

User Manual

DA14585 IoT Multi Sensor Development Kit Developer's Guide

UM-B-101

Abstract

The IoT Multi Sensor Development Kit (MSK) based on DA14585 supports 15 Degrees of Freedom and includes five reference applications: IoT Sensors, IoT Smart Tag, and three different types of beacons. The corresponding apps run on iOS/Android devices and the cloud services offers great flexibility to customers in product design. This document provides a detailed guide for developers on using these reference applications for their own projects.

Contents

Abstract	1
Contents	2
Figures.....	6
Tables	7
1 Terms and Definitions.....	9
2 References	10
3 Introduction.....	11
3.1 DA14585 IoT MSK Hardware Features	11
3.2 DA14585 IoT MSK Hardware Architecture	12
4 DA14585 IoT MSK Reference Application.....	12
4.1 Software Features.....	12
4.2 Software Architecture.....	13
4.2.1 Project Files	13
4.2.2 Source Files.....	14
4.2.3 Application Configuration.....	16
4.2.4 Configure for Air Quality Index	17
4.3 Operation Overview	17
4.3.1 General Description	17
4.3.2 Application Initialization	18
4.3.3 Advertise.....	18
4.3.4 Connected/Sensors Idle	19
4.3.5 Connected/Sensors Active	19
4.3.6 Connected/Sensors Stopped.....	20
4.3.7 Disconnect.....	20
4.4 Wkup_adapter.....	20
4.5 Sensor Interface.....	21
4.5.1 General Description	21
4.5.1.1 Timer.....	22
4.5.1.2 INTERRUPT	23
4.5.1.3 FORCED.....	23
4.5.1.4 FORCED_INTER_SNGL_SHOT.....	23
4.5.2 Sensor Interface API.....	24
4.5.3 Driver Adaptation Layer	25
4.6 Device Drivers.....	25
4.6.1 Environmental Sensor	25
4.6.2 Motion Sensor.....	26
4.6.2.1 TDK ICM-42605.....	26
4.6.2.2 BOSCH BMI160.....	26
4.6.3 Magneto Sensor	27
4.6.4 Optical Sensor	27
4.6.5 GPIO Expander	27
4.6.6 Power Amplifier.....	28
4.7 Adding a New Sensor	28
4.8 Sequence Diagrams.....	28

4.8.1	Sensor Fusion Data Reporting	29
4.8.2	Environmental Data Reporting	30
4.9	Dialog Wearable Service V2	31
4.9.1	Feature Report Structure	32
4.9.2	Multi Sensor Report and Sensor Report.....	33
4.9.2.1	Sensor Report for Accelerometer, Gyroscope, and Magnetometer.....	34
4.9.2.2	Sensor Report for Temperature, Humidity, Gas, and Barometric Pressure	34
4.9.2.3	Sensor Report for Indoor Air Quality (IAQ).....	34
4.9.2.4	Sensor Report for Ambient Light and Proximity	35
4.9.2.5	Sensor Report for Button	35
4.9.2.6	Sensor Report for Sensor Fusion	35
4.9.2.7	Sensor Report for Velocity Delta	36
4.9.2.8	Sensor Report for Euler Angle Delta	36
4.9.2.9	Sensor Report for Quaternion Delta	36
4.9.3	Report Structures for Configuration and Control	36
4.9.3.1	Start Command.....	37
4.9.3.2	Stop Command.....	37
4.9.3.3	Read Parameters from Flash Memory	37
4.9.3.4	Reset to Factory Defaults	37
4.9.3.5	Store Basic Configuration in Flash Memory	37
4.9.3.6	Store Calibration Coefficients and Control Configuration in Flash Memory.....	38
4.9.3.7	Return Running Status	38
4.9.3.8	Reset Sensor Fusion and Calibration Configuration	38
4.9.3.9	Basic Configuration	38
4.9.3.10	Read Basic Configuration.....	40
4.9.3.11	Set Sensor Fusion Coefficients Command	41
4.9.3.12	Read Sensor Fusion Coefficients	42
4.9.3.13	Set Calibration Coefficients	42
4.9.3.14	Read Calibration Coefficients	42
4.9.3.15	Set Calibration Control Flags.....	43
4.9.3.16	Read Calibration Control	44
4.9.3.17	Fast Accelerometer Calibration	44
4.9.3.18	Set Calibration Modes	44
4.9.3.19	Read Calibration Modes	45
4.9.3.20	Read Device Sensors.....	45
4.9.3.21	Read Software Version.....	46
4.9.3.22	Start LED Blink	46
4.9.3.23	Stop LED Blink.....	46
4.9.3.24	Set Proximity Hysteresis Limits	47
4.9.3.25	Read Proximity Hysteresis Limits	47
4.9.3.26	Calibration Complete	47
4.9.3.27	Proximity Calibration Command	47
4.10	Sensor Calibration Library	48
4.10.1	Overview	48

4.10.1.1	Modes of Operation	48
4.10.1.2	Calibration Routines	48
4.10.1.3	Calibration Procedure	49
4.10.2	API Usage	50
4.10.2.1	Allocation	50
4.10.2.2	Initialization	50
4.10.2.3	Processing	52
4.11	Sensor Fusion Library	53
4.11.1	Overview	53
4.11.2	SmartFusion Integration Engine	53
4.11.2.1	Modes of Operation	53
4.11.2.2	API Usage	54
4.11.3	SmartFusion Attitude and Heading Reference System	55
4.11.3.1	Modes of Operation	55
4.11.3.2	API Usage	56
5	Smart Tag Reference Application	57
5.1	Introduction	57
5.2	Software Features	58
5.2.1	Profiles and Services	58
5.2.2	Alerts	58
5.2.3	Advertising and Sleep Phases	58
5.2.4	Push-Button Interface	59
5.2.5	Security	59
5.2.6	Battery Level	59
5.3	Software Architecture	60
5.4	Operation Overview and State Machines	60
5.4.1	Application Configuration Parameters	60
5.4.2	Application Task State Machine	60
5.4.3	Callback Functions	62
5.4.4	Advertising	62
5.4.5	Connection	63
5.4.6	Security	63
5.4.7	Push button	63
5.4.8	Proximity Reporter and Alerts	64
5.4.9	PWM Engine	65
5.4.10	SmartTag Sequence Diagram	66
6	Beacon Reference Applications	66
6.1	Introduction	66
6.2	What is a Beacon?	67
6.3	Beacon Example	67
6.4	Beacon Formats	68
6.4.1	iBeacon	68
6.4.2	AltBeacon	69
6.4.3	Eddystone	70
6.4.3.1	Eddystone-UID	71
6.4.3.2	Eddystone-URL	72

6.4.3.3	Unencrypted Eddystone-TLM.....	73
6.5	Software Features.....	74
6.6	Beacon Parameters.....	74
6.6.1	Advertising Data.....	74
6.6.1.1	Using the <code>user_default_beacon_config Struct</code>	75
6.6.1.2	Reading Advertising Data from Flash.....	76
6.6.2	Advertising Interval.....	77
6.7	Software Architecture.....	77
6.8	Operation Overview.....	78
6.8.1	Configuration Switches.....	79
6.9	User Advertise SW Module.....	80
6.9.1	Style.....	80
6.9.2	Pattern.....	80
6.9.3	User Advertise SW Module Callbacks.....	82
6.10	Device Configuration Service.....	82
6.10.1	Device Configuration Service Specification.....	82
6.11	Environmental Data Notifications Service.....	83
6.11.1	Environmental Data Notifications Service Specification.....	83
6.12	Beacon Configuration.....	84
6.12.1	Beacon Configuration Memory Map.....	84
6.13	Battery Level Sampling.....	86
6.14	Beacon Examples for DA14585 IoT MSK.....	86
6.14.1	AltBeacon.....	86
6.14.1.1	AltBeacon Example Sequence Diagram.....	87
6.14.2	Eddystone.....	88
6.14.2.1	Eddystone Example Sequence Diagram.....	89
6.14.3	iBeacon.....	89
6.14.3.1	iBeacon Example Sequence Diagram.....	91
7	Memory Footprint and Power Measurements.....	91
7.1	Memory Footprint.....	91
7.2	Power consumption.....	92
	Appendix A MSK Boot Sequence.....	93
	Appendix B Memory Map.....	93
	Appendix C Using the <code>mkimage</code> Application.....	95
C.1	<code>mkimage</code> Scripts.....	95
C.2	<code>mkimage</code> Modes.....	96
C.2.1	<code>mkimage single</code>	96
C.2.2	<code>mkimage multi</code>	96
C.2.3	<code>mkimage whole_img</code>	97
C.2.4	<code>mkimage multi_no_suota</code>	97
C.2.5	<code>mkimage cfg</code>	97
	Appendix D Flash Programming in MSK Applications.....	97
D.1	Basic Information About the MSK Applications.....	97
D.1.1	Product Header.....	97
D.1.2	Image Header.....	98

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

D.1.3	Beacon Configuration Struct and Configuration Struct Header.....	98
D.1.4	Smart Tag Bonding Data, IoT Flash Base, IoT Flash Base Cal.....	99
D.2	Flash Programming.....	99
D.2.1	Burning the Whole Image in Flash Memory	99
D.2.2	Preparing the Various .img and .bin Files Manually.....	101
Appendix E Using the SUOTA Application for Android		103
Revision History		109

Figures

Figure 1: Internet of Things (IoT).....	11
Figure 2: Hardware Architecture of DA14585 IoT MSK	12
Figure 3: DA14585 IoT MSK Software Architecture.....	13
Figure 4: General Application Flow of DA14585 IoT MSK.....	18
Figure 5: DA14585 IoT MSK in Advertise State	18
Figure 6: Overview of Sensors Data Path	20
Figure 7: Sensor Interface Overview.....	22
Figure 8: Timer Sensors Timeline	22
Figure 9: INTERRUPT Sensor Access Sequence	23
Figure 10: Forced Read Sequence	23
Figure 11: FORCED_INTER_SINGL_SHOT Functionality	23
Figure 12: SI Registration Path	24
Figure 13: Sequence Diagram of Sensor Fusion Reporting	30
Figure 14: Sequence Diagram of Environmental Sensor Reporting	31
Figure 15: Smart Tag Application Task FSM	61
Figure 16: Smart Tag Reference Application Sequence Diagram	66
Figure 17: Beacon Protocol Logos	67
Figure 18: Bluetooth Low Energy Beacon.....	67
Figure 19: Description of the Exhibit on a Smartphone.....	68
Figure 20: iBeacon Frame.....	68
Figure 21: AltBeacon Frame.....	69
Figure 22: Eddystone Modes Supported by Dialog's Beacon Reference Design	70
Figure 23: Eddystone Different Mode Frames Analyzed	70
Figure 24: Example of a Device Configuration Struct in Flash Memory	76
Figure 25: Beacon SW System Overview	77
Figure 26: Operation Overview	79
Figure 27: User Advertise Usage Example	81
Figure 28: Data Format in Write Configuration.....	82
Figure 29: Indication Data Format in Read Response	83
Figure 30: AltBeacon Example Transition Diagram	87
Figure 31: Altbeacon Sequence Diagram	87
Figure 32: Eddystone UID/URL/TLM Example Transition Diagram	88
Figure 33: Eddystone Sequence Diagram	89
Figure 34: iBeacon Example Transition Diagram.....	90
Figure 35: iBeacon Sequence Diagram	91
Figure 36: Application Programmed in OTP Flags.....	93
Figure 37: Analyzing a Flash Memory Image.....	94
Figure 38: Initial Window to Choose Device and Connection Type	99
Figure 39: Opening SmartSnippets Board Setup	100
Figure 40: Smart Snippets Board Setup Window.....	100
Figure 41: SPI Flash Programmer.....	100
Figure 42: Device Config Struct Format in .txt File	102
Figure 43: Programming the Various Fields of the Device Configuration Struct.....	102
Figure 44: Creating a Custom Dev_Conf_Struct .bin File.....	103
Figure 45: SUOTA App Icon.....	104

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

Figure 46: Device Selection Menu	104
Figure 47: DIS Screen	105
Figure 48: File Selection Screen	105
Figure 49: SUOTA Parameter Settings	106
Figure 50: SUOTA Uploading Screen	107
Figure 51: Successful Update Screen	108

Tables

Table 1: Source Files for DA14585 IoT MSK: Overview	14
Table 2: Source Files Specific to DA14585 IoT MSK	15
Table 3: Header Files for the Configuration of DA14585 IoT MSK	15
Table 4: Configuration Parameters	16
Table 5: DWSv2 Characteristics	31
Table 6: Features Report Structure	32
Table 7: Multi Sensor Report	33
Table 8: Sensor Report	33
Table 9: Report Types/Report ID's	33
Table 10: Report Structure for Accelerometer, Gyroscope, and Magnetometer	34
Table 11: Bitfield Structure for <code>snsr_state</code>	34
Table 12: Environmental Sensor Report	34
Table 13: Indoor Air Quality (IAQ) Report	34
Table 14: Sensor Report for Ambient Light and Proximity	35
Table 15: Sensor Report for Button	35
Table 16: Sensor Report for Sensor Fusion	35
Table 17: Sensor Report for Velocity Delta	36
Table 18: Sensor Report for Euler Angle Delta	36
Table 19: Sensor Report for Quaternion Delta	36
Table 20: Report Structure for Commands	37
Table 21: Start Command	37
Table 22: Start Command Reply	37
Table 23: Stop Command	37
Table 24: Stop Command Reply	37
Table 25: Read Flash Command	37
Table 26: Reset to Defaults (RtD) Command	37
Table 27: Store Basic Configuration Command	37
Table 28: Store Calibration and Control Command	38
Table 29: Return Running Status Command	38
Table 30: Return Running Status Reply	38
Table 31: Reset Sensor Fusion and Calibration Configuration command	38
Table 32: Basic Configuration Command	38
Table 33: Read Basic Configuration Command	40
Table 34: Read Basic Configuration Command Reply	40
Table 35: Set Sensor Fusion Coefficients Command	41
Table 36: Read Sensor Fusion Coefficients Command	42
Table 37: Read Sensor Fusion Coefficients Command Reply	42
Table 38: Set Calibration Coefficients Command	42
Table 39: Read Calibration Coefficients Command	42
Table 40: Read Calibration Coefficients Command Reply	42
Table 41: Set Calibration Control Flags Command	43
Table 42: Calibration Control Flags #1	43
Table 43: Calibration Control Flags #2	43
Table 44: Calibration Parameters	43
Table 45: Read Calibration Control Flags Command	44
Table 46: Read Calibration Control Flags Command Reply	44
Table 47: Fast Accelerometer Calibration Command	44
Table 48: Fast Accelerometer Calibration Reply	44
Table 49: Set Calibration Modes Command	44

Table 50: Read Calibration Modes Command	45
Table 51: Read Calibration Modes Command Reply	45
Table 52: Read Device Sensors Command	45
Table 53: Read Device Sensors Command Reply	45
Table 54: Read Application Software Version Command	46
Table 55: Read Application Software Version Command Reply	46
Table 56: Start LED Blink Command	46
Table 57: Stop LED Blink Command	46
Table 58: Set Proximity Hysteresis Limits Command	47
Table 59: Read Proximity Hysteresis Limits Command	47
Table 60: Read Proximity Hysteresis Limits Command Reply	47
Table 61: Calibration Complete Notification	47
Table 62: Proximity Calibration Command	47
Table 63: Proximity Calibration Command Reply	47
Table 64: Alert Types	58
Table 65: Push-Button Interface	59
Table 66: Smart Tag Application Configurable Parameters	60
Table 67: Application Task: FSM States	60
Table 68: State Transitions of the Application Task FSM	61
Table 69: Example of Advertising Data from a Museum Beacon	68
Table 70: AltBeacon Protocol Fields	69
Table 71: Eddystone Frame Types	71
Table 72: Eddystone UID Frame	71
Table 73: Frame Specification	72
Table 74: URL Scheme Prefix	72
Table 75: Eddystone-URL HTTP URL Encoding	72
Table 76: Eddystone-TLM Frame Specification	73
Table 77: Format of Struct <code>user_beacon_config_tag</code>	74
Table 78: Advertising Interval Location	77
Table 79: Source Files of Beacon Reference Applications	77
Table 80: List of Software Configuration Switches	79
Table 81: List of Profile Configuration Switches	79
Table 82: Characteristics of the Device Configuration Service	82
Table 83: Characteristics of the Environmental Data Notifications Service	83
Table 84: Configuration Data Format	84
Table 85: Memory Footprint	91
Table 86: Power Consumption	92
Table 87: Parts of the Image Depending on the Application	94
Table 88 Available <code>mkimage</code> Scripts	95
Table 89: Product Header Format	97
Table 90: Image Header Format	98
Table 91: Beacon Configuration Header	98
Table 92: Beacon Configuration Struct Format	99
Table 93: Files Needed or Created During Flash Programming	101

1 Terms and Definitions

ADC	Analog to Digital Converter
AHRS	Attitude and Heading Reference System
API	Application Programming Interface
BASS	Battery Service Server
BD Address	Bluetooth Device Address
BLE	Bluetooth Low Energy
CIB	Communication Interface Board
DAL	Driver Adaptation Layer
DISS	Device Information Service Server
DWS	Dialog Wearable Service
FEM	Front End Module
FIFO	First In First Out buffer
FSM	Finite State Machine
GAP	Generic Access Profile
GAPC	Generic Access Profile Controller
GAPM	Generic Access Profile Manager
GATT	Generic Attribute Profile
IAQ	Indoor Air Quality
IE	Integration Engine
IMU	Inertial Measurement Unit
IoT	Internet of Things
IRQ	Interrupt Request
LSB	Least Significant Bit
MEMS	Microelectromechanical Systems
MSB	Most Significant Bit
MSK	Multi Sensor Development Kit
MTU	Maximum Transmission Unit
NTF	Notifications are Allowed
NVDS	Non-Volatile Data Storage
ODR	Output Data Rate
PWM	Pulse Width Modulation
RAM	Random Access Memory
Report	Notification of Sensor Data and Control
RSSI	Received Signal Strength Indicator
SCL	Sensor Calibration Library
SF	Sensor Fusion
SFL	Sensor Fusion Library
SI	Sensors Interface
SUOTA	Software Update over the Air
UART	Universal Asynchronous Receiver/Transmitter
UUID	Universal Unique Identifier
WR	Write Enabled

2 References

- [1] UM-B-080, DA14585/586 SDK 6 Software Developer's Guide, User Manual, Dialog Semiconductor.
- [2] UM-B-079, DA14585/586 SDK 6 Software Platform Reference, User Manual, Dialog Semiconductor.
- [3] RW-BLE-GAP-IS, GAP Interface Specification, Riviera Waves.
- [4] UM-B-048, DA14585/DA14586 Getting Started Guide with the Basic Development Kit V2.0, User Manual, Dialog Semiconductor.
- [5] DA14585, Datasheet v3.2, Dialog Semiconductor.
- [6] UM-B-089, DA14585 Range Extender Reference Application, User Manual, Dialog Semiconductor.
- [7] <https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers>
- [8] UM-B-065, Bluetooth® Smart Communication Interface Board, User Manual, Dialog Semiconductor.
- [9] <https://github.com/google/eddystone>
- [10] <https://github.com/google/eddystone/tree/master/eddystone-uid>
- [11] <https://github.com/google/eddystone/tree/master/eddystone-url>
- [12] <https://github.com/google/eddystone/blob/master/eddystone-tlm/tlm-plain.md>
- [13] <https://github.com/AltBeacon/spec>
- [14] UM-B-095, DA14585 IoT Multi Sensor Development Kit Hardware Design, User Manual, Dialog Semiconductor.
- [15] <https://www.bluetooth.com/specifications/gatt>.
- [16] AN-B-001, DA14580/581/583 Booting from Serial Interfaces, Application Note, Dialog Semiconductor
- [17] UM-B-102, DA14585 Getting Started Guide with the IoT Multi Sensor Development Kit, User Manual, Dialog Semiconductor

3 Introduction

This document allows users to develop their own applications based on the DA14585 IoT Multi Sensor Development Kit (MSK) reference design. The following chapters present detailed features of each project and the software configuration of the reference design.

