mbit	256 B
W25X20CL	Winbond	2 Mbit	256 B
AT25DN011 AT25DF011 AT25DS011	Adesto	1 Mbit	256 B
MX25V1006E	Macronix	1 Mbit	256 B

B.2 Supporting Other SPI Flash Devices

B.2.1 Introduction

The Peripheral devices driver directly supports the SPI Flash memory devices listed in section B.1. This section explains how to add support for any other SPI Flash device to the SPI Flash library.

In most cases, the device to be supported will follow some standards (mostly maintained by JEDEC). The SPI transactions and a basic set of commands are expected to be compatible to the ones used in the already supported devices.

B.2.2 Command Set

Table 6 shows the commands and their respective opcodes of the currently supported SPI Flash devices. The last column of the table can be filled with the commands that are listed in the datasheet of the device under consideration. When multiple opcodes are offered for the same function, care has to been taken to select the opcode that is compatible with the existing SPI Flash driver API.

Table 6: Commands of Currently Supported SPI Flash Memory Devices

Command description	Command POpcode		
	W25x10CL W25x20CL	AT25Dx011 (x: F, N or S)	MX25V1006E
Read commands			
Read Array	0Bh, 03h	0Bh, 03h	0Bh, 03h
Dual Output Read	3Bh	3Bh	3Bh
Program and Erase commands			
Page Erase		81h	
Block Erase (4 kB)	20h	20h	20h
Block Erase (32 kB)	52h, D8h (64)	52h, D8h	52h, D8h
Chip Erase	60h, C7h	60h, C7h	60h, C7h
Chip Erase (legacy command)		62h	
Byte/Page Program (1 to 256 B)	02h	02h	02h
Protection commands			
Write Enable	06h	06h	06h

Command description	Command POpcode		
Write Disable	04h	04h	04h
Security commands			
Program OTP Security Register		9Bh	
Read OTP Security Register		77h	
Status Register commands			
Read Status Register	05h	05h	05h
Write Status Register Byte 1	01h	01h	01h
Write Status Register Byte 2		31h	
Miscellaneous commands			
Reset		F0h	
Read Manufacturer and Device ID	92h , 90h	9Fh	90h
JEDEC ID	9Fh	9Fh	90h
Read ID (legacy command)		15h	
Deep Power-Down	B9h	B9h	B9h
Resume from Deep Power-Down	ABh	ABh	ABh
Ultra-Deep Power-Down		79h	

B.2.3 How to Proceed

In order to use the SPI Flash driver efficiently, the level of compatibility of the new device has to be evaluated.

B.2.3.1 Device Is Highly Compatible

When the new device provides a compatible command set, at least for the functionality that will be actually used, most likely the device will be adequately supported, with no or just a few modifications.

It is recommended to enable the automatic detection and parameter selection of the device:

Add the new device details in the following structure (in `spi_flash.c`):

```
const SPI_FLASH_DEVICE_PARAMETERS_BY_JEDEC_ID_t
SPI_FLASH_KNOWN_DEVICES_PARAMETERS_LIST[] =
{
      {W25X10_JEDEC_ID, W25X10_JEDEC_ID_MATCHING_BITMASK, W25X10_TOTAL_FLASH_SIZE,
W25X10_PAGE_SIZE, W25x_MEM_PROT_BITMASK, W25x10_MEM_PROT_NONE},
      {W25X20_JEDEC_ID, W25X20_JEDEC_ID_MATCHING_BITMASK, W25X20_TOTAL_FLASH_SIZE,
W25X20_PAGE_SIZE, W25x_MEM_PROT_BITMASK, W25x20_MEM_PROT_NONE},
      {AT25Dx011_JEDEC_ID, AT25Dx011_JEDEC_ID_MATCHING_BITMASK,
AT25Dx011_TOTAL_FLASH_SIZE, AT25Dx011_PAGE_SIZE, AT25Dx011_MEM_PROT_BITMASK,
AT25Dx011_MEM_PROT_NONE},
      {MX25V1006E_JEDEC_ID, MX25V1006E_JEDEC_ID_MATCHING_BITMASK,
MX25V1006E_TOTAL_FLASH_SIZE, MX25V1006E_PAGE_SIZE, MX25V1006E_MEM_PROT_BITMASK,
MX25V1006E_MEM_PROT_NONE},
      {MY_DEVICE_JEDEC_ID, MY_DEVICE_JEDEC_ID_MATCHING_BITMASK,
MY_DEVICE_TOTAL_FLASH_SIZE, MY_DEVICE_PAGE_SIZE, MY_DEVICE_MEM_PROT_BITMASK,
MY_DEVICE_MEM_PROT_NONE},
};
```

Optionally add preprocessor constants in order to keep the code more readable (in spi_flash.h).

Example:

```
// 5. MY_DEVICE
#define SPI_FLASH_DEVICE_INDEX_MY_DEVICE 4
#define MY_DEVICE_JEDEC_ID 0x123456
#define MY_DEVICE_JEDEC_ID_MATCHING_BITMASK 0xFFFFFF
#define MY_DEVICE_TOTAL_FLASH_SIZE 0x20000
#define MY_DEVICE_PAGE_SIZE 0x100
#define MY_DEVICE_MEM_PROT_NONE 0
#define MY_DEVICE_MEM_PROT_UPPER_HALF 4
#define MY_DEVICE_MEM_PROT_LOWER_HALF 36
#define MY_DEVICE_MEM_PROT_ALL 8
```

Note: It is important to set the SPI_FLASH_DEVICE_INDEX_MY_DEVICE to the next available value. Also, the SPI_FLASH_DEVICES_SUPPORTED_COUNT must be set to the same value.

Example:

```
#define SPI_FLASH_DEVICE_INDEX_MY_DEVICE 4
#define SPI_FLASH_DEVICES_SUPPORTED_COUNT (4)
```

The …TOTAL_FLASH_SIZE and …PAGE_SIZE are expressed in bytes.

The …JEDEC_ID is the response to the command 0x9F (read JEDEC ID) and is used to identify the device automatically.

The …JEDEC_ID_MATCHING_BITMASK is used to select only parts of the JEDEC is for matching.

The …MEM_PROT_BITMASK is used to select the bits of the status register which are responsible for the memory protection functions.

B.2.3.2 Device Has Some Degree of Compatibility

When the device has some degree of compatibility, the recommended approach is to make a copy of the existing driver (spi_flash.c, spi_flash.h) and modify the opcodes of the commands in order to accommodate the necessary changes to support this new device.

B.2.3.3 Device Is Not Compatible

When either the SPI transaction formats deviate considerably from the ones used in the existing SPI Flash API and/or the command set is highly incompatible, it is recommended to develop a separate driver to support this new device.

B.3 Using Other I2C EEPROM devices

Adding support for new I2C EEPROM devices mostly involves parameter selection. The following parameters have to be determined and passed to the i2c_init() function:

- I2C slave address of the device.
- I2C speed: I2C_STANDARD (100 kbit/s) or I2C_FAST (400 kbit/s) depending on the device used and the hardware design.
- I2C addressing mode: I2C_7BIT_ADDR or I2C_10BIT_ADDR.
- Device address width (in bytes): I2C_1BYTE_ADDR, I2C_2BYTES_ADDR, I2C_3BYTES_ADDR.

The i2c_send_address() function may have to be adapted, in order to accommodate any special format of addressing. Example:

For the M24M01 device, which is currently supported by selecting the I2C_2BYTES_ADDR configuration parameter, in order to address the 1 Mbit (17 bit) memory array the 17th address bit (A16) is passed along with the device address (in i2c_eeprom.c):

```
SetWord16(I2C_TAR_REG, i2c_dev_address | ((address_to_send & 0x10000) >> 16));
```

It is recommended to add a new preprocessor constant (in `i2c_eeprom.h`):

```
enum I2C_ADRESS_BYTES_COUNT{
  I2C_1BYTE_ADDR,
  I2C_2BYTES_ADDR,
  I2C_3BYTES_ADDR,
  I2C_MY_CUSTOM_ADDR_SIZE,
};
```

and to adapt the `i2c_send_address()` function accordingly:

```
void i2c_send_address(uint32_t address_to_send)
{
       if (mem_address_size == I2C_MY_CUSTOM_ADDR_SIZE)
       {
              // Code supporting the new device
       }
       else
       {
              // Existing code for the rest of the devices
       }
}
```

Appendix C Application Software APIs

C.1 Mid Layer API

The files for the Mid Layer API are stored in the folder: `sdk\app_modules\api`.

File	Description
`app_mid.h`	API functions

C.1.1 app_disconnect_msg_create

Function name	`__INLINE struct gapc_disconnect_cmd* app_disconnect_msg_create(uint8_t connection_idx)`
Function description	Allocate a disconnect message for a specific connection.
Parameters	`connection_idx` The connection index.
Return values	Returns the pointer to the disconnect command message.
Notes	None

C.1.2 app_disconnect_msg_send

Function name	`__INLINE void app_disconnect_msg_send(struct gapc_disconnect_cmd *cmd)`
Function description	Send the disconnect message.
Parameters	`cmd` Pointer to the disconnect message.
Return values	None
Notes	The function only calls `ke_msg_send()`.

C.1.3 app_connect_cfm_msg_create

Function name	`__INLINE struct gapc_connection_cfm* app_connect_cfm_msg_create(uint8_t connection_idx)`
Function description	Allocate a connect confirmation message for a specific connection.
Parameters	`connection_idx` The connection index.
Return values	Returns the pointer to the connect confirm message.
Notes	None

C.1.4 app_connect_cfm_msg_send

Function name	`__INLINE void app_connect_cfm_msg_send(struct gapc_connection_cfm*cmd)`
Function description	Send the connect confirmation message.
Parameters	`Cmd` Pointer to the connect confirmation message.
Return values	None
Notes	The function only calls `ke_msg_send()`.

C.1.5 app_advertise_start_msg_create

Function name	**__INLINE struct gapm_start_advertise_cmd* app_advertise_start_msg_create(void)**
Function description	Allocate an advertise start message.
Parameters	None
Return values	Returns the pointer to the advertise start message.
Notes	None

C.1.6 app_advertise_start_msg_send

Function name	**__INLINE void app_advertise_start_msg_send(struct gapm_start_advertise_cmd* cmd)**
Function description	Send the advertise start message.
Parameters	Cmd Pointer to the advertise start message.
Return values	None
Notes	The function only calls ke_msg_send().

C.1.7 app_gapm_cancel_msg_create

Function name	**__INLINE struct gapm_cancel_cmd* app_gapm_cancel_msg_create(void)**
Function description	Allocate a gap manager cancellation message.
Parameters	None
Return values	Returns the pointer to the gap manager cancellation message.
Notes	None

C.1.8 app_gapm_cancel_msg_send

Function name	**__INLINE void app_gapm_cancel_msg_send(struct gapm_cancel_cmd* cmd)**
Function description	Send the gap manager cancellation message
Parameters	Cmd Pointer to the gap manager cancellation message
Return values	None
Notes	The function only calls ke_msg_send().

C.1.9 app_advertise_stop_msg_create

Function name	**__INLINE struct gapm_cancel_cmd* app_advertise_stop_msg_create(void)**
Function description	Allocate a gap manager cancellation message.
Parameters	None
Return values	Returns the pointer to the gap manager cancellation message.
Notes	Wrapper of app_gapm_cancel_msg_create() to provide a more meaningful name.

C.1.10 app_advertise_stop_msg_send

Function name	**__INLINE void app_advertise_stop_msg_send(struct gapm_cancel_cmd* cmd)**
Function description	Send the gap manager cancellation message.
Parameters	cmd Pointer to the gap manager cancellation message
Return values	None
Notes	Wrapper of app_gapm_cancel_msg_send() to provide a more meaningful name. The function only calls ke_msg_send().

C.1.11 app_param_update_msg_create

Function name	**__INLINE struct gapc_param_update_cmd* app_param_update_msg_create(uint8_t connection_idx)**
Function description	Allocate a parameter update message for a given connection.
Parameters	connection_idx The connection index.
Return values	Returns the pointer to the parameter update message message.
Notes	None

C.1.12 app_advertise_stop_msg_send

Function name	**__INLINE void app_advertise_stop_msg_send(struct gapm_cancel_cmd* cmd)**
Function description	Send the gap manager cancellation message.
Parameters	cmd Pointer to the gap manager cancellation message
Return values	None
Notes	Wrapper of app_gapm_cancel_msg_send() to provide a more meaningful name. The function only calls ke_msg_send().

C.1.13 app_connect_start_msg_create

Function name	**__INLINE struct gapm_start_connection_cmd* app_connect_start_msg_create(void)**
Function description	Allocate a start connection message.
Parameters	None
Return values	Returns the pointer to the start connection message.
Notes	The start connection message is used from a central to initiate a connection with a peripheral.

C.1.14 app_connect_start_msg_send

Function name	**__INLINE void app_connect_start_msg_send(struct gapm_start_connection_cmd* cmd)**
Function description	Send the start connection message.
Parameters	cmd Pointer to the start connection message.
Return values	None
Notes	The start connection message is used from a central to initiate a connection with a peripheral. The function only calls ke_msg_send().

C.1.15 app_gapm_configure_msg_create

Function name	`__INLINE struct gapm_set_dev_config_cmd* app_gapm_configure_msg_create(void)`
Function description	Allocate a gap manager configuration message.
Parameters	None
Return values	Returns the pointer to the gap manager configuration message.
Notes	None

C.1.16 app_gapm_configure_msg_send

Function name	`__INLINE void app_gapm_configure_msg_send(struct gapm_set_dev_config_cmd* cmd)`
Function description	Send the gap manager configuration message.
Parameters	`Cmd` Pointer to the gap manager configuration message.
Return values	None
Notes	The function only calls `ke_msg_send()`.

C.1.17 app_gapc_bond_cfm_msg_create

Function name	`__INLINE struct gapc_bond_cfm* app_gapc_bond_cfm_msg_create (uint8_t connection_idx)`
Function description	Allocate a gap bond confirmation message for a given connection.
Parameters	`connection_idx` The connection index.
Return values	Returns the pointer to the gap bond confirmation message.
Notes	None

C.1.18 app_gapc_bond_cfm_msg_send

Function name	`__INLINE void app_gapc_bond_cfm_msg_send (struct gapc_bond_cfm* cmd)`
Function description	Send the gap bond confirmation message.
Parameters	`cmd` Pointer to the gap bond confirmation message.
Return values	None
Notes	The function only calls `ke_msg_send()`.

C.1.19 app_gapc_bond_cfm_pairing_rsp_msg_create

Function name	`__INLINE struct gapc_bond_cfm* app_gapc_bond_cfm_pairing_rsp_msg_create(uint8_t connection_idx)`
Function description	Allocate a gap bond pairing response message for a given connection.
Parameters	`connection_idx` The connection index.
Return values	Returns the pointer to the gap bond pairing response message.
Notes	None

C.1.20 app_gapc_bond_cfm_pairing_rsp_msg_send

Function name	**__INLINE void app_gapc_bond_cfm_pairing_rsp_msg_send (struct gapc_bond_cfm* cmd)**
Function description	Send the gap bond pairing response message.
Parameters	cmd Pointer to the gap bond pairing response message.
Return values	None
Notes	The function only calls ke_msg_send().

C.1.21 app_gapc_bond_cfm_tk_exch_msg_create

Function name	**__INLINE struct gapc_bond_cfm* app_gapc_bond_cfm_tk_exch_msg_create(uint8_t connection_idx)**
Function description	Allocate a temporary key exchange message for a given connection.
Parameters	connection_idx The connection index.
Return values	Returns the pointer to the gap bond confirmation temporary key exchange message.
Notes	None

C.1.22 app_gapc_bond_cfm_tk_exch_msg_send

Function name	**__INLINE void app_gapc_bond_cfm_tk_exch_msg_send (struct gapc_bond_cfm* cmd)**
Function description	Send the temporary key exchange message.
Parameters	cmd Pointer to bond confirmation temporary key exchange message.
Return values	None
Notes	The function only calls ke_msg_send().