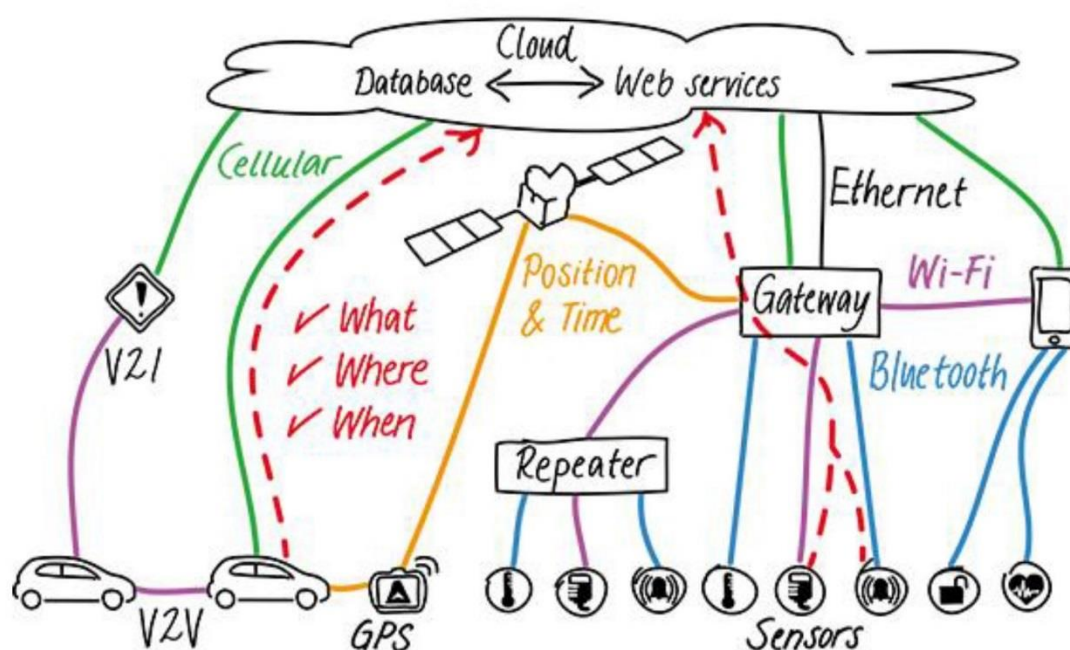


Figure 1: Internet of Things (IoT)

3.1 DA14585 IoT MSK Hardware Features

- Highly integrated Dialog Semiconductor DA14585 Bluetooth® Smart SoC
- Stand-alone module
- Low cost due to printed antenna
- Low cost PCB
- Combined accelerometer/gyroscope sensor unit
- Combined sensors:
 - Accelerometer and gyroscope sensor unit
 - Indoor Air Quality, Temperature, Humidity and Pressure
 - Ambient Light Sensor and Proximity
- Access to processor via JTAG and UART from the enclosure
- Programmable RF power up to +9.3dBm
- Three Led indicators
- General purpose push button
- Expansion slots
- Powered by two low cost AAA alkaline batteries

3.2 DA14585 IoT MSK Hardware Architecture

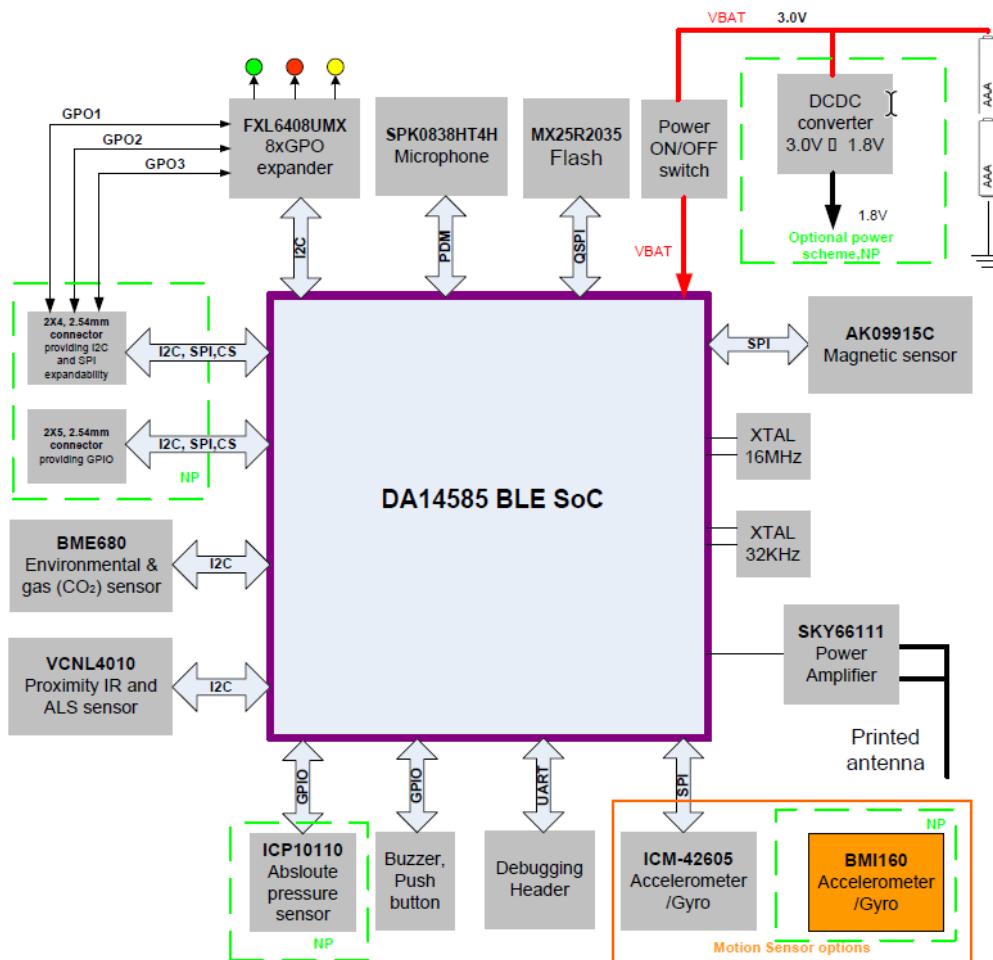


Figure 2: Hardware Architecture of DA14585 IoT MSK

4 DA14585 IoT MSK Reference Application

4.1 Software Features

This section explains the advanced software features of Dialog's DA14585 IoT MSK reference applications:

- GAP peripheral role.
- GATT-based bidirectional data and control transfers.
- Sensor fusion library (SFL) with updated rates from 10 Hz to 100 Hz and selection among three different sensors: accelerometer, gyroscope, and magnetometer.
- Three environmental sensors: pressure, humidity, and temperature.
- Ambient light and proximity sensor.
- Indoor Air Quality (IAQ) feature.
- Switch between Sensor Fusion (SF) data mode and raw data mode.
- Notifications for sensor data streaming.
- Sensor range and updated rate control.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- Save and restore operation settings.
- Three calibration modes for magnetometer.
- Gyroscope drifting elimination algorithm.
- iOS and Android central Apps.
- Auto sleep and motion wakeup.

4.2 Software Architecture

4.2.1 Project Files

The project resides in "projects\target_apps\iot\iot_585\Keil_5\iot585.uvprojx". The same project supports both raw and SF operation. The mode of operation can be switched from the central device application.

Project specific folders are:

- Sensor Fusion library files: iot\common_iot_files\lib
- Sensor Calibration files: target_apps\iot\common_iot_files\src\calibration
- IoT Profile files: target_apps\iot\common_iot_files\src\profiles

The project shares the following common folders:

- Driver Adaptation Layer files: projects\target_apps\common\src\Driver_Adaptation_Layer
- Drivers folders: projects\target_apps\common\src\drivers
- Sensor Interface folder: projects\target_apps\common\src\Sensors_Interface
- Common SDK folder: SDK_585\sdk\

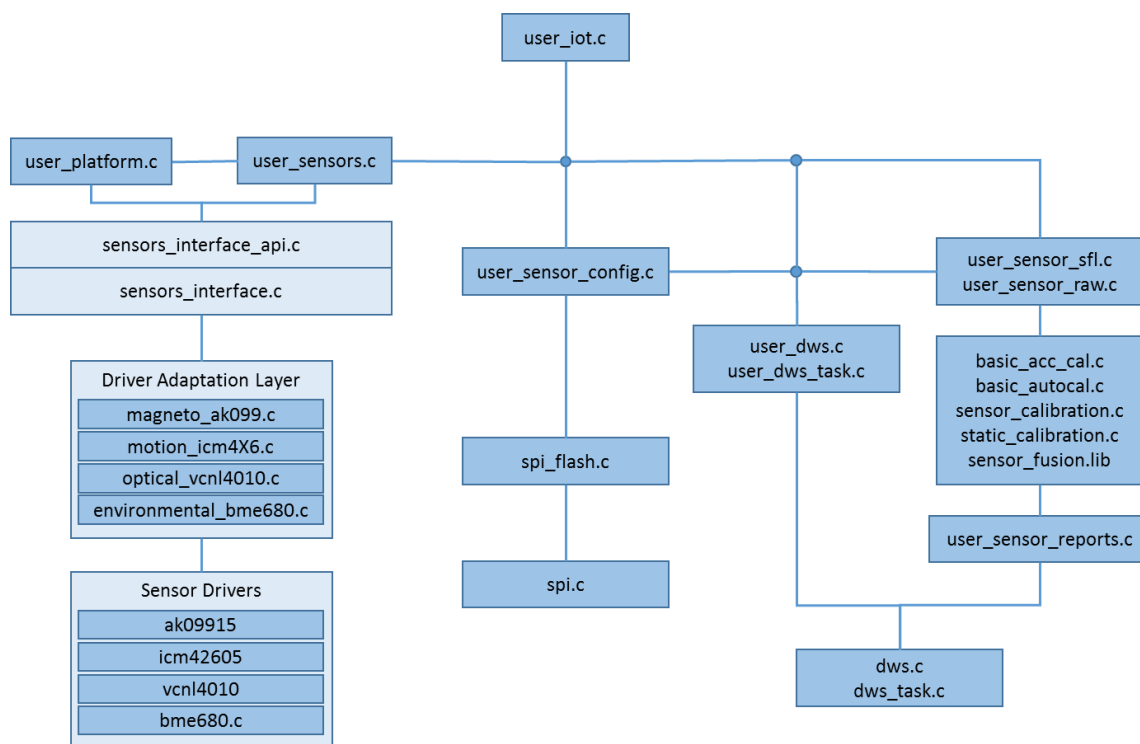


Figure 3: DA14585 IoT MSK Software Architecture

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

4.2.2 Source Files

The reference applications of DA14585 IoT MSK are developed based on the DA14585 SDK software. For more information on the SDK, please refer to [1] and [2].

The source code files of the reference applications are briefly described in Table 1.

Table 1: Source Files for DA14585 IoT MSK: Overview

Group	Files	Description
User Platform Files (IoT DK)	user_periph_setup.c i2c_gpio_extender.c user_iot_dk_utils.c sensors_periph_interface.c	Peripheral setup, Range extender, LED control functions, Sensors i2c-spi communication helper functions.
User Application (IoT DK)	user_dws.c user_dws_task.c user_iot.c	Main application files. File user_iot.c contains the connect/disconnect/advertise handlers. Files user_dws.c and user_iot_task.c contain the user interface to the DWS.
User Profiles (IoT DK)	dws.c dws_task.c	Service functions and FSM handlers.
Sensor Calibration (IoT DK)	basic_acc_cal.c basic_autocal.c static_calibration.c sensor_calibration.c smartfusion_autocal.lib	Accelerometer calibration functions. Magnetometer calibration functions, specific to each calibration mode.
User Sensors (IoT DK)	user_sensor_config.c user_sensors.c user_sensor_reports.c user_sensor_raw.c user_sensor_sfl.c	Sensor initialization. Sensor state machine. Main application callback functions.
User SFL	user_sfl_util.c sensor_fusion.lib	Files to process sensor data
Sensor interface (IoT DK)	sensors_interface_api.c sensors_interface.c	New interface to simplify sensor integration to project.
Driver Adaptation Layer (IoT DK)	magneto_ak099.c motion_icm4x6.c (or motion_bmi160.c) optical_vcnl4010.c environmental_bme680.c	This layer defines a common instruction set for the "Sensor interface" to access the sensors.
bsec_lib (IoT DK)	user_iaq.c libalgobsec_full.lib	Functions that estimate the air quality output.
wkup_adapter (IoT DK)	wkup_adapter.c	A wrapper to the "wkupct_quadec" to further extend the possibilities of the module.
Drivers (IoT DK)	bme680 folder icm426xx (or bmi160) folder ak09915 folder vcnl4010 folder	These folders contain the factory drivers of the sensors.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

Table 2: Source Files Specific to DA14585 IoT MSK

File Name	Description
user_iot.c	Contains all top level BLE callback functions for connection, disconnection, and advertising.
user_dws.c user_dws_task.c	Provide users space access to the DWS custom service.
user_sensors.c	Contains sensor initialization functions. Handles sensor data on application level.
user_sensor_reports.c	The user space functions that are specific for sending reports (sensor and sensor fusion data) to the central device.
user_sensor_sfl.c user_space_raw.c	User space sensor data processing.
basic_acc_cal.c	Accelerometer calibration functions.
static_calibration.c basic_autocal.c sensor_calibration.c smartfusion_autocal.lib	The calibration functions for the three modes, that is, static, basic auto, and SmartFusion auto.
user_sfl_util.c sensor_fusion.lib	Sensor fusion library API.
sensors_interface_api.c sensors_interface.c	Expose the Sensor Interface API to the application user. Core methods to implement the Sensor Interface module functionality.
icm426xx_impl.c (or bmi160_impl.c) bme680_imp.c ak09915_impl.c vcnl4010_impl.c	Driver functions for each sensor that act as middleware for the "driver adaptation layer". These drivers make use of the actual manufactured drivers and use only the functions that are really needed for this implementation instead of all available features.

The project contains a header file for each source file that contains function prototypes and definitions. Some header files contain information that is essential for the operation and configuration of the IoT project. The header files reside in the folder `projects\target_apps\iot\iot_585\src\config`. The header files are listed in [Table 3](#).

Table 3: Header Files for the Configuration of DA14585 IoT MSK

File Name	Description
da1458x_config_basic.h (SDK)	Basic configuration of DA14585 for security, watchdog, and print functions. Furthermore, in this file users can control the sensors that are included in the project.
da1458x_config_advanced.h (SDK)	Advanced features of DA14585, such as low power clock source.
user_config.h (SDK)	Configure data related to sleep modes, advertise, and parameter update.
user_app_iot_config.h (IoT DK)	Application specific definitions: sensor calibration parameters, motion wakeup, advertising time, and LED signaling.
user_callback_config.h (SDK)	Application callback functions.
user_modules_config.h (IoT DK)	User-related module activation/deactivation.
user_periph_setup.h (IoT DK)	Peripheral related definitions (GPIO).
user_profiles_config.h (IoT DK)	Included profiles.
user_dws_config.h (IoT DK)	Definition of Dialog Wearable Service.

4.2.3 Application Configuration

A group of compilation switches allows to control the application's behavior. The most important switches are listed in [Table 4](#).

Table 4: Configuration Parameters

Name	Default value	Description
user_config.h		
app_default_sleep_mode	ARCH_EXT_SLEEP_ON	This controls whether the processor to enter sleep mode or not. Use ARCH_SLEEP_OFF when connected to a debugger.
USER_DEVICE_NAME	"IoT-585"	The advertising name string.
connection_param_configuration	<ul style="list-style-type: none"> Min-Max connection interval: 20 ms to 40 ms, Latency: 4 (events missed), Supervision timeout: 2 s. 	Recommended settings.
da1458x_config_basic.h		
CFG_PRINTF	DISABLED	Enables/disables the use of UART debug messages.
CFG_WDOG	ENABLED	Enables the Watchdog timer.
VCNL4010_OPTO_SENSOR_AVAILABLE	ENABLED	Enables/disables the opto sensor.
AK099XX_MAGNETO_SENSOR_AVAILABLE	ENABLED	Enables/disables the magnetometer sensor.
ICM4XX_ACCEL_SENSOR_AVAILABLE	ENABLED	Enables/disables the ICM4xx accelerometer-gyroscope sensor.
BMI160_ACCEL_SENSOR_AVAILABLE	DISABLED	Enables/disables the BMI160 accelerometer-gyroscope sensor.
BME680_ENVIRONM_SENSOR_AVAILABLE	ENABLED	Enables/disables the environmental sensor.
IAQ_ENABLED	ENABLED	Enables/disables the IAQ algorithm.
IAQ_RESTORE_STATUS_ENABLE	ENABLED	Enables/disables saving the IAQ algorithm state
CFG_RANGE_EXT	BYPASS	Configures the range extender's power value. For more information please refer to [6] .
da1458x_config_advanced.h		
CFG_LP_CLK	LP_CLK_XTAL32	<p>LP_CLK_XTAL32: Use external XTAL32 as a low power clock (recommended).</p> <p>LP_CLK_RCX20: Use internal RCX20 as a low power clock.</p>
CFG_NVDS_TAG_BD_ADDRESS	{0x99, 0x00, 0x00, 0xCA, 0xEA, 0x80}	Defines the BD address if it is not programmed in OTP.