C.1.23 app_gapc_bond_cfm_csrk_exch_msg_create

Function name	**__INLINE struct gapc_bond_cfm* app_gapc_bond_cfm_csrk_exch_msg_create(uint8_t connection_idx)**
Function description	Allocate a Connection Signature Resolving Key (CSRK) exchange message for a given connection.
Parameters	connection_idx The connection index.
Return values	Returns the pointer to the CSRK exchange message.
Notes	None

C.1.24 app_gapc_bond_cfm_csrk_exch_msg_send

Function name	**__INLINE void app_gapc_bond_cfm_csrk_exch_msg_send (struct gapc_bond_cfm* cmd)**
Function description	Send the CSRK key exchange message.
Parameters	cmd Pointer to the CSRK exchange message.
Return values	None
Notes	The function only calls ke_msg_send().

C.1.25 app_gapc_bond_cfm_ltk_exch_msg_create

Function name	`__INLINE struct gapc_bond_cfm* app_gapc_bond_cfm_ltk_exch_msg_create(uint8_t connection_idx)`
Function description	Allocate a Long Term Key (LTK) exchange message for a given connection.
Parameters	`connection_idx` The connection index.
Return values	Returns the pointer to the LTK exchange message.
Notes	None

C.1.26 app_gapc_bond_cfm_ltk_exch_msg_send

Function name	`__INLINE void app_gapc_bond_cfm_ltk_exch_msg_send (struct gapc_bond_cfm* cmd)`
Function description	Send the LTK key exchange message.
Parameters	`cmd` Pointer to the LTK exchange message.
Return values	None
Notes	The function only calls `ke_msg_send()`.

C.1.27 app_gapc_encrypt_cfm_msg_create

Function name	`__INLINE struct gapc_encrypt_cfm* app_gapc_encrypt_cfm_msg_create (uint8_t connection_idx)`
Function description	Allocate an encryption confirmation message for a given connection.
Parameters	`connection_idx` The connection index.
Return values	Returns the pointer to the encryption confirmation message.
Notes	None

C.1.28 app_gapc_encrypt_cfm_msg_send

Function name	`__INLINE void app_gapc_encrypt_cfm_msg_send (struct gapc_bond_cfm* cmd)`
Function description	Send the encryption confirmation message.
Parameters	`cmd` Pointer to the encryption confirmation message.
Return values	None
Notes	The function only calls `ke_msg_send()`.

C.1.29 app_gapc_security_request_msg_create

Function name	`__INLINE struct gapc_security_cmd* app_gapc_security_request_msg_create (uint8_t connection_idx, enum gap_auth auth)`
Function description	Allocate a security request message for a given connection.
Parameters	`connection_idx` The connection index. `auth` The desired authentication level.
Return values	Returns the pointer to the encryption confirmation message.
Notes	None

C.1.30 app_gapc_security_request_msg_send

Function name	**__INLINE void app_gapc_security_request_msg_send (struct gapc_security_cmd* cmd)**
Function description	Send the security request message.
Parameters	cmd Pointer to the security request message.
Return values	None
Notes	The function only calls ke_msg_send().

C.1.31 app_gapm_reset_msg_create

Function name	**__INLINE struct gapm_reset_cmd* app_gapm_reset_msg_create(void)**
Function description	Allocate a gap manager reset message.
Parameters	None
Return values	Returns the pointer to the gap manager reset message.
Notes	None

C.1.32 app_gapm_reset_msg_send

Function name	**__INLINE void app_gapm_reset_msg_send (struct gapm_reset_cmd* cmd)**
Function description	Send the gap manager reset message.
Parameters	cmd Pointer to the gap manager reset message.
Return values	None
Notes	The function only calls ke_msg_send().

C.1.33 app_gapm_reset_op

Function name	**__INLINE void app_gapm_reset_op (void)**
Function description	Reset the gap manager reset.
Parameters	None
Return values	None
Notes	None

C.1.34 app_disconnect_op

Function name	**__INLINE void app_disconnect_op (uint8_t connection_id, uint8_t reason)**
Function description	Disconnect from a specific connection for a specific reason.
Parameters	connection_id The id of the given connection. Reason The reason for the disconnection. Could be one of the following: CO_ERROR_AUTH_FAILURE, CO_ERROR_REMOTE_USER_TERM_CON, CO_ERROR_REMOTE_DEV_TERM_LOW_RESOURCES, CO_ERROR_REMOTE_DEV_POWER_OFF, CO_ERROR_UNSUPPORTED_REMOTE_FEATURE, CO_ERROR_PAIRING_WITH_UNIT_KEY_NOT_SUP, CO_ERROR_UNACCEPTABLE_CONN_INT.
Return values	None
Notes	None

C.1.35 app_connect_confirm_op

Function name	**__INLINE void app_connect_confirm_op(uint8_t connection_id, enum gap_auth auth, enum gap_authz authorize)**
Function description	Confirm a connection.
Parameters	connection_id The id of the given connection. Auth The authentication requirements. authorize _id The authorization requirements.
Return values	None
Notes	None

C.1.36 app_advertise_undirected_start_op

Function name	**__INLINE void app_advertise_undirected_start_op (enum gapm_own_addr_src address_src_type, uint16_t interval, uint8_t channel_map, enum gap_adv_mode advertise_mode,enum adv_filter_policy adv_filt_policy, uint8_t* advertise_data, uint8_t advertise_data_len, uint8_t* scan_response_data, uint8_t scan_response_data_len)**
Function description	Start an advertise undirected operation.
Parameters	address_src_type The source address type used during the advertise operation. Interval The advertise interval. channel_map The channels used during the advertise operation. advertise_mode The advertising mode: GAP_NON_DISCOVERABLE, GAP_GEN_DISCOVERABLE, GAP_LIM_DISCOVERABLE, GAP_BROADCASTER_MODE adv_filt_policy The advertising filter policy: ADV_ALLOW_SCAN_ANY_CON_ANY, ADV_ALLOW_SCAN_WLST_CON_ANY, ADV_ALLOW_SCAN_ANY_CON_WLST, ADV_ALLOW_SCAN_WLST_CON_WLST advertise_data Pointer to an array with the advertise data. advertise_data_len The length of the advertise data. scan_response_data Pointer to an array with the scan response data. scan_response_data_len The length of the scan response data.
Return values	None
Notes	None

C.1.37 app_advertise_directed_start_op

Function name	**__INLINE void app_advertise_directed_start_op (enum gapm_own_addr_src address_src_type, uint16_t interval, uint8_t channel_map, enum address_type target_address_type, uint8_t* target_address)**
Function description	Start an advertise directed operation.
Parameters	address_src_type The source address type used during the advertise operation. Interval The advertise interval. channel_map The channels used during the advertise operation. target_address_type Address type of the target device. target_address Address of the target device.
Return values	None
Notes	None

C.1.38 app_advertise_stop_op

Function name	`__INLINE void app_advertise_stop_op(void)`
Function description	Stop the current advertise operation.
Parameters	None
Return values	None
Notes	None

C.1.39 app_param_update_op

Function name	`__INLINE void app_param_update_op(uint8_t connection_idx, uint16_t intv_min, uint16_t intv_max, uint16_t latency, uint16_t supervision_time_out, uint16_t connection_event_len_min, uint16_t connection_event_len_max)`
Function description	Send a parameter update operation (parameters in 1.25 ms time slots).
Parameters	`connection_idx` The id of the connection. `latency` The slave latency measured in connection event periods. `intv_min` The new preferred minimum connection interval measured in 1.25 ms slots. `intv_max` The new preferred maximum connection interval measured in 1.25 ms slots. `connection_event_len_min` The new preferred minimum connection event length measured in 1.25 ms slots. `connection_event_len_max` The new preferred maximum connection event length measured in 1.25 ms slots.
Return values	None
Notes	None

C.1.40 app_param_update_op_us

Function name	`__INLINE void app_param_update_op_us(uint8_t connection_idx, uint32_t intv_min_us, uint32_t intv_max_us, uint16_t latency, uint32_t supervision_time_out_us, uint32_t connection_event_len_min_us, uint32_t connection_event_len_max_us)`
Function description	Send a parameter update operation (parameters in microseconds).
Parameters	`connection_idx` The id of the connection. `Latency` The slave latency measured in connection event periods. `intv_min_us` The new preferred minimum connection interval measured in microseconds. `intv_max_us` The new preferred maximum connection interval measured in microseconds. `connection_event_len_min_us` The new preferred minimum connection event length measured in micro seconds. `connection_event_len_max_us` The new preferred maximum connection event length measured in micro seconds.
Return values	None
Notes	None

C.1.41 app_gapm_configure_op

Function name	**__INLINE void app_gapm_configure_op (enum gap_role role, uint8_t* irk, uint16_t appearance, enum gapm_write_att_perm appearance_write_perm, enum gapm_write_att_perm name_write_perm, uint16_t max_mtu, uint16_t connection_intv_min, uint16_t connection_intv_max, uint16_t connection_latency, uint16_t supervision_timeout, uint8_t flags)**
Function description	Configure the gap manager (parameters in 1.25 ms time slots).