Name	Default value	Description
user_app_iot_config.h		
FAST_ADV_INTERVAL	160 (= 100 ms)	The advertising interval. Unit: 0.625 ms.
ADV_TIME_OUT	6000 (= 60 s)	The time that the device advertises before it goes to sleep. Unit: 10 ms.
ALWAYS_ADVERTISE	DISABLED	If enabled, the system will remain in advertising state and never go to sleep mode.
USE_FAST_ACC_CAL	ENABLED	If enabled, the accelerometer calibration becomes available.
GYRO_SDC_ENABLED	ENABLED	If enabled, the gyroscope drift compensation is applied.
USE_SPI_FLASH_CONFIG	ENABLED	If enabled, external flash will be active to read and store configuration parameters.
MAGNETO_CAL_ENABLED	ENABLED	Enables the calibration function of magnetometer. For more information see section 4.11.1.

4.2.4 Configure for Air Quality Index

The Bosch BSEC Library that computes the Air Quality Index (AQI) is not included in the default configuration (`IAQ_ENABLED` is undefined). If users would like to compile an image that includes this library, other features should be removed to gain the required memory space. For example, this could be accomplished with the following steps:

1. Remove the VCNL4010 proximity sensor and undefine `VCNL4010_OPTO_SENSOR_AVAILABLE` from `dal458x_config_basic.h`.
2. Disable "wake on motion" feature by defining `ALWAYS_ADVERTISE` in `user_app_iot_config.h`. This also removes the low power configuration parts of ICM426xx driver.
3. Disable the fast accelerometer calibration and undefine `ALWAYS_USE_FAST_ACC_CAL` in `user_app_iot_config.h`.
4. Build.
5. Define `IAQ_ENABLED` from `dal458x_config_basic.h`
6. Build again. The produced `iot585.hex` now includes the AQI feature.

4.3 Operation Overview

4.3.1 General Description

A general view of the application flow is presented in [Figure 4](#).

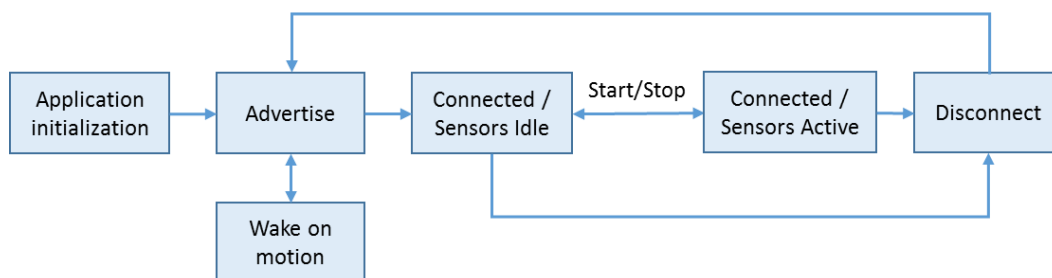


Figure 4: General Application Flow of DA14585 IoT MSK

4.3.2 Application Initialization

The function `user_iot_app_on_init()` located in "user_iot.c" file handles:

- Initialize processor peripherals, `periph_init()`.
- Initialize radio extender GPIOs, `init_ext_gpio()`.
- Disable all external sensors and turn off all LEDs to reduce power consumption, `clr_led(ALL_LED)`.
- Explicitly for the accelerometer sensor inside this routine, certain registers will be set to avoid overconsumption.
- Clear sensor interface variables, `wkup_ad_init()`.
- Load sensor configuration from flash if available, `user_periph_sensors_initialize()`.
- Start BLE parameter updating.

4.3.3 Advertise

In this state, DA14585 IoT MSK will be visible to client applications.

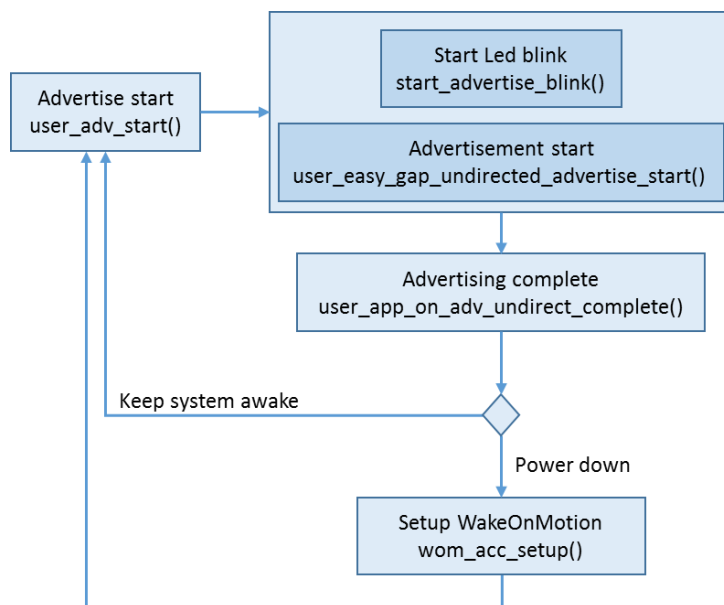


Figure 5: DA14585 IoT MSK in Advertise State

This block has the following operations:

- **"Advertise start"** block calls the `user_adv_start()` function. The following actions take place:

- Start a timer to produce a LED blink using `start_advertise_blink()`. While the system remains in advertising state, the LED will blink periodically according to the `ADVERTISE_LED_ON_TIME` and `ADVERTISE_LED_OFF_TIME`.
- Actual advertisement begins by sending a `gapm_start_advertise_cmd` to the GAP layer using the `user_easy_gap_undirected_advertise_start()` function.
- **"Advertising complete"**: Advertisement time period is determined by the value set on `ADV_TIME_OUT`.
 - When the timer expires, `user_app_on_adv_undirect_complete()` is executed to terminate advertising.
 - Routine `user_adv_start()` will be executed again if sleep mode is set to `ARCH_SLEEP_OFF`.
 - If sleep mode is set to `ARCH_EXT_SLEEP_ON`, the system will attempt to power off and enable wake-on-motion interrupt. This action will not apply if:
 - The user has explicitly set the system to stay "awake" by setting the `ALWAYS_ADVERTISE` switch.
 - The motion sensor is excluded from this project and therefore there are no means to wake the processor.
- **"Setup WakeOnMotion"**. The accelerometer for this purpose is configured for low power operation with the "anymotion" interrupt function set. The processor can go to Extended Sleep mode and wakes up only when it receives an interrupt from the accelerometer. Then the interrupt handler `wkup_intr_non_connected_cb()` is executed and advertising is initiated.

4.3.4 Connected/Sensors Idle

On connected state, the `user_on_connection()` function will execute all the necessary procedures in order to set up the system for run-time operation:

- Stops the advertising timer, `user_stop_adv_timer()`.
- Starts the blink timer to indicate connection event, `start_connection_blink()`.
- Sets the sensors to the default parameters, `user_periph_sensors_initialize()`.
- Suspend the sensors, `user_periph_sensors_suspend()`.
- Starts the BLE parameter update procedure, `app_easy_gap_param_update_start()`.

4.3.5 Connected/Sensors Active

The sensors will enter active state when a "Start" command is sent from the application. In this state the following actions will take place:

- Initialize sensors control variable.
- Initialize gyro static drift compensation, if enabled.
- Initialize magneto calibration, if enabled.
- Initialize sensor fusion algorithm.
- Initialize all external sensors, if enabled.

Now the system will begin retrieving data from all active sensors and forward these to the client application. During this process, if `one_shot_cal_active` is enabled, magnetometer calibration will take place. This is a separate operation and will be discussed in section 4.9.3.27.

A general description for the data path in the "run" state is presented in Figure 6. Note that "Sensor Interface" has a significant role in this operation and will be described in detail in section 4.5.

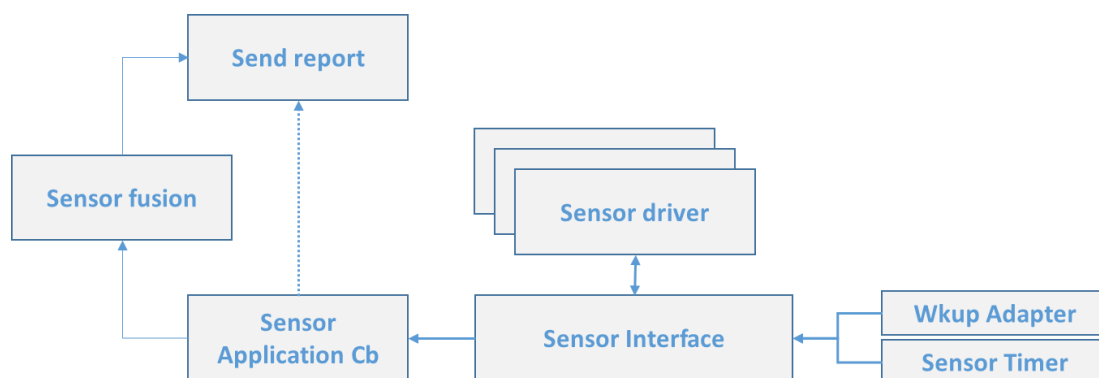


Figure 6: Overview of Sensors Data Path

"Data ready" indication for each sensor is triggered either from timer events or from interrupt events. In both cases this indication is forwarded to the "**Sensor Interface**" to handle. This entity is responsible for discovering which sensor has created the event. This information will be used to select the correct sensor driver to retrieve data. The "**Sensor Interface**" is also aware of which application's callback function is bound to this sensor and will forward the data. The callback function is defined by users and from that point on users will decide how to handle this information. In this project, the information is sent to client application using the reporting mechanism.

This path differentiates if the SFL is used. Data from these devices (accelerometer, gyroscope, and magnetometer) will be processed in the SFL and the outcome will be fed to the reporting mechanism.

4.3.6 Connected/Sensors Stopped

A "Stop" command terminates sensor operations. If calibration takes place during the "run" state, stopping the sensors will initiate the following:

- Save magnetometer or accelerometer calibration results in flash memory
- Inform the client application that calibration has finished

Note that sensor calibration will be described extensively in section 4.10.

4.3.7 Disconnect

On this state, function `user_on_disconnect` will execute the following:

- Clear all LEDs and suspend sensors to minimize power consumption
- Reinitiate the advertising procedure

4.4 Wkup_adapter

This module is an extension to the "WakeUP Capture Timer" to further extend the possibilities of interrupt handling. The need for such an adapter emerges with the IoT project development where it requires to support multiple interrupts as well as a simplified method to decode and handle these events.

NOTE

Currently this module supports interrupts in port 1 and 2. This hardware configuration does not support other ports.

This module provides users the following functions:

- `wkup_ad_register_gpio`: This function adds a new entry to the `wkup_ad_entry` table. Available parameters are:

- `sel_port`, `sel_pins`, `pol_pins`: Register the port, pin, and polarity of the interrupt, respectively.
- `cb`. is the callback function that is called when the interrupt is activated.
- `cb_inv`. is the callback function that that is called when the interrupt with reverse polarity is activated.

Some variables initialized by `wkup_ad_register_gpio` are used internally by the wakeup adapter and are members of `struct wkup_ad_entry`:

- `active`. The `cb` function only examines pins that are set as interrupt sources, thus speeding up the lookup process.
- `Triggered`. This variable will point out whether an interrupt is already served.
- `wkup_ad_init`: This function will initialize the `wkup_ad_entry` table. The size of this table and the number of available interrupts is declared in `MAX_IRQ_ENTRIES` definition.
- `wkup_ad_remove_gpio`: This function will remove a specific entry from the table that matches the pin and port conditions.
- `wkup_ad_clear_all_gpio`: This function will clear all entries on the `wkup_ad_entry` table and further disable all active interrupts.
- `wkup_ad_cb`: This is the main callback function of the module. When an interrupt is triggered:
 - All registered interrupts are checked for their `active` flag.
 - Inspect matching port and pin.
 - Look for the polarity settings of this interrupt.
 - Check triggered status
 - Eventually the appropriate callback function is executed, returning the port and pin that cause the interrupt. The `sensors_interface` module uses this information to locate the sensor that performs the `read` function.

4.5 Sensor Interface

4.5.1 General Description

The purpose of this module is to simplify the integration and operation of any sensors. Users who follows the described methods should ignore the sensor complexity and treat new sensors or the currently installed sensors with the same approach. With the **Sensor Interface** (SI) software module, users do not have to program complex implementations that directly use timers and the `wkup_adapter` module. An overview of this module is presented in [Figure 7](#).

The **SI** isolates the application code from the drivers and use the Driver Adaptation Layer which will be discussed in [4.5.3](#).

Two different lists of sensors are created:

- Timed-based sensors
- Event driven (GPIO) sensors

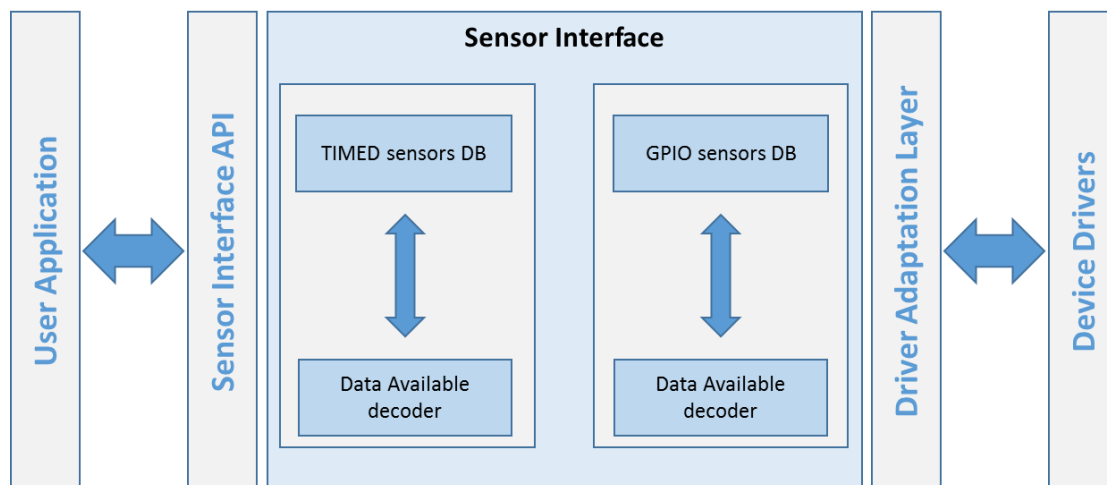


Figure 7: Sensor Interface Overview

Users can configure a sensor using the `si_config_t` structure and set the operation mode using the `si_sensor_operation_mode_t` parameter. The available values for this parameter are described in the following sections in detail.

4.5.1.1 Timer

Setting a sensor as "TIMER" falls in the "Timed-based sensors" category. This approach is used when users would like to access the sensor in periodic intervals appointed in the `periodic_read_interval` variable. When the `periodic_read_interval` expires, the callback function set in `read_fn` member is executed to retrieve the actual sensor data.

Most sensors need to perform sampling before the data are actually available for retrieval. For this reason, a second timer and a callback function are made available in the `si_config_t` configuration structure to handle this scenario:

- `force_read_fn`: Holds the callback function that initializes the measurement procedure at `read_delay` time before the actual read.
- `read_delay`: Specify the time period that `force_read_fn` is executed before the actual read is scheduled. Users shall give enough time for sampling to take place before actual data are read.

Implementing both timers and their functionality is illustrated in Figure 8.

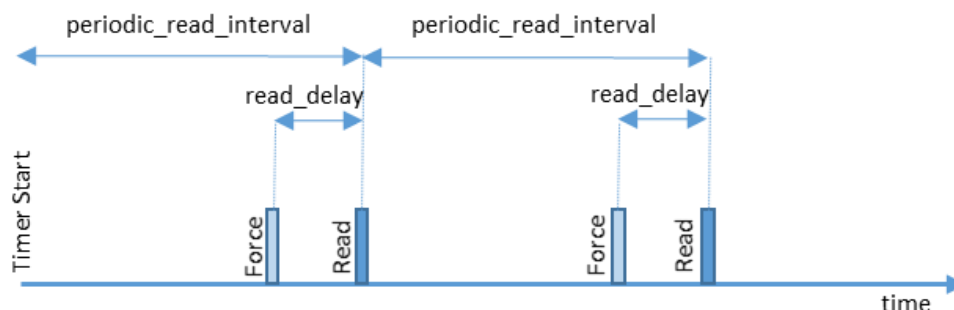


Figure 8: Timer Sensors Timeline

As an example, the VCNL4010 optical sensor is setup as a "TIMED" sensor. A callback function set in `force_read_fn` and `read_delay` is used to initiate measurements of ambient light and proximity values. A `read` callback function and `periodic_read_interval` timer is used to read the actual data collected from the sampling mode.

4.5.1.2 INTERRUPT

This operation mode falls in the Event-driven (GPIO) sensors category. It is a quite straightforward mode for INTERRUPT based sensors, as the sensor in question is only accessed when an interrupt event occurs. The interrupt hardware parameters (pin, port, and polarity), as well as an interrupt callback, must be registered in the `si_config` structure during the registration of the sensor.

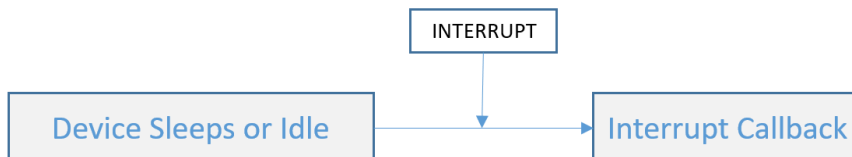


Figure 9: INTERRUPT Sensor Access Sequence

In this reference design, this mode is used with the ICM42605 accelerometer sensor.

4.5.1.3 FORCED

This mode does not fall into the categories above. A sensor of this type is usually "free running", taking samples at a rate previously set during the sensor configuration. Users can freely issue an `SI_READ_CMD` command (through the SI API) to retrieve the sensor data.

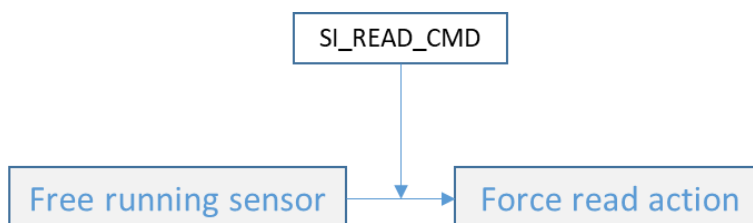


Figure 10: Forced Read Sequence

In this reference design, this mode is used with the AK099XX magnetometer sensor.

4.5.1.4 FORCED_INTER_SINGL_SHOT

"Forced Interrupt Single Shot" is a hybrid of `FORCED` and `INTERRUPT` mode. When registering a sensor with an operation mode of `FORCED_INTER_SINGL_SHOT`, an interrupt is registered (the same way as in section 4.5.1.2).

Initially, the sensor will be set in sampling mode. The registered interrupt will signal that data are available. To repeat the process, issue an `SI_FORCE_TO_READ_CMD` to reassign the device to the sampling mode.