Parameters	`role` The role of the device:
	`GAP_NO_ROLE, GAP_OBSERVER_SCA,`
	`GAP_BROADCASTER_ADV,`
	`GAP_CENTRAL_MST,`
	`GAP_PERIPHERAL_SLV`
	`irk` Pointer to an array that contains the device Identity Root Key (IRK).
	`appearance` The device appearance.
	`appearance_write_perm` Appearance write permission requirement.
	`name_write_perm` Name write permission requirement.
	`max_mtu` Maximum `mtu` supported.
	`connection_intv_min` Preferred minimum connection interval measured in 1.25 ms slots.
	`connection_intv_max` Preferred maximum connection interval measured in 1.25 ms slots.
	`connection_latency` Preferred slave latency measured in connection events.
	`supervision_timeout` Preferred supervision timeout measured in 10 ms slots.
	`flags` Privacy settings bit field (0b1 = enabled, 0b0 = disabled)
	`[bit 0]` Privacy Support
	`[bit 1]` Multiple Bond Support (Peripheral only). Read-only if enabled.
	`[bit 2]` Reconnection address visible.

Return values	None
Notes	None

C.1.42 app_gapm_configure_op_us

Function name	**__INLINE void app_gapm_configure_op_us (enum gap_role role, uint8_t* irk, uint16_t appearance, enum gapm_write_att_perm appearance_write_perm, enum gapm_write_att_perm name_write_perm, uint16_t max_mtu, uint32_t connection_intv_min_us, uint32_t connection_intv_max_us, uint16_t connection_latency, uint32_t supervision_timeout_us, uint8_t flags)**
Function description	Configure the gap manager (parameters in microseconds).
Parameters	`role` The role of the device: `GAP_NO_ROLE`, `GAP_OBSERVER_SCA`, `GAP_BROADCASTER_ADV`, `GAP_CENTRAL_MST`, `GAP_PERIPHERAL_SLV` `irk` Pointer to an array that contains the device Identity Root Key (IRK). `appearance` The device appearance. `appearance_write_perm` Appearance write permission requirement. `name_write_perm` Name write permission requirement. `max_mtu` Maximum `mtu` supported. `connection_intv_min_us` Preferred minimum connection interval measured in microseconds. `connection_intv_max_us` Preferred maximum connection interval measured in microseconds. `connection_latency` Preferred slave latency measured in connection events. `supervision_timeout_us` Preferred supervision timeout measured in microseconds. `flags` Privacy settings bit field (0b1 = enabled, 0b0 = disabled) `[bit 0]` Privacy Support `[bit 1]` Multiple Bond Support (Peripheral only). Read-only if enabled. `[bit 2]` Reconnection address visible.
Return values	None
Notes	None

C.1.43 app_security_request_op

Function name	**__INLINE void app_security_request_op(uint8_t connection_idx, enum gap_auth auth)**
Function description	Send a security request.
Parameters	`connection_idx` The id of the connection. `auth` The authentication requirements.
Return values	None
Notes	None

C.1.44 app_gapc_bond_cfm_pairing_rsp_op

Function name	**__INLINE void app_gapc_bond_cfm_pairing_rsp_op (uint8_t connection_idx, enum gap_io_cap io_capabilities, enum gap_oob oob, enum gap_auth authentication, uint8_t key_size, enum gap_kdist initiator_key_dist, enum gap_kdist responder_key_dist, enum gap_sec_req security_requirements)**
Function description	Send a security request.
Parameters	connection_idx The id of the connection. io_capabilities Device capabilities: GAP_IO_CAP_DISPLAY_ONLY, GAP_IO_CAP_DISPLAY_YES_NO, GAP_IO_CAP_KB_ONLY, GAP_IO_CAP_NO_INPUT_NO_OUTPUT, GAP_IO_CAP_KB_DISPLAY. oob Out of band info: GAP_OOB_AUTH_DATA_NOT_PRESENT, GAP_OOB_AUTH_DATA_PRESENT authentication The authentication requirements. key_size The key size. initiator_key_dist Initiator key distriburion flags: GAP_KDIST_NONE, GAP_KDIST_ENCKEY, GAP_KDIST_IDKEY, GAP_KDIST_SIGNKEY responder_key_dist Responder key distriburion flags: GAP_KDIST_NONE, GAP_KDIST_ENCKEY, GAP_KDIST_IDKEY, GAP_KDIST_SIGNKEY security_requirements Security definition.
Return values	None
Notes	None

C.1.45 app_gapc_bond_cfm_tk_exch_op

Function name	**__INLINE void app_gapc_bond_cfm_tk_exch_op (uint8_t connection_idx, uint8_t* temporary_key)**
Function description	Exchange the temporary key.
Parameters	connection_idx The id of the connection. temporary_key Array containing the temporary key.
Return values	None
Notes	None

C.1.46 app_gapc_bond_cfm_csrk_exch_op

Function name	**__INLINE void app_gapc_bond_cfm_csrk_exch_op (uint8_t connection_idx, uint8_t* csrk)**
Function description	Exchange the CSRK key.
Parameters	connection_idx The id of the connection. csrk Array containing the CSRK key.
Return values	None
Notes	None

C.1.47 app_gapc_bond_cfm_ltk_exch_op

Function name	**__INLINE void app_gapc_bond_cfm_ltk_exch_op (uint8_t connection_idx, uint8_t* long_term_key, uint8_t encryption_key_size, uint8_t* random_number, uint16_t encryption_diversifier)**
Function description	Exchange the LTK key and parameters.
Parameters	connection_idx The id of the connection. long_term_key Array containing the long term key. encryption_key_size Encryption key size. random_number Random number. encryption_diversifier Encryption diversifier.
Return values	None
Notes	None

C.1.48 app_gapc_encrypt_cfm_op

Function name	**__INLINE void app_gapc_encrypt_cfm_op (uint8_t connection_idx, bool found, uint8_t key_size, uint8_t* long_term_key)**
Function description	Confirm the success of the encryption.
Parameters	connection_idx The id of the connection. found Confirm that the entry has been found. key_size Size of the key. long_term_key The long term key.
Return values	None
Notes	None

C.2 Easy API

The files for the Easy API are stored in the folder: `sdk\app_modules\api`.

C.2.1 conhdl_to_conidx

Function name	**`__INLINE int8_t conhdl_to_conidx (uint16_t conhdl)`**
Function description	Returns the connection index from the connection handle.
Parameters	`conhdl` The id of the connection handle.
Return values	The value of the connection index.
Notes	Macro for the `gapc_get_conidx`.

C.2.2 conidx_to_conhdl

Function name	**`__INLINE int16_t conidx_to_conhdl (uint8_t conidx)`**
Function description	Returns the connection handle value from the connection index.
Parameters	`conidx` The id of the connection index
Return values	The value of the connection handle.
Notes	Macro for the `gapc_get_conhdl`.