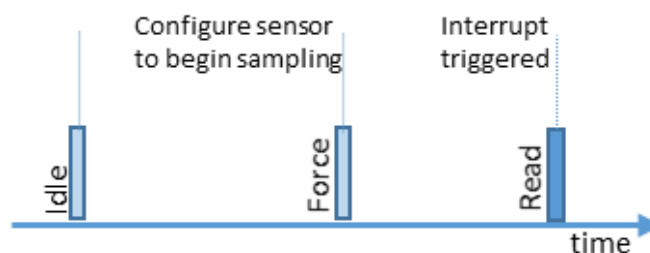


Figure 11: FORCED_INTER_SINGL_SHOT Functionality

In this reference design, this mode is used with the AK099XX magnetometer sensor.

4.5.2 Sensor Interface API

The "Sensor Interface API" is part of the "Sensor Interface" module shown in Figure 7 and provides the following functionalities to users:

- `si_reset()`
 - The reset operation clears all information in both TIMED and GPIO sensors table in the "SI" module.
 - This action takes place when all sensors are suspended to avoid unintentional operation of any devices. On system startup this information is already initialized.
- `si_register_sensor()`
 - This command registers a new sensor in the module.
 - The sensor registration procedure in SI module is shown in Figure 12.

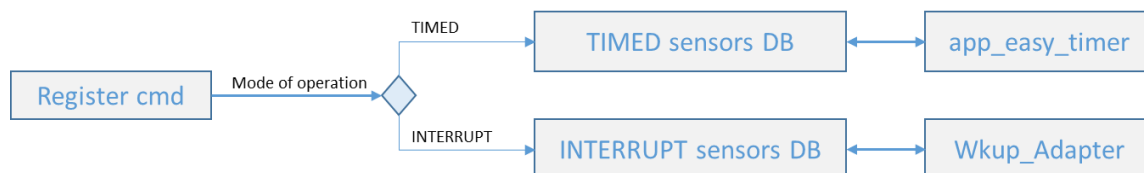


Figure 12: SI Registration Path

In the SI registration path, the first thing to inspect is the mode of operation. After this, each device will be registered in the corresponding database (DB). TIMED based sensors will further use the "app_easy_timer" library. INTERRUPT based sensors will register necessary data to the "wkup_adapter" module that keeps track of all HW interrupts.

- `si_send_command()`:
Provided that a sensor is already registered, the following commands are available:
 - `SI_FORCE_TO_READ_CMD` immediately executes the user-defined callback function `force_read_fn`. This operation applies to TIMED based sensors.
 - `SI_READ_CMD` immediately executes the user-defined callback function `read_fn`. This operation applies to INTERRUPT based sensors.
- `SI_STOP_SENSOR_CMD` disables the sensor.

Each device type has separate and common configuration parameters, all of which are members of the `si_config` struct.

- Available settings for TIMED devices include:
 - `operation_mode`: This value will define the sensor as a TIMED device and should be set as `TIMER`.
 - `read_delay`: The amount of time when users apply a forced mode to the device prior to reading the actual data.
 - `periodic_read_interval`: The amount of time that users expect to read data from the device.
 - `force_read_fn`: A callback function defined by users to setup the device in forced mode.
- Available settings for INTERRUPT devices:
 - `operation_mode`: This value will define the sensor as an INTERRUPT device and should be set as `INTERRUPT`.
 - `sensors_pin_conf`: Users should provide the pin, port, and polarity settings for the "wkup_adapter" to enable the interrupt.
- Common settings:

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- "Driver specific settings": Users are required to pass all necessary device specific information to the driver layer using this variable.
- `sensor_id`: Each sensor should have a unique ID.
- `set_sensor_config`: A user-defined callback function to setup the device in driver space. This function uses the previously defined driver specific settings.
- `data_size`: The actual size of the data that are expected during a read operation.
- `read_fn`: A user defined callback function to read the actual data from the device.
- `pre_data_read_fn`: A user defined callback function that is executed before the normal read.

4.5.3 Driver Adaptation Layer

"Driver Adaptation Layer (DAL)" is a middleware for the SI to interconnect directly to the device drivers. The DAL shall provide certain hook functions that are common in the SI for all device types. These functions are described below:

- `<device>_setup`: This function uses the "driver specific settings" to configure the desirable operation of the device.
- `<device>_force_read`: this function puts the device in forced mode.
- `<device>_read`: this function retrieves data from the device.
- `<device>_disable`: this function stops the device operation and put it in minimum consumption mode. This is currently not supported by the SI module and should be called from `user_periph_sensors_suspend()` function.

4.6 Device Drivers

4.6.1 Environmental Sensor

BOSCH BME 680 is an environmental sensor capable of:

- Measuring pressure
- Measuring humidity
- Measuring temperature
- Detecting volatile organic compounds

DA14585 IoT MSK reference application also includes the "Bosch Software Environmental Cluster" library that provides IAQ output.

Used functions:

- `bme680_init` performs software reset as well as reads the chip-ID and calibration data from the sensor.
- `bme680_set_sensor_settings` is used to set the oversampling, filter and temperature, pressure, humidity, and gas selection settings in the sensor.
- `bme680_get_profile_dur` retrieves the profile duration of the sensor.
- `bme680_set_sensor_mode` sets the power mode of the sensor. In this implementation, two modes, forced and sleep, are used.
- `bme680_get_sensor_data` returns the sensor data with compensated values.

These driver files are located in "projects\target_apps\common\src\drivers\bme680".

Communication interface: I2C

4.6.2 Motion Sensor

4.6.2.1 TDK ICM-42605

The TDK ICM-42605 motion sensor, used in this reference design, is a 6-axis motion tracking device with the following features:

- 3-axis gyroscope
- 3-axis accelerometer
- User-programmable interrupts
- Wake-on-motion interrupt for low power operation of applications processor

Used functions:

- `SetupInvDevice426` sets up the communication interface of the device, FIFO size, and callback functions to handle data read.
- `ConfigureInvDevice` is used to configure the sampling rate and sensitivity of each sensor.
- `HandleInvDeviceFifoPacket426` is the callback function that retrieves available data from the sensor.
- `wom_acc_setup` pauses sensor operations and puts ICM-42605 in "Wake-on-Motion" state with minimum power consumption.
- `icm426xxInterrupt_wom_cb` is the callback function that is executed right after a motion event.
- `DisableBothModules` set both sensors (accelerometer and gyroscope) in low power mode to reduce consumption.

These driver files are located in "projects\target_apps\common\src\drivers\icm426xx".

Communication interface: SPI

4.6.2.2 BOSCH BMI160

This reference design also provides support for the BOSCH BMI160 motion sensor. To enable the BMI160 sensor, users should un-define `ICM4XX_ACCEL_SENSOR_AVAILABLE` and define `BMI160_ACCEL_SENSOR_AVAILABLE` in `da1458x_basic.h`.

Among others the BMI160 has the following features:

- 16-bit digital, triaxial accelerometer
- 16-bit digital, triaxial gyroscope
- Integrated interrupts for enhanced autonomous motion detection

Used functions:

- `setup_bmi160()` setups the communication interface of the device, FIFO size, and callback functions to handle data read. It calls the following:
 - `set_bmi160_init()` is a helper function used to set up the serial interface, SPI read/write functions, and delay functions of the module.
 - `set_bmi160_operating_mode()` is a helper function used to set up rate, range, and power mode of the sensors.
 - `set_bmi160_fifo_watermark_interrupt()` sets up the sensor watermark level.
- `handle_bmi_fifo_packet160()` is the callback function that retrieves available data from the sensor via SPI.
- `wom_acc_setup` pauses sensor operations and puts BMI160 in "Wake-on-Motion" state with minimum power consumption.
- `bmi160_interrupt_wom_cb()` is the callback function that is executed right after a motion event.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- `DisableBothModules()` sets both sensors (accelerometer and gyroscope) in low power mode to reduce consumption.

These driver files are located in "projects\target_apps\common\src\drivers\bmi160".

Communication interface: SPI.

4.6.3 Magneto Sensor

Asahi Kasei AK09915 is 3-axis electronic compass sensor with highly sensitive Hall-sensor technology.

Used functions:

- `ak09915_init` configures operation modes of the sensor: single measurement, continuous, and power down.
- `ak09915_single_measurement_mode` retrieves data from the sensor.
- `ak09915_power_down_and_stop` stops all activities and sets the sensor to low power mode to reduce consumption.

These driver files are located in "projects\target_apps\common\src\drivers\ak09915".

Communication interface: SPI.

4.6.4 Optical Sensor

VISHAY VCNL4010 is an optical sensor with the following features:

- Proximity sensor
- Ambient light sensor
- Built-in infrared emitter
- Programmable LED drive current

Used functions:

- `vcnl4010_config` determines which sensor is enabled (proximity, ambient, or both) and set the power level of the infrared emitter.
- `vcnl4010_set_force_mode_fn` puts the sensor into forced mode.
- `vcnl4010_read_after_force_fn` retrieves data from the sensor.
- `vcnl4010_disable_sensor` sets the sensor to low power mode to reduce consumption.

These driver files are located in "projects\target_apps\common\src\drivers\vcnl4010".

Communication interface: I2C.

4.6.5 GPIO Expander

ON Semiconductor FXL6408UMX is a low power GPIO expander with the following features:

- Eight independently configurable I/O ports.
- Low-power quiescent current of 1.5 μ A.

This device handles the three LEDs as well as the power output of the power amplifier.

Used functions:

- `set_ctrl_pwm_bp` sets range extender PWM bypass. For more information please refer to [6].
- `clr_ctrl_pwm_bp` clears range extender PWM bypass. For more information please refer to [6].
- `init_ext_gpio` initializes the driver.
- `set_led` turns on one single LED or all LEDs.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- `clr_led` turns off one single LED or all LEDs.

These driver files are located in "projects\target_apps\common\src\drivers\gpio_extender".

User functions are located in the "user_iot_dk_utils.c" file.

Communication interface: I2C.

4.6.6 Power Amplifier

Skyworks SKY66111-11 is a fully integrated radio frequency Front End Module (FEM) designed for Smart Energy applications.

Used functions:

- No user API is available for this device. Power settings can be changed in "dal458x_config_basic.h" from `RANGE_EXT_MODE` to act as bypass or full power.

The driver files are located in "SDK_585\sdk\platform\driver\range_ext\sky66111".

4.7 Adding a New Sensor

To add a new sensor to the DA14585 IoT MSK, it is recommended for developers to follow the following steps:

1. Create a new folder in the `common\src\drivers` directory with the device name. Place all files needed to drive the new sensor in the new folder. Use a file with the name `<xxx>_impl` to summarize all actions that are relevant to the DAL calls. In the `Driver Adaptation Layer` folder, place a file related to this sensor to provide the specific functionality described in section 4.5.3.
2. Edit the `user_periph_setup` source and header files to support the hardware connection parameters for the new device. Most common parameters are the ports and pins connections to the DA14585.
3. In the `user_sensors.c` file, add various routines to handle the device in user space:
 - a. Initialization function: it shall contain all necessary information related to the driver space and SI on how the new device is setup for operation. The format of this routine is similar to `user_sensor_<xxx>_init()`.
 - b. Data callback function: include a routine with the following format `void user_<xxx>_data_cb(uint8_t *data_ptr, uint16_t *data_size)` to handle data retrieval from the driver layer. Preserve the parameters format where `*data_ptr` points to the memory location of the data and `*data_size` indicates the size of the data.
 - c. Disable function: `user_periph_sensors_suspend()` contains the callback functions to disable all available sensors.
 - d. Translation functions: these are for compatibility reasons and are entirely up to the user.
4. In `user_sensors.h` add the new sensor to obtain a different ID needed by the `sensor_interface` to distinguish the sensor entities.
5. The `sensors_periph_interface.c` file contains routines to access both SPI and I2C interface peripherals. If the new sensor requires a different approach, users can add their own routines in this file for this purpose.
6. In the `user_dws_reports.h` file, declare a report and sensor type ID for the new device. To complete this action, in the `user_sensors.c` file the `user_prepare_<xxx>_data` functions must include the data related to sensor.

4.8 Sequence Diagrams

This section outlines the sequence of events and processes when data is transferred from the sensors to the GATT in Dialog's wearable devices. The sequence diagrams also describe the data reporting from sensor fusion and environmental sensors.

4.8.1 Sensor Fusion Data Reporting

Generation of sensor fusion data is triggered by the "Data ready" IRQ signal of an inertial sensor (accelerometer or gyroscope). An inertial sensor is registered in the **SI** as an interrupt driven device, hence the **SI** calls the read function of the DAL for sampling the sensor data and the application callback function to deliver sampled data to the application.

The application parses the obtained inertial sensor data and stores them in a dynamically allocated area. It combines the first accelerometer/gyro sample with the stored magneto sensor sample and starts subsequent executions of SF algorithm in `user_sensor_fusion_process` function.

Then it adds the latest raw data samples of accelerometer/gyro/magneto sample and the SF result in a DWS report notification and passes it to GATT layer to be transmitted over to the BLE interface.

Finally, it initiates the magneto data sampling by issuing a `FORCE_TO_READ` command to **SI** through the `si_send_command` API function of the **SI**. **SI** sets the magneto sensor in single shot mode and waits for the "data ready" IRQ of the magneto sensor. When the interrupt is asserted, the **SI** will fetch the data and call the application callback, which then stores the magneto sample and combine it with the next bunch of inertial sensor samples.

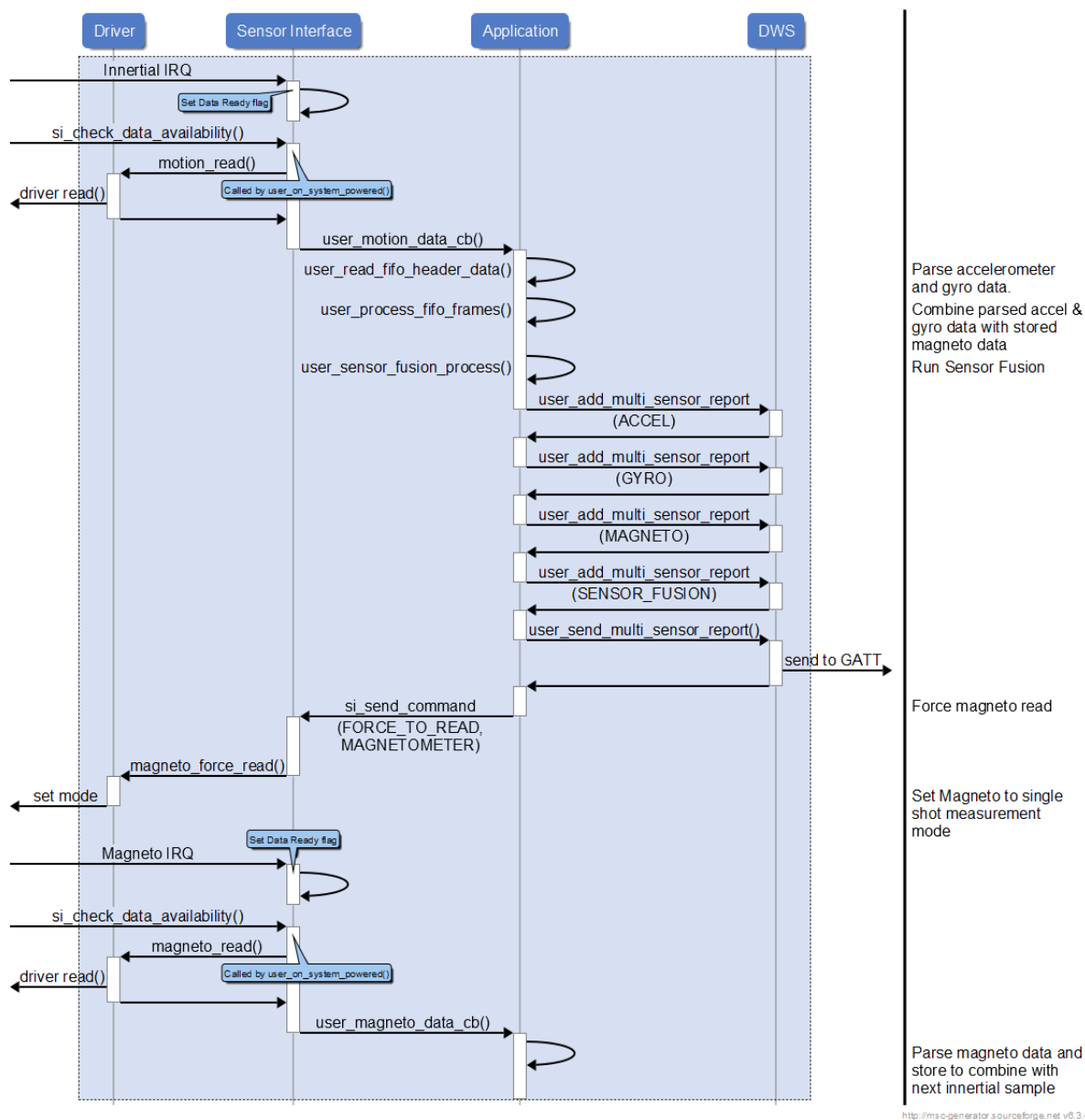


Figure 13: Sequence Diagram of Sensor Fusion Reporting

4.8.2 Environmental Data Reporting

An environmental sensor is registered in **SI** in **TIMER** mode. The initialization of measurement procedure and the read of the measured values are done in different time instances. Two functions of the adaptation layer of the environmental sensor are registered for this purpose:

`environmental_force_read()` to force read operation on the sensor and `environmental_read()` for reading sampled data.

The **SI** initiates two timer instances: `environmental_force_read()` is called on the expiration of the first timer to initiate the measurement procedure, and `environmental_read()` on the expiration of the second timer to read the measured data. When sensor data are available, the **SI** will call the application callback function.

If "volatile organic compounds" sample is available, the application will run the air quality algorithm and add reports for temperature/humidity/pressure sensors. The Air Quality classification results in a DWS notification message and sends it to GATT.

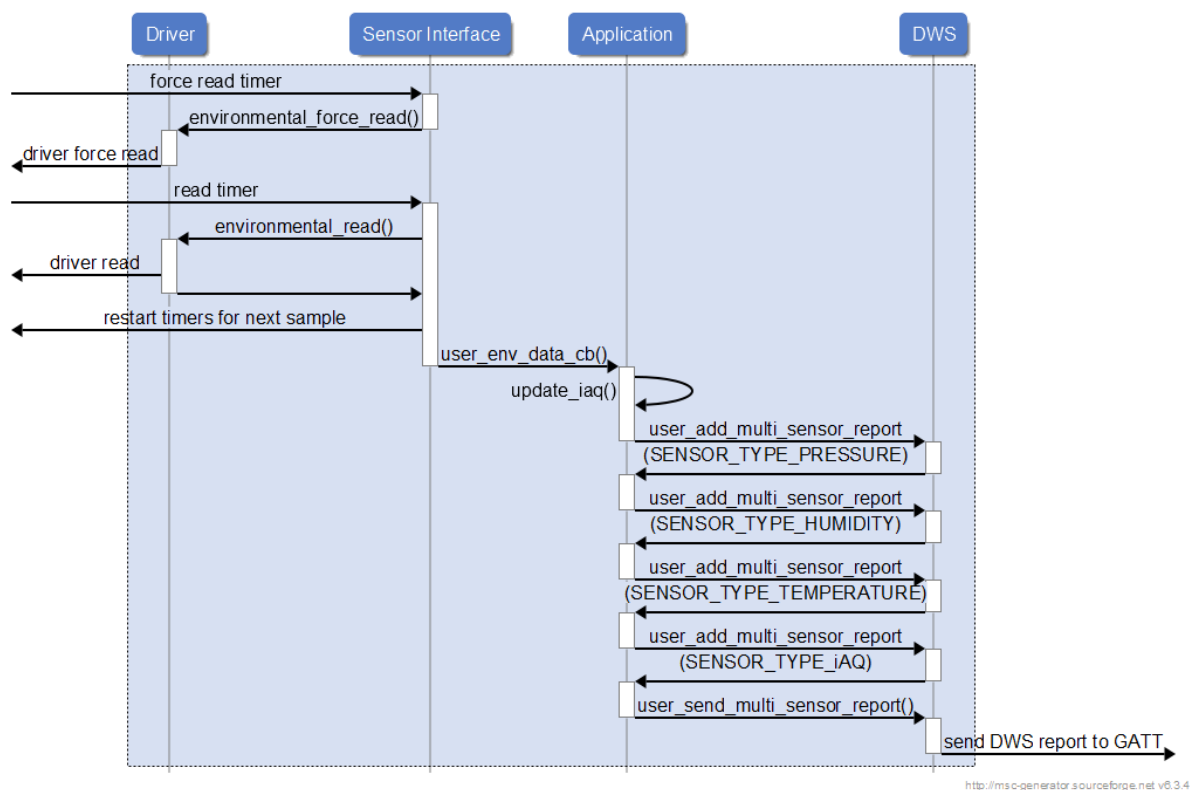


Figure 14: Sequence Diagram of Environmental Sensor Reporting

4.9 Dialog Wearable Service V2

The DA14585 IoT MSK reference application contains the Dialog Wearable Service version 2 (DWSv2) in order to transfer and control sensor data. This service includes several characteristics that are listed in Table 5.

The DWSv2 provides a means to:

- Transfer raw and calibrated sensor data.
- Transfer sensor fusion data.
- Configure the device, such as setting the operating parameters.
- Control the device, such as start/stop sensor operation and load/store data to non-volatile memory.

Table 5: DWSv2 Characteristics

Service/Characteristic	UUID	Properties (Note 1)	Size (B)	Description
wrbl_dws_svc	2EA7-8970-7D44-4BB-B097-2618-3F40-2400	RD	16	Service attribute
wrbl_dws_accel_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2401	NTF	25	Accelerometer Report. Legacy DWS compatibility, not used.
wrbl_dws_gyro_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2402	NTF	25	Gyroscope Report. Legacy DWS compatibility, not used.

Service/Characteristic	UUID	Properties (Note 1)	Size (B)	Description
wrbl_dws_mag_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2403	NTF	25	Magnetometer Report. Legacy DWS compatibility, not used.
wrbl_dws_baro_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2404	NTF	25	Barometer Report. Legacy DWS compatibility, not used.
wrbl_dws_hum_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2405	NTF	25	Humidity Report. Legacy DWS compatibility, not used.
wrbl_dws_temp_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2406	NTF	25	Temperature Report. Legacy DWS compatibility, not used.
wrbl_dws_sensf_char	2EA7-8970-7D4444BB-B097-2618-3F40-2407	NTF	25	Sensor Fusion Report. Legacy DWS compatibility, not used.
wrbl_dws_feat_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2408	RD	25	Device Features
wrbl_dws_control_char	2EA7-8970-7D44-4BB-B097-2618-3F40-2409	WR	32	Control Point
wrbl_dws_control_reply_char	2EA7-8970-7D44-4BB-B097-26183F40-240A	NTF	32	Command Reply
wrbl_dws_multi_sens_char	2EA7-8970-7D44-4BB-B097-26183F40-2410	NTF	109	Sensors Report

Note 1 RD: read, WR: write, NTF: notify, IND: indicate.

4.9.1 Feature Report Structure

Upon connection, the central device reads the feature characteristic (`wrbl_dws_feat_char`) in order to determine the capabilities and the firmware version of the connected device.

Table 6: Features Report Structure

Offset (B)	Name	Description
0	accelerometer_en	0: Accelerometer not present 1: Accelerometer present
1	gyro_en	0: Gyroscope not present 1: Gyroscope present
2	magn_en	0: Magnetometer not present 1: Magnetometer present
3	pressure_en	0: Barometer not present 1: Barometer present
4	humidity_en	0: Humidity sensor not present 1: Humidity sensor present
5	temp_en	0: Temperature sensor not present 1: Temperature sensor present
6	s_fusion_en	0: Sensor fusion capability not present 1: Sensor fusion capability present

Offset (B)	Name	Description
7 to 22	version[]	Version number, ASCII
23	device_type	Device version: 0: DA14580 IoT 2: DA14585 IoT MSK

4.9.2 Multi Sensor Report and Sensor Report

All sensor data are encapsulated in reports named "sensor reports" specific for each sensor type. The sensor reports are concatenated into a multi sensor report using `wrbl_dws_multi_sens_char` and the multi sensor report is transferred as a BLE notification to the central device.

Table 7: Multi Sensor Report

Preamble (1 Byte)	Timestamp (1 Byte)	Sensor Reports (Up to 107 bytes)
Always 0xA5	Integer number that increments after each report.	Concatenated sensor reports (Table 8).

Table 8: Sensor Report

Report ID (1 Byte)	Sensor State (1 Byte)	Calibration State (1 Byte)	Sensor Data (N Bytes)
1 to 24 (Table 9)	Value depending on sensor type.	Value depending on sensor type.	Depends on sensor type.

Table 9: Report Types/Report ID's

Report ID	Report Type
1	ACCELEROMETER_REPORT_ID
2	GYROSCOPE_REPORT_ID
3	MAGNETOMETER_REPORT_ID
4	PRESSURE_REPORT_ID
5	HUMIDITY_REPORT_ID
6	TEMPERATURE_REPORT_ID
7	SENSOR_FUSION_REPORT_ID
8	COMMAND_REPLY_REPORT_ID
9	AMBIENT_LIGHT_REPORT_ID
10	PROXIMITY_REPORT_ID
11	GAS_REPORT_ID
12	iAQ_REPORT_ID
13	BUTTON_REPORT_ID
14	VELOCITY_DELTA_REPORT_ID
15	EULER_ANGLE_DELTA_REPORT_ID
16	QUATERNION_DELTA_REPORT_ID
17+	Reserved

4.9.2.1 Sensor Report for Accelerometer, Gyroscope, and Magnetometer

Table 10: Report Structure for Accelerometer, Gyroscope, and Magnetometer

Report field	Field size (B)	Description
ucReportId	1	1, 2, or 3 (Table 9)
snsr_state	1	Sensor state (snsr_state, see Table 11)
cal_state	1	Calibration state used only for magnetometer. 0: Calibration State Disabled 1: Calibration State Initialized 2: Calibration State Bad 3: Calibration State OK 4: Calibration State Good 5: Calibration State Error
val_x	2	X axis value for the selected sensor
val_y	2	Y axis value for the selected sensor
val_z	2	Z axis value for the selected sensor

Table 11: Bitfield Structure for snsr_state

Bit(s)	Field	Description
0	in_data_valid	Input (pre-calibration) data valid flag
1	out_data_valid	Output (post calibration) data valid flag
2	cal_enabled	Calibration enabled flag
3	cal_settled	Calibration settled flag
4	cal_converged	Calibration converged flag
5:7	cal_mode	Calibration mode

4.9.2.2 Sensor Report for Temperature, Humidity, Gas, and Barometric Pressure

Table 12: Environmental Sensor Report

Report field	Field size (B)	Description
ucReportId	1	4, 5, 6, 11 (Table 9)
ucSensorState	1	Always 2 (Sensor Ready)
ucSensorEvent	1	Always 3 (Update Value)
Val32	4	Sensor value

4.9.2.3 Sensor Report for Indoor Air Quality (IAQ)

Table 13: Indoor Air Quality (IAQ) Report

Report field	Field size (B)	Description
ucReportId	1	12 (Table 9)
ucSensorState	1	Always 2 (Sensor Ready)

Report field	Field size (B)	Description
ucSensorEvent	1	Accuracy 0-3: 0: Unreliable 1: Low Accuracy 2: Medium Accuracy 3: High Accuracy
Val32	4	Sensor value 0-500

4.9.2.4 Sensor Report for Ambient Light and Proximity

Table 14: Sensor Report for Ambient Light and Proximity

Report field	Field size (B)	Description
ucReportId	1	9 and 10 (Table 9)
ucSensorState	1	Always 2 (Sensor Ready)
ucSensorEvent	1	Always 3 (Update Value)
Val32	4	Sensor value

4.9.2.5 Sensor Report for Button

Table 15: Sensor Report for Button

Report field	Field size (B)	Description
ucReportId	1	13 (Table 9)
ucSensorState	1	Button Status 0-1: 0: Released 1: Pressed
ucSensorEvent	1	Always 3 (Update Value)
Val32	4	Reserved

4.9.2.6 Sensor Report for Sensor Fusion

Table 16: Sensor Report for Sensor Fusion

Report field	Field size (B)	Description
ucReportId	1	7 (Table 9)
ucSensorState	1	Always 2 (Sensor Ready)
mcal_state	1	Magnetometer calibration state: 0: Calibration State Disabled 1: Calibration State Initialized 2: Calibration State Bad 3: Calibration State OK 4: Calibration State Good 5: Calibration State Error
val_w	2	W sensor fusion value
val_x	2	X sensor fusion value
val_y	2	Y sensor fusion value
val_z	2	Z sensor fusion value

4.9.2.7 Sensor Report for Velocity Delta

Table 17: Sensor Report for Velocity Delta

Report field	Field size (B)	Description
ucReportId	1	14 (Table 9)
snsr_state	1	Accelerometer state (see Table 11)
q_format	1	Q format of δV data
val_x	2	X axis value for δV data
val_y	2	Y axis value for δV data
val_z	2	Z axis value for δV data

4.9.2.8 Sensor Report for Euler Angle Delta

Table 18: Sensor Report for Euler Angle Delta

Report field	Field size (B)	Description
ucReportId	1	15 (Table 9)
snsr_state	1	Gyroscope state (see Table 11)
q_format	1	Q format of $\delta \Theta$ data
val_x	2	X axis value for $\delta \Theta$ data
val_y	2	Y axis value for $\delta \Theta$ data
val_z	2	Z axis value for $\delta \Theta$ data

4.9.2.9 Sensor Report for Quaternion Delta

Table 19: Sensor Report for Quaternion Delta

Report field	Field size (B)	Description
ucReportId	1	16 (Table 9)
ucSensorState	1	Always 2 (Sensor Ready)
mcal_state	1	n/a
val_w	2	W value for δQ data
val_x	2	X value for δQ data
val_y	2	Y value for δQ data
val_z	2	Z value for δQ data

4.9.3 Report Structures for Configuration and Control

The DWSv2 provides the `wrbl_dws_control_char` (WR) and `wrbl_dws_control_reply_char` (NTF) characteristics for configuring and controlling the device. The device may also send unsolicited messages (such as STOP) to signal events.

Typically, the central device issues a command using the control characteristic and waits for a reply from the notification reply characteristic. The replies issued by the IoT sensor always start with byte 0x08 (COMMAND_REPLY_REPORT_ID, omitted from the following tables), followed by the Command ID and the command data.

Table 20: Report Structure for Commands

Report ID (1 Byte)	Command ID (1 Byte)	Command Data (N Bytes)
COMMAND_REPLY_REPORT_ID =8	See following tables.	Depending on Command ID, varies in length and field types.

4.9.3.1 Start Command

Table 21: Start Command

Offset (B)	Description	Value
0	Command ID	1

Table 22: Start Command Reply

Offset (B)	Description	Value
0	Command ID	1
1	Running Status	1: Running

4.9.3.2 Stop Command

Table 23: Stop Command

Offset (B)	Description	Value
0	Command ID	0

Table 24: Stop Command Reply

Offset (B)	Description	Value
0	Command ID	0
1	Running Status	0: Stopped

4.9.3.3 Read Parameters from Flash Memory

Table 25: Read Flash Command

Offset (B)	Description	Value
0	Command ID	2

4.9.3.4 Reset to Factory Defaults

Table 26: Reset to Defaults (RtD) Command

Offset (B)	Description	Value
0	Command ID	3

4.9.3.5 Store Basic Configuration in Flash Memory

Table 27: Store Basic Configuration Command

Offset (B)	Description	Value
0	Command ID	4

4.9.3.6 Store Calibration Coefficients and Control Configuration in Flash Memory

Table 28: Store Calibration and Control Command

Offset (B)	Description	Value
0	Command ID	5

4.9.3.7 Return Running Status

Table 29: Return Running Status Command

Offset (B)	Description	Value
0	Command ID	6

Table 30: Return Running Status Reply

Offset (B)	Description	Value
0	Command ID	6
1	Running Status	0: Stopped 1: Running

4.9.3.8 Reset Sensor Fusion and Calibration Configuration

Table 31: Reset Sensor Fusion and Calibration Configuration command

Offset (B)	Description	Value
0	Command ID	7

4.9.3.9 Basic Configuration

Table 32: Basic Configuration Command

Offset (B)	Description	Value
0	Command ID	10
1	Sensor Combination	Bit 0: Accelerometer Enable Bit 1: Gyroscope Enable Bit 2: Magnetometer Enable Bit 3: Environmental Sensor Enable Bit 4: Gas Sensor Enable Bit 5: Proximity Sensor Enable Bit 6: Ambient Light Sensor Enable Note: For SF to operate, only certain combinations are allowed regarding accelerometer, gyroscope, and magnetometer. The combinations allowed are: <ul style="list-style-type: none"> • Gyroscope only. • Gyroscope and accelerometer. • Accelerometer and magnetometer. • Accelerometer, gyroscope, and magnetometer.

Offset (B)	Description	Value
2	Accelerometer Range	0x03: 2 G 0x05: 4 G 0x08: 8 G 0x0C: 16 G
3	Accelerometer Rate	0x06: 1 kHz (ICM42605) or 800 Hz (BMI160) 0x07: 200 Hz 0x08: 100 Hz 0x09: 50 Hz 0x0A: 25 Hz
4	Gyroscope Range	0x00: 2000 deg/s 0x01: 1000 deg/s 0x02: 500 deg/s 0x03: 250 deg/s 0x04: 125 deg/s
5	Gyroscope Rate	0x06: 1 kHz (ICM42605) or 800 Hz (BMI160) 0x07: 200 Hz 0x08: 100 Hz 0x09: 50 Hz 0x0A: 25Hz
6	Magnetometer Rate	Valid only if SF is off. 0: Accelerometer ODR/1 1: Accelerometer ODR/2 3: Accelerometer ODR/4 7: Accelerometer ODR/8
7	Environmental Sensors Rate	1: 0.33 Hz 2: 0.5 Hz 4: 1 Hz 6: 2 Hz Note: Only option 1 is available when the gas sensor is enabled.
8	Sensor Fusion Rate	0: SF Off 10: 10 Hz (not applicable for BMI160 when the rate is 800Hz). 25: 25 Hz 50: 50 Hz 100: 100 Hz
9	Sensor Fusion Mode	0: Sensor Fusion (AHRS) Off 1: Sensor Fusion (AHRS) On
10	Sensor Fusion Raw Data Enable	0: Disabled 1: Enabled Raw Data (decimated at SF ODR) 2: Enabled Integration Engine Data
11	Reserved	NA
12	Gas Sensor rate	Not used. The gas sensor rate is always 0.33 Hz (Low Power Mode).
13	Proximity/Ambient Light Mode	Not used. The mode is always polled.