C.2.3 app_easy_gap_disconnect

Function name	**`void app_easy_gap_disconnect (uint8_t connection_idx)`**
Function description	Send the BLE disconnect command.
Parameters	`connection_idx` The id of the connection index.
Return values	None
Notes	None

C.2.4 app_easy_gap_confirm

Function name	**`void app_easy_gap_confirm (uint8_t connection_idx, enum gap_auth auth, enum gap_authz authorize)`**
Function description	Send the gap confirmation message.
Parameters	`connection_idx` The id of the connection index. `auth` The authentication requirements. `authorize` The authorization requirements.
Return values	None
Notes	None

C.2.5 app_easy_gap_undirected_advertise_start

Function name	**void app_easy_gap_undirected_advertise_start (void)**
Function description	Start an undirected advertise operation according to the settings of the active message.
Parameters	None
Return values	None
Notes	None

C.2.6 app_easy_gap_directed_advertise_start

Function name	**void app_easy_gap_directed_advertise_start (void)**
Function description	Start a directed advertise operation according to the settings of the active message.
Parameters	None
Return values	None
Notes	None

C.2.7 app_easy_gap_non_connectable_advertise_start

Function name	**void app_easy_gap_non_connectable_advertise_start (void)**
Function description	Start a non-connectable advertise operation to the settings of the active message.
Parameters	None
Return values	None
Notes	None

C.2.8 app_easy_gap_advertise_stop

Function name	**void app_easy_gap_advertise_stop (void)**
Function description	Stop the active advertise operation.
Parameters	None
Return values	None
Notes	None

C.2.9 app_easy_gap_undirected_advertise_with_timeout_start

Function name	**void app_easy_gap_undirected_advertise_with_timeout_start (uint16_t delay, void (*timeout_callback)(void))**
Function description	Start an undirected advertise operation according to the settings of the active message for a specific time. When the advertise finishes callback a user function.
Parameters	`delay` The maximum duration of the advertise operation. `timeout_callback` The callback to call if the advertise operation times out.
Return values	None
Notes	None

C.2.10 app_easy_gap_advertise_with_timeout_stop

Function name	`void app_easy_gap_advertise_with_timeout_stop (void)`
Function description	Stop the undirected advertise and the timeout timer.
Parameters	None
Return values	None
Notes	None

C.2.11 app_easy_gap_undirected_advertise_get_active

Function name	`struct gapm_start_advertise_cmd* app_easy_gap_undirected_advertise_get_active (void)`
Function description	Get the active advertise message. If None exists allocate a new one and fill it with the undirected related configuration from `user_config.h`.
Parameters	None
Return values	The active message.
Notes	None

C.2.12 app_easy_gap_directed_advertise_get_active

Function name	`struct gapm_start_advertise_cmd* app_easy_gap_directed_advertise_get_active (void)`
Function description	Get the active advertise message. If None exists allocate a new one and fill it with the directed related configuration from `user_config.h`.
Parameters	None
Return values	The active message.
Notes	None

C.2.13 app_easy_gap_param_update_start

Function name	`void app_easy_gap_param_update_start (uint8_t connection_idx)`
Function description	Start a parameter update sequence according to the settings of the active message for a specific connection.
Parameters	`connection_idx` The id of the connection.
Return values	None
Notes	None

C.2.14 app_easy_gap_param_update_get_active

Function name	`struct gapc_param_update_cmd* app_easy_gap_param_update_get_active (uint8_t connection_idx)`
Function description	Get the active parameter update message. If None exists allocate a new one according to the setting in the `user_config.h` for a specific connection.
Parameters	`connection_idx` The id of the connection.
Return values	The active message.
Notes	None

C.2.15 app_easy_gap_start_connection_to

Function name	**void app_easy_gap_start_connection_to (void)**
Function description	Start a new connection with a peripheral.
Parameters	None
Return values	None
Notes	None

C.2.16 app_easy_gap_start_connection_to_set

Function name	**void app_easy_gap_start_connection_to_set (uint8_t peer_addr_type, uint8_t *peer_addr, uint16_t intv)**
Function description	Set the parameters of a `connection_to` operation. Set the peer address, its type and the connection interval to use.
Parameters	`peer_addr_type` The peer address type. `peer_addr` The peer address. `intv` The connection interval to use.
Return values	None
Notes	None

C.2.17 app_easy_gap_start_connection_to_get_active

Function name	**struct gapm_start_connection_cmd* app_easy_gap_start_connection_to_get_active (void)**
Function description	Get the active message of the `connection_to` operation. If None exists create a new empty one.
Parameters	None
Return values	The active message.
Notes	None

C.2.18 app_easy_gap_dev_config_get_active

Function name	**struct gapm_set_dev_config_cmd* app_easy_gap_dev_config_get_active (void)**
Function description	Get the active gap configure message. If None exists create a new one according to the settings in `user_config.h`.
Parameters	None
Return values	The active message.
Notes	None

C.2.19 app_easy_gap_dev_configure

Function name	**void app_easy_gap_dev_configure (void)**
Function description	Configure the gap manager according to the settings of the active message.
Parameters	None
Return values	None
Notes	None

C.2.20 app_easy_security_pairing_rsp_get_active

Function name	**struct gapc_bond_cfm* app_easy_security_pairing_rsp_get_active (uint8_t connection_idx)**
Function description	Get the pairing response active message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	The active message.
Notes	None

C.2.21 app_easy_security_tk_get_active

Function name	**struct gapc_bond_cfm* app_easy_security_tk_get_active (uint8_t connection_idx)**
Function description	Get the temporary key exchange active message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	The active message.
Notes	None

C.2.22 app_easy_security_csrk_get_active

Function name	**struct gapc_bond_cfm* app_easy_security_csrk_get_active (uint8_t connection_idx)**
Function description	Get the CSRK exchange active message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	The active message.
Notes	None

C.2.23 app_easy_security_ltk_exch_get_active

Function name	**struct gapc_bond_cfm* app_easy_security_ltk_exch_get_active (uint8_t connection_idx)**
Function description	Get the LTK exchange active message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	The active message.
Notes	None

C.2.24 app_easy_security_encrypt_cfm_get_active

Function name	**struct gapc_encrypt_cfm* app_easy_security_encrypt_cfm_get_active (uint8_t connection_idx)**
Function description	Get the encryption confirmation message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	The active message.
Notes	None

C.2.25 app_easy_security_set_tk

Function name	**void app_easy_security_set_tk (uint8_t connection_idx, uint8_t *key, uint8_t keylen)**
Function description	Set the temporary key in the TK exchange message for a specific connection..
Parameters	connection_idx The id of the connection. key The actual key. keylen The length of the key
Return values	None
Notes	None

C.2.26 app_easy_security_set_ltk_exch_from_sec_env

Function name	**void app_easy_security_set_ltk_exch_from_sec_env (uint8_t connection_idx)**
Function description	Fill in the LTK settings for the app_sec_env structure for the specific id.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.27 app_easy_security_set_ltk_exch

Function name	**void app_easy_security_set_ltk_exch (uint8_t connection_idx, uint8_t* long_term_key, uint8_t encryption_key_size, uint8_t* random_number, uint16_t encryption_diversifier)**
Function description	Set all the LTK settings of the active LTK message.
Parameters	connection_idx The id of the connection. long_term_key The long term key. encryption_key_size The encryption size. random_number The random number to use. encryption_diversifier The encryption diversifier.
Return values	None
Notes	None

C.2.28 app_easy_security_set_encrypt_req_valid

Function name	**void app_easy_security_set_encrypt_req_valid (uint8_t connection_idx)**
Function description	Set the encrypt request message as valid for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.29 app_easy_security_set_encrypt_req_invalid

Function name	**void app_easy_security_set_encrypt_req_invalid (uint8_t connection_idx)**
Function description	Set the encrypt request message as invalid for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.30 app_easy_security_send_pairing_rsp

Function name	**void app_easy_security_send_pairing_rsp (uint8_t connection_idx)**
Function description	Send the pairing response for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.31 app_easy_security_tk_exch

Function name	**void app_easy_security_tk_exch (uint8_t connection_idx, uint8_t *key, uint8_t length)**
Function description	Send the TK exchange message for a specific connection.
Parameters	connection_idx The id of the connection. *key Pointer to the key that will be sent via the TK exchange message. length Length of the pass key in octets.
Return values	None
Notes	The # key can be either a 6-digit (4 octets) pass key or an OOB provided key. The maximum size of the OOB key is 128-bit (16 octets).

C.2.32 app_easy_security_csrk_exch

Function name	**void app_easy_security_csrk_exch (uint8_t connection_idx)**
Function description	Send the CSRK exchange message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.33 app_easy_security_ltk_exch

Function name	**void app_easy_security_ltk_exch (uint8_t connection_idx)**
Function description	Send the LTK exchange message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.34 app_easy_security_encrypt_cfm

Function name	**void app_easy_security_encrypt_cfm (uint8_t connection_idx)**
Function description	Send the encrypt confirmation message for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.35 app_easy_security_request_get_active

Function name	**struct gapc_security_cmd* app_easy_security_request_get_active (uint8_t connection_idx)**
Function description	Get the active security request message.
Parameters	connection_idx The id of the connection.
Return values	The active message.
Notes	None

C.2.36 app_easy_security_request

Function name	**void app_easy_security_request (uint8_t connection_idx)**
Function description	Issue a security request for a specific connection.
Parameters	connection_idx The id of the connection.
Return values	None
Notes	None

C.2.37 app_easy_timer

Function name	**timer_hnd app_easy_timer (const uint16_t delay, void(*fn)(void))**
Function description	Issue a timer with specific delay and call a callback after the timer expires.
Parameters	delay The timer delay in timer slots (10 ms). fn The callback to call.
Return values	The timer handler to use for cancelation or modification of the timer.
Notes	None

C.2.38 app_easy_timer_cancel

Function name	**void app_easy_timer_cancel (const timer_hnd timer_id)**
Function description	Cancel a specific timer.
Parameters	timer_id The handler of the timer to cancel.
Return values	None
Notes	None

C.2.39 app_easy_timer_modify

Function name	**timer_hnd app_easy_timer_modify (const timer_hnd timer_id, const uint16_t delay)**
Function description	Modify the callback and delay of a specifc timer. The timer will be set with the new additional delay.
Parameters	delay The timer delay in timer slots (10 ms). fn The callback to call.
Return values	Returns the modified timer handler.
Notes	None

C.2.40 app_easy_timer_cancel_all

Function name	**void app_easy_timer_cancel_all (void)**
Function description	Cancel all active timers.
Parameters	None
Return values	None
Notes	None

Appendix D Supporting Custom Profiles

The SDK 5.0.2 and later supports the creation of Custom profiles defined by the user. The maximum number of the supported Custom profiles is limited to two.

D.1 Custom Profile API

The following sections include the Custom profile functions API.

D.1.1 app_custs1_create_db

Function name	**void app_custs1_create_db (void)**
Function description	Create custom1 profile database.
Parameters	None
Return values	None
Notes	This function has to be placed as an entry in the cust_prf_funcs[] array, defined in configuration header file user_custs_config.h. The user can override this entry and place the callback reference of his function implementation.

D.1.2 app_custs2_create_db

Function name	**void app_custs2_create_db (void)**
Function description	Create custom2 profile database.
Parameters	void
Return values	void
Notes	This function has to be placed as an entry in the cust_prf_funcs[] array, defined in configuration header file user_custs_config.h. The user can override this entry and place the callback reference of his function implementation.

D.1.3 app_custs1_enable

Function name	**void app_custs1_enable (uint16_t conhdl)**
Function description	Enable custom1 profile.
Parameters	conhdl Connection handle.
Return values	None
Notes	This function has to be placed as an entry in the cust_prf_funcs[] array, defined in configuration header file user_custs_config.h. The user can override this entry and place the callback reference of his function implementation.

D.1.4 app_custs2_enable

Function name	**void app_custs2_enable(uint16_t conhdl)**
Function description	Enable custom2 profile
Parameters	conhdl Connection handle.
Return values	None/
Notes	This function has to be placed as an entry in the cust_prf_funcs[] array, defined in configuration header file ‘user_custs_config.h’. The user can override this entry and place the callback reference of his function implementation.

D.2 Configuration Header Files

The following configuration header files have to be modified in order to include a Custom profile in the user application.

- `user_profiles_config.h`, declares the header file, either custs1.h or cust2.h, of the Custom profile to be included in the user application. If the respective header file is not included, any other configuration relative to the Custom profile will not have any effect in the user application.
- `user_custs_config.h`, defines the structure of the Custom profile database and the cust_prf_funcs[] array, which contains the Custom profile API functions calls.
- `user_modules_config.h`, the following preprocessor definition has to be modified accordingly to the user application needs. For example:

  `#define EXCLUDE_DLG_CUSTS1 (0)` The Custom1 application profile is included. The SDK takes care of the Custom1 application profile message handling.

  `#define EXCLUDE_DLG_CUSTS1 (1)` The Custom1 profile application is excluded. The user application has to take care of the Custom1 application profile message handling.

The following picture marks the aforementioned configuration header files in the Keil project layout.

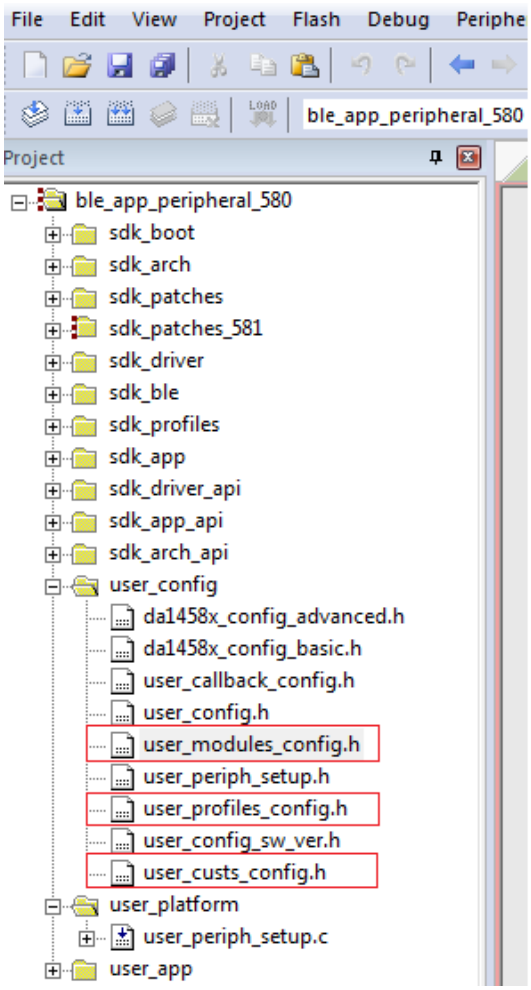


Figure 26: Custom Profile User Configuration

Appendix E Advanced Features APIs

E.1 How to Select the Low Power Clock

Support of the RCX clock as low power clock source is added in SDK v5.0.2.

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

```
#define CFG_LP_CLK   0x00
```

Where:

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

A calibration mechanism has been developed to measure the RCX clock frequency changes over temperature. This mechanism consists of the functions `calibrate_rcx20()` and `read_rcx_freq()`. Both functions are implemented in `sdk\platform\arch\main\arch_system.c`.

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

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

E.2 True Random Number Generator (TRNG)

This section describes the function `trng_acquire()` which generates a 128-bit random number. The function is implemented in the file `trng.c` in the folder: `sdk\platform\driver\trng`.

E.2.1 trng_acquire

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

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

```
save_TEST_CTRL_REG=GetWord16(TEST_CTRL_REG);
save_RF_ENABLE_CONFIG1_REG=GetWord16(RF_ENABLE_CONFIG1_REG);   // LNA off
save_RF_ENABLE_CONFIG2_REG=GetWord16(RF_ENABLE_CONFIG2_REG);   // Mixer off
save_RF_DC_OFFSET_CTRL1_RE=GetWord16(RF_DC_OFFSET_CTRL1_REG);  // Fixed DC offset
     compensation values for I and Q
save_RF_DC_OFFSET_CTRL2_REG=GetWord16(RF_DC_OFFSET_CTRL2_REG); // Use the manual
     DC offset compensation values
save_RF_ENABLE_CONFIG4_REG=GetWord16(RF_ENABLE_CONFIG4_REG);   // VCO_LDO_EN=0,
     MD_LDO_EN=0. You need this for more isolation from the RF input
save_RF_ENABLE_CONFIG14_REG=GetWord16(RF_ENABLE_CONFIG14_REG); // LOBUF_RXIQ_EN=0,
     DIV2_EN=0. This increases the noise floor for some reason. So you get more
     entropy. Need to understand it and then decide...
save_RF_SPARE1_REG=GetWord16(RF_SPARE1_REG);  // Set the IFF in REAL transfer
     function, to remove I-Q correlation. But it affects the DC offsets!
save_RF_AGC_CTRL2_REG=GetWord16(RF_AGC_CTRL2_REG);  // AGC=0 i.e. max RX gain
```

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

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

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

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

E.3 DCDC_VBAT3V API

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

E.3.1 syscntl_set_dcdc_vbat3v_level

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

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

E.4 Near Field API

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

E.4.1 rf_nfm_enable

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

E.4.2 rf_nfm_disable

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

E.4.3 rf_nfm_is_enabled

Function name	**`bool rf_nfm_is_enabled(void)`**
Function description	Checks if Near Field mode is enabled (true) or not (false).
Parameters	None
Return values	`true` Near Field mode enabled `false` Near Field mode disabled
Notes	

E.5 Crypto API

The files for the Crypto API are stored in the folder: sdk\platform\core_modules\crypto.