Offset (B)	Description	Value
14	Proximity/Ambient Light Rate	0: Sensor Off 1: 10 Hz 2: 5 Hz 3: 2 Hz 4: 1 Hz 5: 0.5 Hz 6: 0.2 Hz

4.9.3.10 Read Basic Configuration

Table 33: Read Basic Configuration Command

Offset (B)	Description	Value
0	Command ID	11

Table 34: Read Basic Configuration Command Reply

Offset (B)	Description	Value
0	Command ID	11
1	Sensor Combination	Bit 0: Accelerometer Enable Bit 1: Gyroscope Enable Bit 2: Magnetometer Enable Bit 3: Environmental Sensor Enable Bit 4: Gas Sensor Enable Bit 5: Proximity Sensor Enable Bit 6: Ambient Light Sensor Enable Note: For SF to operate, only certain combinations are allowed regarding accelerometer, gyroscope, and magnetometer. The combinations allowed are: <ul style="list-style-type: none"> • Gyroscope only. • Gyroscope and accelerometer. • Accelerometer and magnetometer. • Accelerometer, gyroscope and magnetometer.
2	Accelerometer Range	0x03: 2 G 0x05: 4 G 0x08: 8 G 0x0C: 16 G
3	Accelerometer Rate	0x06: 1 kHz (ICM42605) or 800 Hz (BMI160) 0x07: 200 Hz 0x08: 100 Hz 0x09: 50 Hz 0x0A: 25Hz

Offset (B)	Description	Value
4	Gyroscope Range	0x00: 2000 deg/s 0x01: 1000 deg/s 0x02: 500 deg/s 0x03: 250 deg/s 0x04: 125 deg/s
5	Gyroscope Rate	0x06: 1 kHz (ICM42605) or 800 Hz (BMI160) 0x07: 200 Hz 0x08: 100 Hz 0x09: 50 Hz 0x0A: 25Hz
6	Magnetometer Rate	Valid only if SF is off. 0: Accelerometer ODR/1 1: Accelerometer ODR/2 3: Accelerometer ODR/4 7: Accelerometer ODR/8
7	Environmental Sensors Rate	1: 0.33 Hz 2: 0.5 Hz 4: 1 Hz 6: 2 Hz
8	Sensor Fusion Rate	0: SF Off 10: 10 Hz 25: 25 Hz 50: 50 Hz 100: 100 Hz
9	Sensor Fusion Mode	Reserved
10	Sensor Fusion Raw Data Enable	0: Disabled 1: Enabled (RAW) 2: Enabled Integration Engine
11	Reserved	NA
12	Gas Sensor rate	Not used. The gas sensor rate is always 0.33 Hz (Low Power Mode).
13	Proximity/Ambient Light Mode	Not used. The mode is always polled.
14	Proximity/Ambient Light Rate	0: Sensor Off 1: 10 Hz 2: 5 Hz 3: 2 Hz 4: 1 Hz 5: 0.5 Hz 6: 0.2 Hz

4.9.3.11 Set Sensor Fusion Coefficients Command

Table 35: Set Sensor Fusion Coefficients Command

Offset (B)	Description	Value
0	Command ID	12
1	BETA A LSB	Sensor Fusion Beta A Gain (LSB)

Offset (B)	Description	Value
2	BETA A MSB	Sensor Fusion Beta A Gain (MSB)
3	BETA M LSB	Sensor Fusion Beta M Gain (LSB)
4	BETA M MSB	Sensor Fusion Beta M Gain (MSB)
5:8	TEMPERATURE_REPORT_ID	Reserved

4.9.3.12 Read Sensor Fusion Coefficients

Table 36: Read Sensor Fusion Coefficients Command

Offset (B)	Description	Value
0	Command ID	13

Table 37: Read Sensor Fusion Coefficients Command Reply

Offset (B)	Description	Value
0	Command ID	13
1	BETA A LSB	Sensor Fusion Beta A Gain (LSB)
2	BETA A MSB	Sensor Fusion Beta A Gain (MSB)
3	BETA M LSB	Sensor Fusion Beta M Gain (LSB)
4	BETA M MSB	Sensor Fusion Beta M Gain (MSB)
5:8	Reserved	NA

4.9.3.13 Set Calibration Coefficients

Table 38: Set Calibration Coefficients Command

Offset (B)	Description	Value
0	Command ID	14
1	Sensor Type	2: Magnetometer
2	Q Format	Integer value
3:8	Offset Vector (3 × int16)	Integer value
9:26	Matrix 3 × 3 × int16	Signed fixed point value

4.9.3.14 Read Calibration Coefficients

Table 39: Read Calibration Coefficients Command

Offset (B)	Description	Value
0	Command ID	15

Table 40: Read Calibration Coefficients Command Reply

Offset (B)	Description	Value
0	Command ID	15
1	Sensor Type	Magnetometer = 2
2	Q Format	Integer value

Offset (B)	Description	Value
3:8	Offset Vector 3 × 1 int16	Integer value
9:26	Matrix 3 × 3 int16	Signed fixed point value

4.9.3.15 Set Calibration Control Flags

Table 41: Set Calibration Control Flags Command

Offset (B)	Description	Value
0	Command ID	16
1	Sensor Type	2: Magnetometer
2:3	Calibration Control Flags	Byte 2: see Table 42 Byte 3: see Table 43
4:15	Calibration Parameters	See Table 44

Table 42: Calibration Control Flags #1

	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Calibration Mode	Reserved	Reserved	Offset apply	Matrix apply	Offset update	Matrix update	Init from static	Offset post apply
Static	X	X	1: Yes	1: Yes	0: No	0: No	0: No	1: Yes
Basic Auto	X	X	1: Yes	1: Yes	1: Yes	1: Yes	0: No	1: Yes
SmartFusion Auto	X	X	1: Yes	1: Yes	1: Yes	1: Yes	1: Yes	1: Yes

Table 43: Calibration Control Flags #2

	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Calibration Mode	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Converged (read only)	Settled (read only)
Static	X	X	X	X	X	X	X	0: No
Basic Auto	X	X	X	X	X	X	0: No	1: Yes
SmartFusion Auto	X	X	X	X	X	X	1: Yes	1: Yes

Table 44: Calibration Parameters

Offset (B)	Static	Basic	SmartFusion	Description
0	Reserved	ref_mag	ref_mag	Reference magnitude
1	Reserved	mag_range	mag_range	Magnitude range
2	Reserved	mag_alpha	mag_alpha	Magnitude filter coefficient
3	Reserved	mag_delta_thresh	mag_delta_thresh	Magnitude gradient threshold
4	Reserved	Reserved	mu_offset	Offset update rate
5	Reserved	Reserved	mu_matrix	Matrix update rate
6	Reserved	Reserved	err_alpha	Overall error filter coefficient
7	Reserved	Reserved	err_thresh	Overall error threshold

4.9.3.16 Read Calibration Control

Table 45: Read Calibration Control Flags Command

Offset (B)	Description	Value
0	Command ID	17

Table 46: Read Calibration Control Flags Command Reply

Offset (B)	Description	Value
0	Command ID	17
1	Sensor Type	2: Magnetometer
2:3	Calibration Control Flags	Byte 2: see Table 42 Byte 3: see Table 43
4:15	Calibration Parameters	See Table 44

4.9.3.17 Fast Accelerometer Calibration

Table 47: Fast Accelerometer Calibration Command

Offset (B)	Description	Value
0	Command ID	18

Table 48: Fast Accelerometer Calibration Reply

Offset (B)	Description	Value
0	Command ID	18
1	Fast Calibration Status	0: Stopped 1: Started

4.9.3.18 Set Calibration Modes

Table 49: Set Calibration Modes Command

Offset (B)	Description	Value
0	Command ID	19
1	Calibration Mode for Accelerometer	Not used, reserved for future use.
2	Calibration Mode for Gyroscope	Not used, reserved for future use.
3	Calibration Mode for Magnetometer	0: None 1: Static 2: Continuous Auto 3: Auto One Shot
4	Auto Calibration Mode for Accelerometer	Not used, reserved for future use.
5	Auto Calibration Mode for Gyroscope	Not used, reserved for future use.
6	Auto Calibration Mode for Magnetometer	0: Basic Auto Calibration 1: Smart Auto Calibration

4.9.3.19 Read Calibration Modes

Table 50: Read Calibration Modes Command

Offset (B)	Description	Value
0	Command ID	20

Table 51: Read Calibration Modes Command Reply

Offset (B)	Description	Value
0	Command ID	20
1	Calibration Mode for Accelerometer	Not used, reserved for future use.
2	Calibration Mode for Gyroscope	Not used, reserved for future use.
3	Calibration Mode for Magnetometer	0: None 1: Static 2: Continuous Auto 3: Auto One Shot
4	Auto Calibration Mode for Accelerometer	Not used, reserved for future use.
5	Auto Calibration Mode for Gyroscope	Not used, reserved for future use.
6	Auto Calibration Mode for Magnetometer	0: Basic Auto Calibration 1: Smart Auto Calibration

4.9.3.20 Read Device Sensors

Table 52: Read Device Sensors Command

Offset (B)	Description	Value
0	Command ID	21

Table 53: Read Device Sensors Command Reply

Offset (B)	Description	Value
0	Command ID	21

Offset (B)	Description	Value
1-17	Sensor Type	Physical Sensors: 0: None 1: Accelerometer 2: Gyroscope 3: Magnetometer 4: Barometer 5: Humidity Sensor 6: Temperature Sensor 7: Ambient Light 8: Proximity Sensor 9: Button 10: RAW GAS 11: Proximity Calibration 12-24: Reserved Virtual Sensors: 64: Sensor Fusion 65: Integration Engine (IE) 66: Indoor Air Quality 67-74: Reserved

4.9.3.21 Read Software Version

Table 54: Read Application Software Version Command

Offset (B)	Description	Value
0	Command ID	22

Table 55: Read Application Software Version Command Reply

Offset (B)	Description	Value
0	Command ID	22
1-17	Version Number	Version number of the application in ASCII representation, for example, "v6.160.2"

4.9.3.22 Start LED Blink

Table 56: Start LED Blink Command

Offset (B)	Description	Value
0	Command ID	23

4.9.3.23 Stop LED Blink

Table 57: Stop LED Blink Command

Offset (B)	Description	Value
0	Command ID	24

4.9.3.24 Set Proximity Hysteresis Limits

Table 58: Set Proximity Hysteresis Limits Command

Offset (B)	Description	Value
0	Command ID	25
1-2	Proximity Low Limit (proximity off)	0-65535
3-4	Proximity Low Limit (proximity on)	0-65535

4.9.3.25 Read Proximity Hysteresis Limits

Table 59: Read Proximity Hysteresis Limits Command

Offset (B)	Description	Value
0	Command ID	26

Table 60: Read Proximity Hysteresis Limits Command Reply

Offset (B)	Description	Value
0	Command ID	26
1-2	Proximity Low Limit (proximity off)	0-65535
3-4	Proximity Low Limit (proximity on)	0-65535

4.9.3.26 Calibration Complete

The calibration complete notification is only sent from the device to the central application when a calibration operation is completed.

Table 61: Calibration Complete Notification

Offset (B)	Description	Value
0	Command ID	27
1	Sensor Type	0: Accelerometer 1: Gyroscope 2: Magnetometer
2	Status	0: OK 1: Error

4.9.3.27 Proximity Calibration Command

Table 62: Proximity Calibration Command

Offset (B)	Description	Value
0	Command ID	28

Table 63: Proximity Calibration Command Reply

Offset (B)	Description	Value
0	Command ID	28

Offset (B)	Description	Value
1	Start/Stop	1: Started 0: Ended

4.10 Sensor Calibration Library

4.10.1 Overview

The SmartFusion Sensor Calibration Library (SCL) provides a set of routines for calibrating microelectromechanical systems (MEMS) sensors, such as gyroscopes, accelerometers, and magnetometers. By applying these routines to captured sensor data, it is possible to compensate for the typically exhibited imperfections and distortion.

4.10.1.1 Modes of Operation

The routines provided by the SCL have been designed with flexibility in mind and support various modes of operation depending on the specific characteristics of the sensors and the system requirements.

The supported calibration modes are as follows:

- **Static Calibration Mode:**
 - When the sensor distortions are measurable, stable, and consistent between devices, static calibration is the preferred mode of operation as it gives the best performance in terms of distortion correction.
 - In this mode the calibration routine is initialized with static calibration coefficients which are then applied to the sensor data and do not change.
 - These static calibration coefficients are typically calculated off-line by the device manufacturer by analyzing recordings of raw sensor data made under controlled conditions. The calibration coefficients are stored in either the device's firmware or non-volatile memory.
- **Continuous Automatic Calibration Mode:**
 - When the sensor distortions are unstable and/or inconsistent between devices, continuous automatic calibration is the preferred mode of operation as it allows the calibration coefficients to be determined automatically at runtime without requiring them to be built into the firmware or programmed into non-volatile memory.
 - In this mode the auto-calibration function continually monitors the sensor data for distortions and adapts the calibration coefficients to compensate for them.
- **One-shot Automatic Calibration Mode:**
 - When the sensor distortions are relatively stable in the short term, one-shot auto-calibration mode may be preferable.
 - In this mode the auto-calibration function is run upon device startup to determine the sensor distortions but is disabled once the calibration is complete and suitable calibration coefficients have been calculated.

4.10.1.2 Calibration Routines

The SCL provides a number of routines for both static and automatic calibration of three-dimensional sensor data.

NOTE

Not all these routines are appropriate for all types of sensors or can be used in all calibration modes.

The supported calibration routines are as follows:

- **Static Calibration:**

- The static calibration routine applies user-defined static calibration coefficients in the form of a 3×3 transformation matrix and a 3-dimensional vector offset.
- This routine is suitable for all types of sensors but is only appropriate for use in static calibration mode as the coefficients do not change.
- **Basic Auto-Calibration:**
 - The basic auto-calibration routine monitors the data captured from sensors for basic offset and scaling distortions, calculating and applying appropriate calibration coefficients at runtime in order to compensate for them.
 - This routine is not designed to detect and compensate for the sophisticated types of distortions such as cross-axis, spherical and rotational distortions.
 - It presumes that the magnitude of the external stimulus to the sensor (such as magnetic or gravitational field strength) is constant and only varies according to device orientation.
 - Neither is it able to support scenarios where the distortions are not constant but vary over time.
- **SmartFusion Auto-Calibration:**
 - To overcome some of the shortcomings of the basic auto-calibration routine, a more sophisticated algorithm is also provided.
 - In addition to being able to calculate and compensate for basic offset and scaling distortions, this algorithm can also compensate for more sophisticated distortions such as magnetometer soft iron spherical distortions.
 - It is also able to cope with gradual changes in sensor distortions and external stimulus over time, although some adaptation time is required.
- **Static Drift Compensation:**
 - This routine has been specifically designed to reduce gyroscope drift and is not appropriate for use with any other type of sensors.
 - Gyroscopes typically exhibit small biases, indicating slow rotation even when the device is stationary. This results in drifts in the calculated orientation when the gyroscope data is integrated.
 - Although these biases typically have large static components that can be compensated for using static calibration, there is often a residual bias that varies over time due to temperature or gravitational effects.
 - The static drift compensation routine provides tracks and removes these dynamic biases, eliminating drift when the device is stationary.
 - The algorithm also includes a noise gate to eliminate drift induced by random walk due to noise.

4.10.1.3 Calibration Procedure

It requires sampling an external stimulus, such as a magnetic or gravitational field, at a wide range of different orientations for the basic and SmartFusion auto-calibration routines to operate correctly. Therefore, it is necessary to rotate the device to determine the distortions and calculate appropriate calibration coefficients. It is also important that these external stimuli remain constant in both magnitude and direction.

It is required to calibrate the magnetometer in a place where the magnetic field is reasonably strong and uniform. The rotation can be performed manually by randomly rotating the device until the calibration routine signals its completion.

It is important that the device is not subjected to any lateral movement while being rotated during the accelerometer calibration, as lateral movements will distort the sampling of the gravitational field. It is therefore recommended to use a gimbal in conjunction with these routines to rotate the accelerometer around its center without any lateral movement.

When using the device in an environment where the external stimulus is constantly changing, such as when the magnetic field fluctuates with position or the device is subjected to lateral accelerations,

it is recommended to perform an initial automatic calibration in one-shot mode under controlled situations and then switch to static calibration mode using the resultant calibration coefficients.

4.10.2 API Usage

The various calibration routines provided by the SCL are designed to work together and complement each other. For the common functionalities, the routines use generic controls, parameters, and code. The more advanced routines re-use the functionality of the basic routines. For example, the basic auto-calibration routine re-uses the static calibration routine to apply its calibration parameters to the sensor data.

More specifically, the parameter structures for all routines are designed to overlap so that they can share the same location in memory, thus sharing common parameters. The purpose is to aid switching between calibration modes without unnecessary copying of parameter data between different routines and to minimize the memory footprint.

Wrapper code (`sensor_calibration.h|c`) has been provided in the SDK, which implements the overlapping of the various calibration routines and provides a common interface through which the calibration routines can be used.

4.10.2.1 Allocation

An instance of the appropriate calibration parameter structure (`static_calibration_params`, `basic_autocal_params`, `smartfusion_autocal_params`, or `static_drift_compensation_params`) or combined wrapper instance structure (`cal_instance`) shall be instantiated, either statically or on the heap, and shall be maintained during the life-cycle of sensor calibration processing.

4.10.2.2 Initialization

Before processing can be performed on an instance of a calibration routine, a subset of parameters within the appropriate parameter structure must be initialized. These parameters are as follows:

- **Generic:** Common to all
 - `in_data`: Pointer from which to read raw sensor input data
 - `out_data`: Pointer to which to write processed sensor output data
- **Static calibration:** In addition to the generic parameters:
 - `offset`: Offset vector coefficients to be subtracted from sensor data in same representation as raw sensor data
 - `matrix`: 3×3 matrix of signed fixed point coefficients to apply to sensor data.
 - `q_format`: Q format of matrix coefficients
 - `flags`: Control flags
 - `apply`: Flag to control whether calibration coefficients are applied
 - `matrix_apply`: Flag to control whether matrix coefficients are applied in addition to offset
 - `offset_post_apply`: Flag to control whether vector coefficients are applied before or after the matrix coefficients are applied
- **Basic Auto-calibration:** In addition to the static calibration parameters:
 - `ref_mag`: Reference magnitude indicating expected geomagnetic field strength in same representation as raw sensor data
 - `mag_range`: Q15 unsigned fixed-point scaler indicating range +/- reference magnitude of valid sensor data
 - `mag_alpha`: Q15 unsigned fixed-point coefficient controlling the filtering applied to the calculated sensor vector magnitude
 - `mag_delta_threshold`: Q15 unsigned fixed-point threshold applied to the calculated gradient of the filtered sensor vector magnitude below which the algorithm is considered to have settled

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- flags: Control flags
 - update: Flag to control whether calibration coefficients are updated
 - matrix_update: Flag to control whether matrix coefficients are updated in addition to offset
- **SmartFusion Auto-calibration:** In addition to the basic auto-calibration parameters:
 - mu_offset: Q15 unsigned fixed-point convergence speed of offset parameters
 - mu_matrix: Q15 unsigned fixed-point convergence speed of matrix parameters
 - err_alpha: Q15 unsigned fixed-point coefficient controlling the filtering applied to the overall error in calculated cost function
 - err_thresh: Q15 unsigned fixed-point threshold applied to the calculated overall error below which the algorithm is considered to have converged
 - flags: Control flags
 - init_from_static: Flag to control whether calibration coefficients are initialized externally by a user or should be reset by initialization routine
- **Static Drift Compensation:** In addition to the static calibration parameters:
 - bias_thresh: Threshold indicating the maximum range of dynamic shift in dynamic bias magnitude that can be tracked
 - bias_alpha: Q15 unsigned fixed-point convergence speed of bias tracking
 - bias_range_limit: Q8 limit indicating maximum range of biases that can be tracked
 - noise_gate_thresh: Threshold indicating the magnitude of output data (after bias has been removed) below which the noise gate is applied
 - flags: Control flags
 - update: Flag to control whether the bias offset is updated
 - init_from_static: Flag to control whether calibration coefficients are initialized externally by a user or should be reset by initialization routine.

NOTE

This can be used to specify the static component of the bias, allowing a tighter range of `bias_thresh` to be specified so that slower movements can be tracked.

- **Sensor Calibration Wrapper:** If the wrapper instance structure is used, the mode field should be initialized to indicate which calibration routine should be used. The valid modes specified by the `cal_mode` enumeration are:
 - `CAL_NONE`: No calibration routine is applied
 - `CAL_STATIC`: The static calibration routine is applied
 - `CAL_BASIC_AUTOCAL`: The basic auto-calibration routine is applied
 - `CAL_SMARTFUSION_AUTOCAL`: The SmartFusion auto-calibration routine is applied
 - `CAL_STATIC_DRIFT_COMPENSATION`: The static drift compensation routine is applied

Once the calibration parameters have been initialized, it is necessary to call the appropriate routine's initialization function (if it has one) or the `cal_init()` function when the wrapper is being used.

NOTE

It is presumed that un-initialized parameters will be reset to zero. It is therefore advised to use the `memset()` function to reset the parameter structure before initialization.

When using the wrapper, the `controls` field is a union of a 16-bit word and the overlapped calibration routine flags bit field structure, allowing the flags to be set individually or all at once.

To generate optimal code for the ARM M0, pointers to the input/output data vectors refer to vector types where the elements are declared as 32-bit signed integer types. However, the range of these

elements should not exceed a signed 16-bit range ($-32768 \leq x|y|z \leq 32767$). The overall magnitude of the represented vector should also not exceed unsigned 16-bit range ($|v| \leq 32768$).

Similarly, the fixed-point matrix coefficients also use 32-bit types but should not exceed a signed 16-bit range. In cases where the matrix coefficients represent values greater than one, they should be scaled to a 16-bit range and the `q_format` parameter should also be adjusted accordingly. For example, to represent coefficients in the range of ± 2.0 , `q_format` should be set to 14 and the coefficients scaled by 214. Values of ± 2.0 (32768 once scaled) should saturate at 32767 to prevent overflow.

Depending on the algorithm used to calculate the calibration coefficients, the offset vector may need to be applied before or after applying the matrix. The static calibration routine supports both methods, but this should be indicated by setting the `offset_post_apply` appropriately. When set, the offset vector will be applied after applying the matrix.

When applying static calibration, it is necessary to initialize the calibration coefficients (`offset` and `matrix`) to be applied.

When applying SmartFusion auto calibration, in some cases it may be preferred to backup and restore the calculated calibration coefficients between instantiations rather than starting from scratch each time. In this case it is necessary to prevent the calibration routine's initialization function from resetting the coefficients by setting the `init_from_static` flag.

In order to work correctly, the auto calibration routines need to know the expected magnitude of the vector produced by the sensor once distortions have been removed. This reference magnitude is provided by initializing the `ref_mag` parameter. The value of this parameter should ideally be determined by performing an external measurement of the gravitational/geomagnetic field strength and expressing this in the same format and sensitivity as generated by the sensor. Without this, the value can be set according to estimated or published values. Alternatively, this value can be set to zero which enables a mode in which the calibration attempts to determine the sensor vector magnitude for itself.

As there is inevitably some margin of error in setting the reference magnitude and potentially quite a lot of variability in the uncalibrated sensor vector magnitude, the `mag_range` parameter has been provided to specify the tolerance for the reference magnitude. This should be set in order to encompass the full range of expected acceptable magnitude values. For example, when the expected magnitude is 1000 and the desired tolerance $\pm 10\%$ (a range from 900 to 1100), the `mag_range` should be set to 0.1 (3277 in Q15 fixed point). In cases where the actual magnitude lies outside this range, the auto calibration routine will never complete. Conversely, when the range is set too wide, the calibration routine will detect completion too early and result in imperfect calibration. In case of noisy sensor data, the magnitude range can also be used to filter out outliers to some extent.

The SmartFusion auto calibration routine includes functionality for detecting when it has achieved convergence and the sensor distortions have been sufficiently removed. This is useful for determining when calibration is complete or when it has subsequently become unsettled. As the level of attainable convergence is sensitive to the amount of noise that exists in the sensor data, tuning parameters have been provided to tune the performance. The `err_alpha` parameter tunes the amount of filtering applied to the calculated overall error and can be increased in situations where the amount of noise in the sensor data is higher. The `err_thresh` parameter sets the threshold at which convergence is detected and should be increased in situations where higher sensor noise has reduced the level of convergence that is attainable.

4.10.2.3 Processing

Sensor calibration processing is performed either by calling the appropriate process function on an instantiation of the related parameter structure or by calling function `cal_process()` when using the wrapper.

Prior to calling the process function, it is necessary to indicate to the algorithm whether the sensor data is valid and has been updated by setting the `in_data_valid` flag. This is done to prevent invalid data from getting into the calibration routine and corrupting its operation. Examples of invalid data

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

include null data while the sensor is starting up or saturated sensor data. Detection of these conditions are sensor specific, so this must be done prior to calling the calibration routine.

For SmartFusion auto-calibration, it is possible to speed up the rate of convergence by calling the process function multiple times between samples. In this case the `in_data_valid` flag should only be set when the sensor data is updated.

In some cases, it may be desirable to disable the application of the calibration coefficients at runtime or selectively only apply offset correction. The `apply` flag enables/disables the application of the calibration coefficients altogether. The `matrix_apply` flag enables/disables the application of just the matrix coefficients. It is not possible to only apply the matrix coefficients.

In some cases (for instance upon completion of one-shot mode), it may be desirable to disable the updating of the calibration coefficients by the auto calibration routine at runtime or selectively only update the offset coefficients. The `update` flag enables/disables the updating of the calibration coefficients altogether. The `matrix_update` flag enables/disables the update of just the matrix coefficients. It is not possible to only update the matrix coefficients.

Once the sensor calibration routine's processing cycle is complete, the calibrated sensor data can be read from the location referenced by the `out_data` pointer. The `out_data_valid` flag can also be read to determine whether the calibration routine detects the sensor data to be within its valid range of acceptable magnitudes (as specified by `ref_mag` and `mag_range`).

Since the state of the calibration routine can be determined, flags have been provided indicating when calibration is complete or has become unsettled. For basic auto-calibration, the `settled` flag can be used and for SmartFusion auto-calibration routine the `converged` flag can be used. In one-shot mode, calibration can be stopped when the appropriate flag is set.

4.11 Sensor Fusion Library

4.11.1 Overview

The Sensor Fusion Library (SFL) provides a set of modules for the processing and fusing of sensor data.

4.11.2 SmartFusion Integration Engine

The SmartFusion Integration Engine is provided for integrating and decimating sensor data from MEMS gyroscopes and accelerometers.

Sensor fusion applications often have restrictions on the maximum rates at which raw sensor can be sampled and utilized. These restrictions are often determined by the maximum rate at which the sensor data can be processed and/or the bandwidth available to transmit it. However, limiting the rate at which the inertial sensors are sampled in this way reduces the accuracy at which the motion can be tracked.

By numerically integrating the inertial sensor data, it is possible to sample it at much higher rates and decimate it to a lower rate without losing any of the critical motion information. This allows the data to be processed and transmitted at much lower rates without losing any accuracy.

4.11.2.1 Modes of Operation

The SmartFusion Integration Engine supports different modes of operation depending on what types of inertial sensor information are available and in what format they are required. These modes of operation are handled by a set of sensor data integrators as follows:

- δQ Integrator: Integrates 3D gyroscope data and outputs changes in orientation in body relative unit quaternion (w, x, y, z) form.
- $\delta \Theta$ Integrator: Integrates 3D gyroscope data and outputs changes in orientation in body relative Euler angle (ϕ, θ, ψ) form.

- δV Integrator: Integrates 3D accelerometer data and outputs changes in velocity in body relative form.

Each of these integrators operate independently and can be used all at once or in any combination. They can also operate at different sample rates and decimation factors.

4.11.2.2 API Usage

Allocation

An instance of the `smartfusion_integration_engine` structure must be instantiated, either statically or on the heap, and must be maintained for the life-cycle of sensor fusion processing.

Initialization

Before processing can be performed on an instance of the SmartFusion Integration Engine module, a subset of the parameters within the `smartfusion_integration_engine` structure must be initialized. These parameters are as follows:

- `controls`: Module control and status flags.
- `dv_integrator`: δV integrator parameters
- `dt_integrator`: $\delta \Theta$ integrator parameters
- `dq_integrator`: δQ integrator parameters

The `controls` field is an instance of the `smartfusion_integration_engine_flags` bit-field structure. This contains the control flags for the module. These include the `a_data_valid` and `g_data_valid` input flags for indicating when valid accelerometer or gyroscope input data is available as well as the `dv_data_valid`, `dt_data_valid`, and `dq_data_valid` flags for indicating when valid δV , $\delta \Theta$, and/or δQ output data is available. These should all be initialized to 0.

`dv_integrator` and `dt_integrator` are instances of the `smartfusion_vector_integrator` structure, while `dq_integrator` is an instance of the `smartfusion_quaternion_integrator`. These structures share common parameters which must be initialized. These parameters are as follows:

- `in_data`: Pointer to the appropriate type of input sensor data.
- `dec_factor`: Decimation factor.
- `scale_factor`: Scale factor applied to integrated sensor data.
- `scale_shift`: Scale shift applied to integrated sensor data (used in conjunction with `scale_factor`).

NOTE

All remaining parameters are either output or state parameters and need not be initialized.

The `in_data` pointer should reference raw input sensor data of a type which is appropriate for each integrator. For the δV integrator, this should reference accelerometer data. For the $\delta \Theta$ and δQ integrators, this should reference gyroscope data.

`dec_factor` is an integer value and should be set according to how much the rate of the raw sensor data should be decimated by. One output delta sample is output for every `dec_factor` samples. Setting `dec_factor` to 0 disables the integrator.

The `scale_factor` and `scale_shift` are used in conjunction with each other and are applied to the integrated sensor data to scale it to the desired output representation. As they have sensitivity scaling, sample rate scaling, unit conversion and dynamic range scaling all encoded within them, selecting appropriate values can be quite complex.

After initializing all the parameters indicated previously, it is necessary to call the `smartfusion_integration_engine_init()` function on the instantiated of the `smartfusion_integration_engine` structure.

Processing

Integration Engine module processing is performed by calling the `smartfusion_integration_engine_process()` function on an instance of the `smartfusion_integration_engine` structure. Prior to calling this function, the contents of the vectors referenced by the `in_data` pointers in each enabled integrator should be updated with the appropriate sensor data and the associated data valid flag should be set to 1.

If the sensors are sampled at different rates and data from individual sensors are not available at every processing cycle, the data valid flags for these inputs should be set to 1 when they have data available and 0 when not.

The integrators each have their own countdown timers which they use to determine when to calculate their output `delta` data. When these counters expire (reach zero), `delta` is calculated, the counter is reset to `dec_factor`, and the appropriate `delta` data ready flag in the `controls` field is set to 1.

These `delta` data ready flags can be polled after calling the `smartfusion_integration_engine_process()` function. However, as a convenience they are also returned by the function in the form of a bit-mask:

- Bit 0: `dv_data_valid`
- Bit 1: `dt_data_valid`
- Bit 2: `dq_data_valid`

When the appropriate flag is set after calling the process function, the corresponding `delta` data can be read and used. These remain valid until the next `delta` value is calculated, although the corresponding flags are cleared the next time `smartfusion_integration_engine_process()` is called.

δV and $\delta \Theta$ are represented in 16-bit fixed point form with a Q format and units determined by `scale_factor` and `scale_shift`. δQ is always represented in Q15 fixed point unit quaternion form.

4.11.3 SmartFusion Attitude and Heading Reference System

The SmartFusion Attitude and Heading Reference System (AHRS) is provided for fusing sensor data from MEMS gyroscopes, accelerometers and magnetometers to determine and track the absolute orientation of the device in which they are mounted relative to the Earth frame of reference.

The algorithm assumes that the sensor data supplied to it has been calibrated and that all distortions have been compensated for. When this is not the case, the performance will be compromised and drift artifacts may be observed.

4.11.3.1 Modes of Operation

The SmartFusion AHRS algorithm supports different modes of operation depending on what types of sensor information are available. The supported modes are as follows:

- Gyroscope, Accelerometer, and Magnetometer (GAM) Mode:
With information from all the sensors, the algorithm can track the absolute orientation of the device and compensate for any drift in the gyroscope data and noise in the accelerometer and magnetometer data.
- Gyroscope and Accelerometer (GA) Mode:
Using information from only the gyroscope and accelerometer, the algorithm can track the absolute orientation of the device and compensate for any drift in the pitch and roll components of the gyroscope data as well as noise in the accelerometer data. The reference heading is taken to be whatever the heading is at initialization, but this can drift over time.
- Gyroscope Only (G) Mode:

Using information from only the gyroscope, the algorithm can track changes in the orientation of the device but cannot compensate for any drift in the gyroscope data. The reference orientation is taken to be whatever the orientation is at initialization, but this can drift over time.

- Accelerometer and Magnetometer (AM) Mode:
Using information from only the accelerometer and magnetometer, the algorithm is able to track the absolute orientation of the device but is less able to compensate for noise in the accelerometer and magnetometer data.

NOTE

Gyroscope and Magnetometer (GM) Mode is not supported.

4.11.3.2 API Usage**Allocation**

An instance of the `smartfusion_ahrs` structure must be instantiated, either statically or on the heap, and must be maintained for the life-cycle of sensor fusion processing.

Initialization

Before processing can be performed on an instance of the SmartFusion AHRS module, a subset of the parameters within the `smartfusion_ahrs` structure must be initialized. These parameters are as follows:

- `controls`: Module control flags.
- `dq_data`: Pointer to the quaternion step rotation (δQ) input data.
- `a_data`: Pointer to the accelerometer input data.
- `m_data`: Pointer to the magnetometer input data.
- `beta_a`: Scaling factor controlling the relative weight of accelerometer data.
- `beta_m`: Scaling factor controlling the relative weight of magnetometer data.
- `q`: Output quaternion representing the orientation of the device in Earth frame of reference.

The `controls` field is a union of an 8-bit word and an instance of the `smartfusion_ahrs_flags` bit-field structure. This contains the control flags for the module, allowing them to be set and read individually using the `flags` field or all at once using the `word` field. These flags should be initialized to indicate which sensors are available by setting the appropriate `dq_data_valid`, `a_data_valid`, and `m_data_valid` flags to 1 or 0 to indicate the mode of operation.

Rather than use raw gyroscope data directly, the AHRS module receives gyroscope data in unit quaternion form, which represents the step rotation undergone by the device over the given sample period (δQ) in body relative terms. The elements (w, x, y and z) of `dq_data` are represented using Q15 signed fixed point values in the range of -1.0 to 1.0 (-32768 to 32767). Although this step rotation may be calculated directly from the angular velocity reported by the gyroscope, it is recommended to use the output of the δQ integrator in the Integration Engine.

Accelerometer and magnetometer data should be supplied to the AHRS module in raw form. The representations and magnitudes of these vectors are not important as they have no impact on the module.

Although the data structures for the δQ quaternion and the accelerometer/magnetometer vectors use underlying 32-bit types (for optimization reasons), they actually represent 16-bit quantities. Therefore, the magnitudes of these quaternions and vectors should never exceed signed 16-bit signed integer range (that is, $-32768 \leq ||v|| \leq 32767$).

If data for a particular sensor is unavailable (for example, when using the module in GA, G, and AM modes), the associated pointer should be set to `NULL`.

`beta_a` and `beta_m` are both Q15 unsigned fixed point parameters representing positive scaling factors in the range of 0 to 1.0 (32768). Suitable values for `beta_a` and `beta_m` should be selected to match the requirements of the use-case in terms of the maximum rate of rotation and tolerance to noise. Increasing these values will allow faster rates of rotation to be tracked at the expense of tolerance to accelerometer/magnetometer noise.

Although `q` is a unit quaternion representing the orientation of the device output by the module, subsequent orientations are dependent on previous ones, so it is necessary to initialize this with a starting orientation. If the device orientation is known at the time of initialization, this can be used, otherwise it is recommended to use the reference orientation [1.0, 0.0, 0.0, 0.0] (that is, upright and facing north). The elements (w, x, y and z) of `q` are represented using Q15 signed fixed point values in the range of -1.0 to 1.0 (-32768 to 32767).

Processing

AHRS module processing is performed by calling the `smartfusion_ahrs_process()` function on an instance of the `smartfusion_ahrs` structure. Prior to calling this function, the contents of the vectors referenced by `dq_data`, `a_data`, and `m_data` should be updated with the appropriate sensor data.

If the sensors are sampled at different rates and data from individual sensors are not available at every processing cycle, the flags for these inputs should be set to 1 when they have data available and 0 when not. They can also be used to indicate to the module when the sensor data is invalid for some reason and should therefore be ignored. Examples of invalid data include when the sensor is in an initialization or error state, saturated, or the magnitude is too low.

The algorithm uses a right-handed coordinate system, where the x-axis is aligned with north, the y-axis is aligned with east, and the z-axis is aligned with down. The gyroscope, accelerometer, and magnetometer data must be converted to this coordinate space for the algorithm to function correctly.

NOTE

Positive gyroscope values represent a clockwise rotation around the associated axis (when looking in the direction it is pointing).

The algorithm also assumes that the sensors have been sampled synchronously at a regular interval and that all distortions (for example, magnetometer hard/soft iron distortions) have been properly compensated for. If not, the algorithm performance will be degraded.

Once processing is complete, an updated orientation estimate can be read from `q`. This parameter represents the current Earth frame of reference orientation of the device in Q15 signed fixed point unit quaternion form.

5 Smart Tag Reference Application

5.1 Introduction

Smart Tag, Dialog's Bluetooth® low energy proximity tag reference application, provides an ideal starting point to develop a proximity tag application with the shortest time-to-market and lowest development cost and effort. The design comes with a complete software solution for the full proximity application and profile source codes. Dialog also provides fully-featured Android and iOS applications to manage the proximity tag's settings, trigger alerts, check the battery status, and play a fun 'Seek & Find' game, all in source code.

The Smart Tag reference application functions in the role of a Bluetooth Low Energy Proximity Reporter defined in the Proximity Profile listed in the Bluetooth specification [15]. The Proximity profile defines the behavior of a Bluetooth device when it moves away from a peer device, and it covers the use case where a connection loss causes an immediate alert. This alert notifies the user that the devices have become separated.

The Smart Tag application is designed to run on Dialog's DA14585 MSK HW reference design.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

As a starting point, developers are suggested to get familiar with the DA14585 datasheet [5], the software developer's guide of the DA14585 SDK [1], and the software platform reference of the DA14585 SDK [2].

5.2 Software Features

5.2.1 Profiles and Services

Besides the Proximity Reporter profile, the Smart Tag application implements the following profiles and services for monitoring and supporting additional features:

- Proximity profile, Reporter role
 - Link Loss service
 - Immediate Alert service
 - Tx Power service
- Battery Service, Server role
- Data Information service, Server role
- Find Me profile, Locator role
- SUOTA, Server role

5.2.2 Alerts

The Smart Tag application supports two types of alerts for user notifications, high level and mild level, as described in Table 64.

Table 64: Alert Types

Alert Type	Description
High level	<ul style="list-style-type: none"> ● LED blinking with a buzzer tone with the following pattern: <ul style="list-style-type: none"> ○ LED: 150 ms on, 150 ms off. ○ Buzzer: 150 ms on, 150 ms off, alternating between 392 Hz ("G" note) and 440 Hz ("A" note) ● Triggered when peer device writes immediate alert with 'High Alert', or Smart Tag disconnects from peer and Link Loss is set to 'High Alert'.
Mild level	<ul style="list-style-type: none"> ● LED blinking with a buzzer tone with the following pattern: <ul style="list-style-type: none"> ○ LED: 500 ms on, 500 ms off ○ Buzzer: 500 ms on, 500 ms off, 440 Hz ("A" note) ● Triggered when peer device writes immediate alert with 'Mild Alert', or Smart Tag disconnects from peer and Link Loss is set to 'Mild Alert'.