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

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

E.5.1 aes_init

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

E.5.2 aes_operation

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

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

```
#if (BLE_APP_PRESENT)
      app_init();         // Initialize APP
#endif /* #if (BLE_APP_PRESENT) */
#if (USE_AES)
      aes_test();
#endif
unsigned char key[16]={
  0x06,0xa9,0x21,0x40,0x36,0xb8,0xa1,0x5b,0x51,0x2e,0x03,0xd5,0x34,0x12,0x00,0x06};
unsigned char Plaintext[16]={
  0x53,0x69,0x6e,0x67,0x6c,0x65,0x20,0x62,0x6c,0x6f,0x63,0x6b,0x20,0x6d,0x73,0x67};
unsigned char aes_result[16];

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

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

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

The Native Crypto API includes the following functions:

E.5.3 aes_set_key

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

E.5.4 aes_enc_dec

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

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

E.5.5 AES_set_key

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

E.5.6 AES_convert_key

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

E.5.7 AES_decrypt

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

E.5.8 AES_cbc_decrypt

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

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

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

E.6 Coexistence API

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

E.6.1 wlan_coex_init

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

E.6.2 wlan_coex_enable

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

E.6.3 wlan_coex_reservations

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

E.6.4 wlan_coex_prio_criteria_add

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

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

Return values	None
Notes	

E.6.5 wlan_coex_prio_criteria_del

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

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

1. Add `sdk\platform\core_modules\wlan_coex\wlan_coex.c` in Keil project.
2. Add `sdk\platform\core_modules\wlan_coex` in the Keil project's include path.
3. Add and customise the following defines in `da14580_config.h`:

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

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

E.7 Preferred RF settings

Preferred radio settings for the DA14580/581/583 are stored in the file `sdk\platform\arch\system_settings.h`.

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

E.8 Packet Error Rate (PER)

An application that needs Packet Error Rate metrics must:

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

E.8.1 metrics_packet_rx_func

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

Appendix F Development Environment Known Issues

F.1 Issues When Opening Your Project in Keil for the First Time

F.1.1 Keil IDE Crashes When Clicking on “J-LINK/J-TRACE Cortex” Settings

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

F.1.2 Possible Causes

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

F.1.3 Affected Versions of Keil uVision

At least uVision versions 5.11.1.0 and 5.10.0.2 are affected.

F.1.4 Circumstances of the Error

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

F.1.5 Proposed Solution

Ensure that the .uvoptx file does not exist in the folder of your project. If the file exists and a crash has been identified to happen, delete the .uvoptx file.

Open the Keil project and close it. The .uvoptx file is created automatically in the project folder where the .uvprojx is located.

Open the .uvoptx file, using your favorite text editor.

Under the key <TargetOption> add the flowing lines:

```
<TargetDriverDllRegistry>
<SetRegEntry>
<Number>0</Number>
<Key>JL2CM3</Key>
<Name>-U228202424 -O78 -S0 -A0 -C0 -JU1 -JI127.0.0.1 -JP0 -RST0 -N00("ARM CoreSight
SW-DP") -D00(0BB11477) -L00(0) -TO18 -TC10000000 -TP21 -TDS8007 -TDT0 -TDC1F -
TIEFFFFFFFF -TIP8 -TB1 -TFE0 -FO7 -FD20000000 -FC800 -FN0</Name>
</SetRegEntry>
</TargetDriverDllRegistry>
```

Save the .uvoptx file and close the text editor.

Open the Keil project in uVision.

Click on Project→Options for Project ‘XXX’.

On the ‘Debug’ Tab, select J-Link / J-TRACE Cortex debugger and click on the ‘Settings’ button for the debugger (not the simulator). This is the instance where the crash would happen.

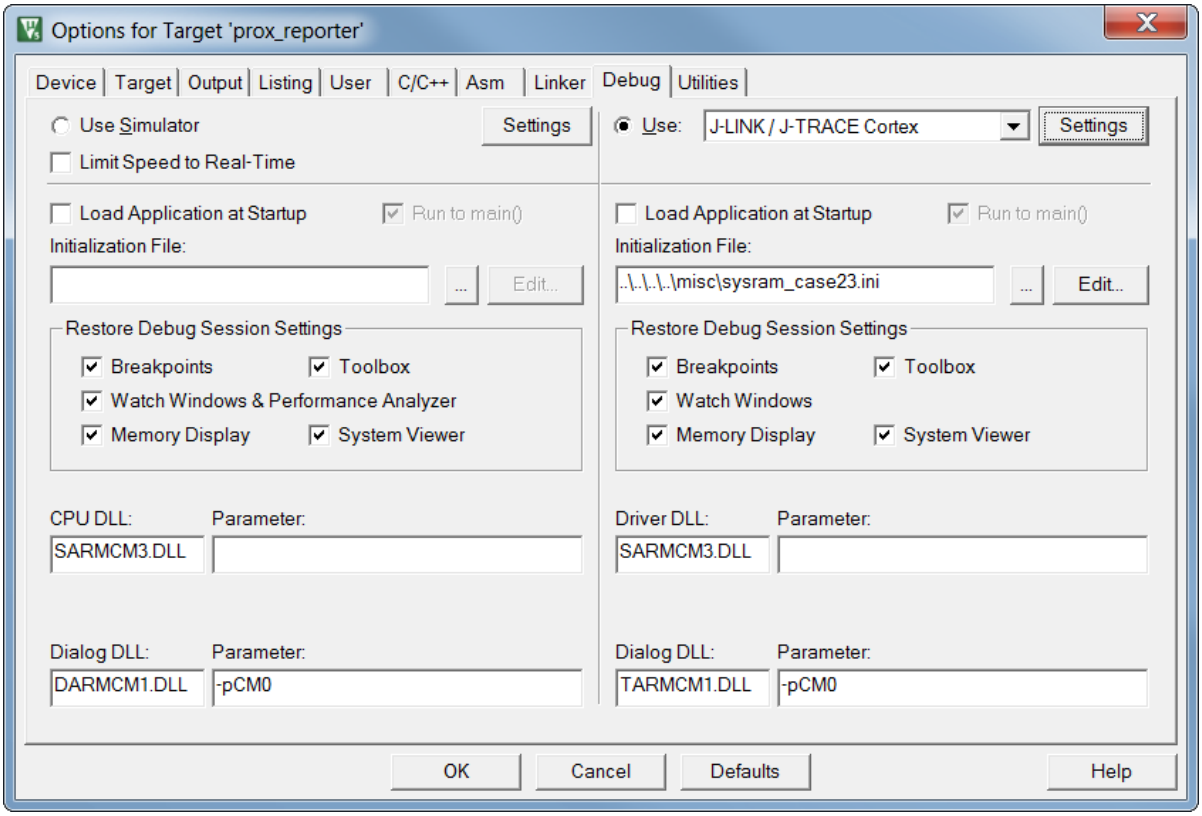
The ‘Cortex JLink/JTrace Target Driver Setup’ Dialog opens. Select your debugger as you would normally do.

Close the dialog windows by clicking OK.

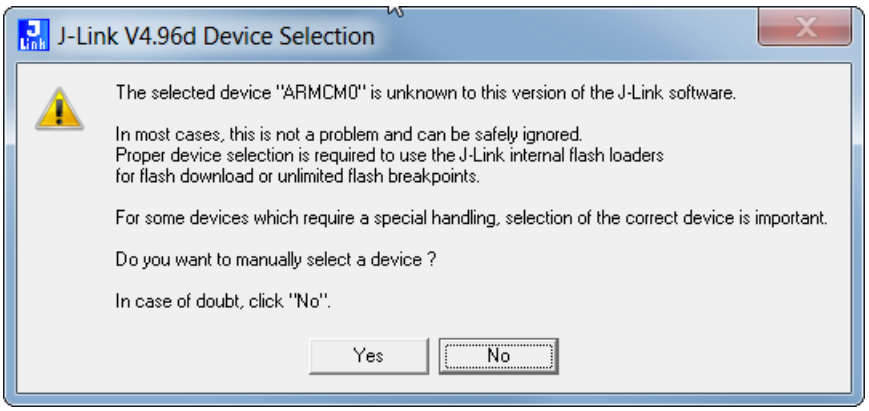
Now, normal operation of j-link debugger is resumed. After you have finished your work, close the Keil uVision IDE to allow for updates to the .uvoptx file to be saved.

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

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



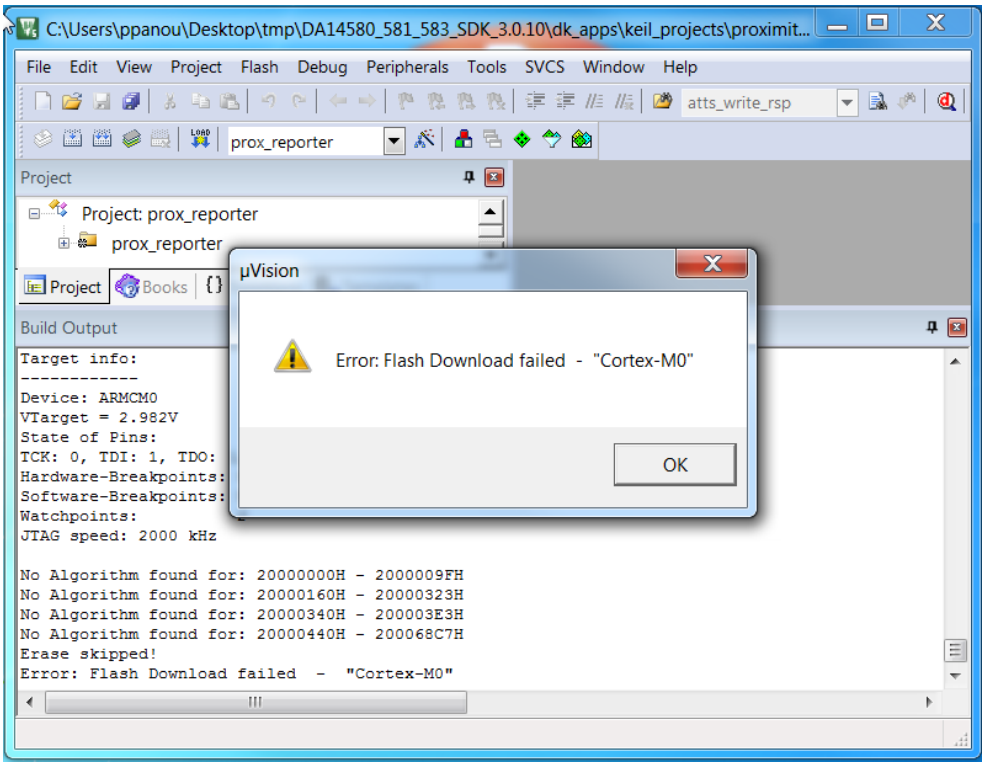
J-Link reports the ARMCM0 device as an unknown device.



The solution is to select “No”.

F.3 Keil 5 IDE Reports Flash Download Failure

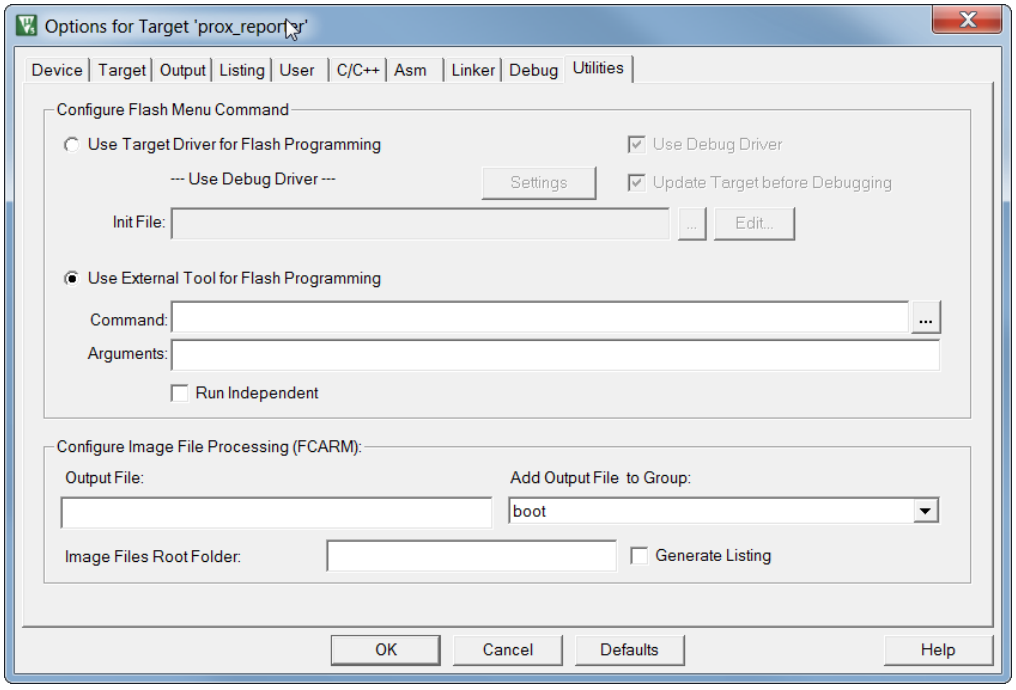
The issue occurs when a new DA14580/581/583 Keil 5 project is created or when the project's `.uvoptx` file is deleted. When the developer attempts to start a new debugging session the following error message appears:



The solution is the following:

Open menu **Project -> Options for Target 'xxx'** and go to the **Utilities** tab page.

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



Appendix G Support for Custom Handling of ATT Read Requests

Custom ATT read request handling behaves as follows:

1. When an ATT read request arrives then its validity is checked first.
   a. If OK, continue to next step.
   b. Otherwise reply with an ATT error.
2. Find the task that manages the service of the handle being read.
3. If the task is registered to receive `ATTS_READ_REQ_IND` messages then:
   a. Prepare and send the `ATTS_READ_REQ_IND` to the task and stop further processing.
   b. The task takes responsibility to reply by calling the `dg_atts_read_cfm()` function.
4. Else:
   a. Read the attribute value from the DB.
   b. Send an ATT read response (also taking the current ATT MTU into account).

A task that requires this mechanism will typically register for `ATTS_READ_REQ_IND` messages at DB creation time using the `dg_register_task_for_read_request()` API. Thereafter it shall receive an `ATTS_READ_REQ_IND` message whenever a peer sends an ATT read request on any of the attribute handles it manages.

Upon reception of the `ATTS_READ_REQ_IND` message a task can modify the ATT DB and then it must reply by calling `dg_atts_read_cfm()`. There are two use cases:

1. The task decides that the read request is valid, (optionally) modifies the value in ATT DB and finally replies by passing `ATT_ERR_NO_ERROR` in the `status_code` argument of `dg_atts_read_cfm()`. This results to sending the ATT read response to the peer device.
2. The task decides that the read request is invalid and responds by passing an ATT error code to `dg_atts_read_cfm()`. This results to sending the ATT error response to the peer device.

Notes:

- This mechanism is supported on both DA14580 and DA14581. The old DA14581-specific `GATTC_READ_CMD_IND` was kept for backwards compatibility. Now it is also available for DA14580.
- The `atts_util.obj` object file must be added in DA14580/583 Keil projects when porting from SDK 5.0.2.1/5.0.3 to SDK 5.0.4.
- API function declarations are located in `arch_patch.h`.
- Code size overhead:
  - `dg_atts_read_cfm()`: 82 bytes.
  - `dg_register_task_for_read_request()`: 26 bytes.
  - `dg_unregister_task_from_read_request()`: 32 bytes.
  - `atts_read_resp_patch()`: 126 bytes.
- Retention memory overhead:
  - `uint64_t dg_registered_tasks`: 8 bytes.
- For examples, please refer to the UDS server role implementation (`udss_task.c`), which uses the custom ATT read request handling mechanism in order to return custom read error codes.

Revision History

Revision	Date	Description
1.0	27-Aug-2015	Initial version. Applies to SDK 5.x for DA14580/581/583.
1.1	01-Aug-2016	Added comment about watchdog in app_on_ble_powered(), app_on_system_powered(). Updated I2C EEPROM API. Updated SDK tree structures. Added Appendix G (Custom Handling of ATT Read Requests).
1.2	20-Dec-2016	Removed systick_usec_units() API details as it is no longer supported
1.3	18-Feb-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.