5.2.3 Advertising and Sleep Phases

Smart Tag advertises in undirected mode with different intervals for specific advertising phases:

- Advertising phase (200 ms interval):
It is the first minute after start-up or disconnection.
- Advertising phase (1000 ms interval):
It is the period of three minutes after the 200 ms interval phase.
- Deep Sleep phase:
After four minutes, the Smart Tag stops advertising and enters continuous Deep Sleep mode.

In Advertising mode, the Smart Tag blinks the green LED with a pattern of 50 ms on and 1000 ms off. Once devices are connected, the LED stops blinking; when the devices are disconnected, the LED starts blinking again since Smart Tag goes again in advertising mode.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

The Smart Tag application supports Extended Sleep mode during Connectable and Connected states, and Deep Sleep mode during the Deep Sleep Phase. The supported sleep modes of the DA14585 are explained in *section 8.6* in [1].

5.2.4 Push-Button Interface

The actions that are triggered on a push-button event depend on the operating state of the Smart Tag, as described in [Table 65](#).

Table 65: Push-Button Interface

Operating State	Action on Button Press
Alert is active	Stop alert.
Deep Sleep phase	Wake up Smart Tag and start advertising.
Connected and 'find me' locator discovered, immediate alert service on peer device	If the alert is not active, writes alert characteristic of immediate alert service on peer device. Stops the alert in peer device if alert is active.
Advertising phase	Long press (currently set to 3 s): bonding data are deleted from SPI Flash memory. When long press is detected, an 880 Hz tone ('A' note, 5th octave) will be played for 125 ms (Note 1)
All other states	None

Note 1 When deleting the bonding data from the Smart Tag SPI Flash and the Android device is paired with the Smart Tag device, the Smart Tag device needs to be removed from the list of paired devices of the Android device (usually via menu **Settings > Bluetooth > Forget Device**).

5.2.5 Security

According to the Bluetooth Core specification, the purpose of bonding is to create a relation between two Bluetooth devices based on a common link key (a bond). The link key is created and exchanged (pairing) during the bonding procedure and is expected to be stored by both Bluetooth devices to be used for future authentication.

Since the DA14585 IoT MSK reference design does not have a keyboard or display, it only supports the 'Just Works' pairing method. Upon completion of a successful bonding procedure, the Smart Tag application stores the security information (LTK, EDIV, and RAND which will be also referred as 'bonding data') in the SPI Flash memory for reuse on subsequent reconnections of the bonded device. For more information regarding the bonding procedure refer to [3], *section 5.6*.

When the Smart Tag is in Advertising mode, users can 'forget' a bonded central device by keeping the button pressed until a tone is heard. This indicates that the security information has been deleted from the SPI Flash memory and a new central device can now pair with the Smart Tag device.

NOTE

The Smart Tag transmits a Security Request command to the central device in order to trigger the pairing procedure upon receiving the connection request from the central device. However, this command could be ignored by the central device, when it does not wish to start a pairing procedure.

5.2.6 Battery Level

In Connected state, the Smart Tag software samples the battery level and updates the value of the battery level characteristic. The notification capability of the characteristic is disabled at the beginning of each connection. To enable value update notifications, the peer device must write the configuration attribute of the characteristic with the corresponding value.

5.3 Software Architecture

Group	File Name	Description
user_platform	i2c_gpio_extender.c	User drivers for the I2C GPIO extender
user_app	user_smarttag_proj.c user_smarttag_utils.c	Application code
Utilities	user_iot_dk_utils.c battery.c	User drivers for battery and MSK HW peripherals

5.4 Operation Overview and State Machines

This section provides information about important functions of the application and a detailed description of the used Finite State Machines (FSM).

5.4.1 Application Configuration Parameters

The main parameters of the Smart Tag application software, which can be adjusted to customize certain functionality of the application, are listed in [Table 66](#).

Table 66: Smart Tag Application Configurable Parameters

Parameter	Description	Current Value
APP_SPI_POWEROFF_DELAY	Upon application initialization, the timer APP_FLASH_POWEROFF_TIMER is set with this value to delay the SPI Flash power down mode and allow developers to use the SmartSnippets Flash Programmer application to connect to the Smart Tag device and re-program the SPI Flash device.	1 s
APP_SLEEP_DELAY	Upon application initialization, the function app_set_startup_sleep_delay(APP_SLEEP_DELAY) is called to modify the system startup sleep delay. This delay allows the developer to have an active JTAG interface and to use the debugger to connect to the Smart Tag device.	5 s
APP_FIRST_ADV_PHASE_DUR	This parameter sets the APP_ADV_TIMER to control the advertising intervals. Refer to section 5.2.3 for details.	60 s
APP_FIST_ADV_PHASE_INTVAL	This parameter sets the advertising interval for the first advertising phase.	200 ms
APP_SECOND_ADV_PHASE_DUR	This parameter sets the APP_ADV_TIMER to control the advertising intervals. Refer to section 5.2.3 for details.	3 min
APP_SECOND_ADV_PHASE_INTVAL	This parameter sets the advertising interval for the second advertising phase.	1 s
APP_BOND_DB_DATA_OFFSET	The SPI Flash start address where the bonding data are stored.	0x32000
APP_ADV_BLINK_ON_DUR	Controls the LED 'on' duration during advertising.	50 ms
APP_ADV_BLINK_OFF_DUR	Controls the LED 'off' duration during advertising.	1 s

5.4.2 Application Task State Machine

The FSM of the application task of Smart Tag consists of the following states ([Table 67](#)):

Table 67: Application Task: FSM States

State	Description
APP_DISABLED	Application task initiated. Waiting for GAPM_DEVICE_READY_IND message.

State	Description
APP_DB_INIT	Database initialization in progress.
APP_CONNECTABLE	Advertising or Continuous Extended Sleep.
APP_CONNECTED	Device connected to Proximity Monitor.

Figure 15 graphically illustrates the FSM. The state transitions are described in Table 68.

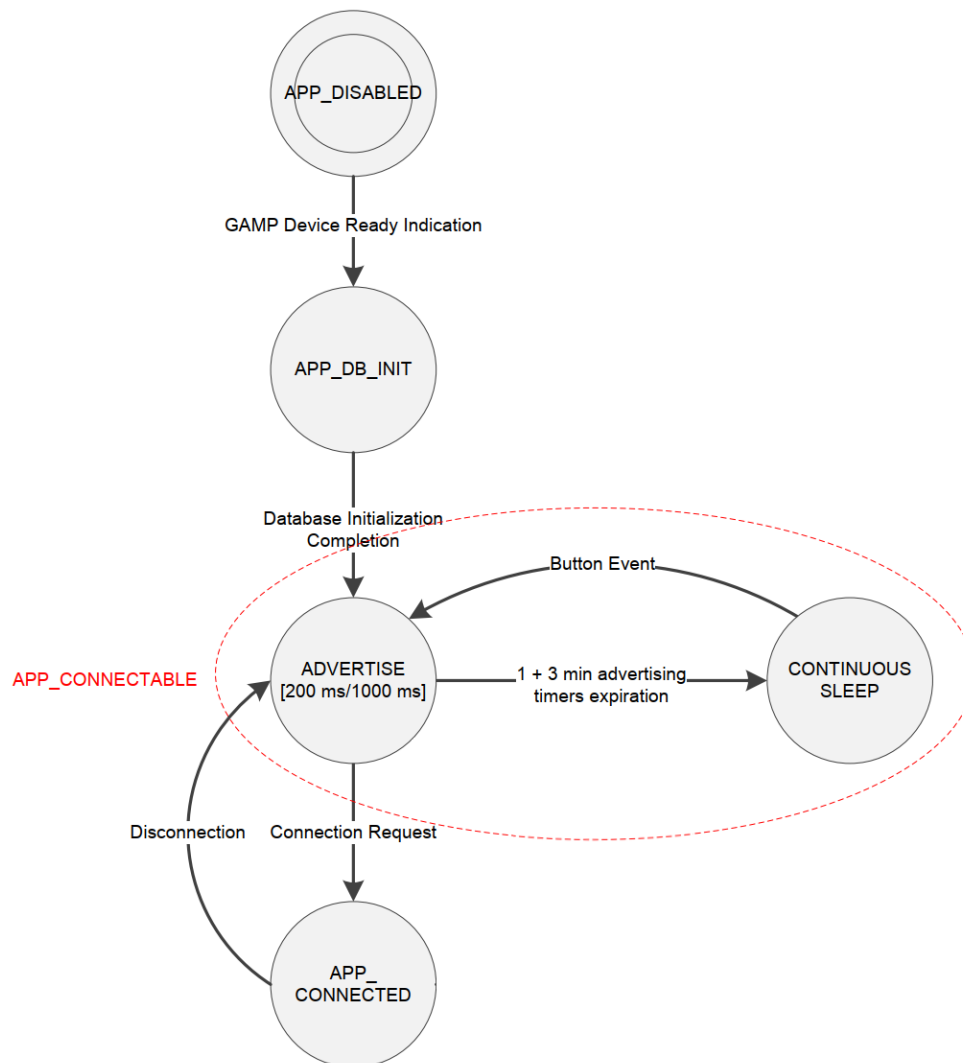


Figure 15: Smart Tag Application Task FSM

Table 68: State Transitions of the Application Task FSM

State Transition		Event
From	To	Action
APP_DISABLED		GAPM_DEVICE_READY_IND message reception.
	APP_DB_INIT	Start database initialization of supported profiles.
APP_DB_INIT		Database initialization procedure is completed.
	ADVERTISE (APP_CONNECTABLE)	Start advertising in 200 ms advertising interval.

State Transition		Event
From	To	Action
ADVERTISE (APP_CONNECTABLE)		Timer for 1000 ms advertising interval expired.
	CONTINUOUS SLEEP (APP_CONNECTABLE)	Device stops advertising and/or active alerts and switches to Deep Sleep mode.
CONTINUOUS SLEEP (APP_CONNECTABLE)		Button press event.
	ADVERTISE (APP_CONNECTABLE)	Device exits Deep Sleep mode and restarts advertising in 200 ms interval.
ADVERTISE (APP_CONNECTABLE)		Connection Request has been received.
	APP_CONNECTED	Enable profiles and start battery polling.
APP_CONNECTED		Device disconnects.
	ADVERTISE (APP_CONNECTABLE)	Start advertising in 200 ms advertising interval.

5.4.3 Callback Functions

A set of callback functions is defined in `user_callback_config.h` which consist of the entry points of the application.

- `user_app_on_init()`:
 - It is the main entry point of the application task.
 - It is used to initialize the parameters of high-level profiles and low-level hardware modules.
- `user_app_on_set_dev_config_complete()`:
 - It is called after the device configuration is complete.
 - This means that the device database of the supported profiles has been created and the device can enter Advertising mode.
 - Also, in this function the Flash power-off timer is initialized. This timer allows users to use the [SmartSnippets](#) tool to program the Flash memory by leaving the Flash memory in power-up mode for $(APP_SPI_POWEROFF_DELAY * 10)$ ms. The default time is 1 s.

5.4.4 Advertising

- `user_advertise_operation()`:
 - It starts or restarts the Advertising mode.
 - It is called upon completion of the database initialization, upon device disconnection, and upon expiration of the advertising timer to start the advertising interval of the second phase.
 - The function also initializes the advertising timer for the variable advertising interval feature and the blink timer for Advertising mode LED blinking.
 - Finally, function `user_undirected_advertise_start()` constructs and sends the command `GAPM_START_ADVERTISE_CMD` to the `GAPM_TASK` in order to initiate the advertising mode.
- `user_adv_timer_handler()`:
 - It is called upon expiration of the advertising timer.
 - Its main functionality is to program the correct advertising phase.
 - For Continuous Sleep state, the function disables all timers and alerts before setting the device in Deep Sleep mode.

- `user_blink_timer_handler()`
 - It handles the expiration of the LED blinking timer, restarts the timer, and inverts the state of the LED.

5.4.5 Connection

- `user_app_on_connection()`:
 - It is called upon reception of a connection request from a central role device.
 - When this function is called, the Smart Tag application stops the LED blinking timer, enables the profiles and services, selects the proper sleep mode, and enables the device profiles.
- `user_on_disconnect()`:
 - It is called upon reception of a `GAPC_DISCONNECT_IND` message, which indicates that the connection does not exist anymore.
 - When this function is called, the battery level polling is stopped, and the advertising procedure starts again.

5.4.6 Security

- `user_app_on_pairing_request()`:
 - It is called during the pairing process.
 - It informs the peer device about the security capabilities of the device. The Smart Tag application uses 'Just Works' mode with bonding capability.
 - It also checks whether the Smart Tag device is already paired with another device by looking for bonding data stored in SPI Flash memory. When the Smart Tag device is already paired with another central device, it will not accept the new pairing request. Smart Tag only supports bonding with one central device at a time.
- `default_app_on_ltk_exch()`:
 - It is the SDK 5 default function and is called upon reception of a `GAPC_BOND_REQ_IND` message with request set to `GAPC_LTK_EXCH` (Long Term Key Exchange).
 - This function generates the LTK and sends it to the host.
- `user_app_on_pairing_succeed()`:
 - It is called upon reception of a `GAPC_BOND_IND` message with status `GAPC_PAIRING_SUCCEED`.
 - The function stores the security information into SPI Flash memory and completes the connection establishment phase.
- `user_app_on_encrypt_req_ind()`:
 - It is called to initiate a secure connection upon the reception of a `GAPC_ENCRYPT_REQ_IND` message.
 - In order to validate the connecting host, this function uses the parameters `RAND` and `EDIV` to check whether the Smart Tag has already stored the bonding data in SPI Flash memory. If not, the request is rejected, and the peer is disconnected.
- `user_app_on_encrypt_ind()`
 - It is called upon the reception of a `GAPC_ENCRYPT_IND` message to indicate that encryption is completed.
 - The database is updated with values from the SPI Flash memory.

5.4.7 Push button

In the application initialization function `user_app_on_init()`, a wakeup interrupt (IRQ) is enabled on the GPIO that is allocated to the push-button interface. This is done via the API functions `wkupct_register_callback()` and `wkupct_enable_irq()` of the wakeup module driver. The callback

function `user_button_press_cb()` is registered to enable a wakeup interrupt when the button is pressed.

- `user_button_press_cb()`:
 - It is the callback function of the application, called from the `WKUP_QUADEC_IRQn()` interrupt handler of the wakeup module driver when the button is pressed.
 - The function checks the state of the application and triggers the required action, as described in Table 65.
 - It also calls the API functions `wkupct_register_callback()` and `wkupct_enable_irq()` of the wakeup module driver to register the `user_button_release_cb()` callback function and enable a wakeup interrupt when the button is released.
 - Finally, the function sends a wakeup message to the `TASK_APP` to start the button press timer, which is used to detect a long key press for deleting the bonding data stored in the SPI Flash memory.
- `user_button_release_cb()`:
 - It is the callback function of the application, called from the `WKUP_QUADEC_IRQn()` interrupt handler of the wakeup module driver when the button is released.
 - The function calls the API functions `wkupct_register_callback()` and `wkupct_enable_irq()` of the wakeup module driver to register the `app_button_press_cb()` callback function and enable a wakeup interrupt when the button is pressed.
 - Finally, the function sends a wakeup message to the `TASK_APP` to stop the button press timer.
- `user_wakeup_handler()`:
 - It is called upon reception of the wakeup message and calls function `user_advertise_operation()` to start advertising.
 - It also starts/stops the button press timer depending on the button status (`user_button_status`).

5.4.8 Proximity Reporter and Alerts

- `app_proxr_enable()`:
 - It enables the Proximity Reporter profile upon connection.
- `user_proxr_alert_ind_handler()`:
 - It is the message handler of a `PROXR_LLS_ALERT_IND` message, which is sent by the Proximity Reporter profile to trigger an alert on the device.
 - This function calls functions `user_proxr_alert_start()` or `user_proxr_alert_stop()` to start or stop an alert, depending on the alert level received in `PROXR_ALERT_IND`.
- `user_proxr_alert_start()`:
 - It initiates user alert indications.
 - It updates the alert state parameters and depending on the alert level it starts the PWM engine by calling the function `user_proxr_pwm_enable()` to generate the alert melody.

NOTE

The LED functionality is controlled from within the functions that program the PWM tones, as explained in section 5.4.9.

- `user_proxr_alert_stop()`:
 - It stops user alert indications.
 - It clears the alert state parameters, turns off the alert LED, and stops the `APP_PXP_TIMER`.

5.4.9 PWM Engine

This section describes how the Smart Tag uses the PWM0 and PWM1 (TIMER 0) outputs to create the alert melodies. Details on the PWM0 and PWM1 can be found in *section 7.12* in [1].

- `user_proxr_pwm_enable()`: it initializes TIMER 0:
 - It enables the TIMER 0 peripheral clock by calling the `set_tmr_enable()` PWM API driver function.
 - It calls `set_tmr_div()` to sets the TIMER 0 clock division factor to 8 (16 MHz clock source).
 - It calls `timer0_init()` to initialize the PWM with the desired PWM mode, TIMER 0 'on' time division option, and clock source selection.
 - In this example, the timer tick period is configured to:

$$(1/16 \text{ MHz}) \times 8 (\text{clock division}) \times 10 (\text{TIM0_CLK_DIV_BY}_10) = 5 \mu\text{s} \quad (1)$$
 - It sets the TIMER 0 'on', 'high', and 'low' times by calling function `timer0_set()`.
 - It registers a callback function for `SWTIM_IRQn` interrupts by calling `timer0_register_callback()`. The callback function pointer is an input parameter to this function. In the Smart Tag application three different melodies/tones are needed, which are handled by the following callback functions:
 - `high_alert_pwm_callback()`: programs the high alert melody.
 - `mild_alert_pwm_callback()`: programs the mild alert melody.
 - `button_pwm_callback()`: programs the button long press tone.
 - It enables the `SWTIM_IRQn` by calling the `timer0_enable_irq()` function.
 - It starts TIMER 0 by calling the `timer0_start()` function.
- `high_alert_pwm_callback()`:

It is a callback function for the `SWTIM_IRQn` interrupt that handles the high alert melody. The following parameters configure the alert melody:

 - The melody is defined in the constant array `alert_high_notes[]`.
 - In this array developers can define a new sequence of notes and pauses. Note values represent the frequency of the musical notes. For example, '880' means 880 Hz which is the 'A' note of the 5th octave.
 - Each time this callback function is called, a note or a pause from the array will be programmed to the PWM engine by calling the functions `timer0_set_pwm_high_counter()` and `timer0_set_pwm_low_counter()`.
 - The duration of the note/pause is determined by the parameter value `ALERT_HIGH_DURATION` that is passed to the function `timer0_set_pwm_on_counter()`. At the end of this duration, an interrupt will be triggered, and the callback function will be executed again to program the next note/pause from the array.
 - For the LED to blink synchronously with the melody, the LED is controlled within `high_alert_pwm_callback()`. When a pause is programmed, the LED is turned off. When a note is played, the LED is turned on.
- `mild_alert_pwm_callback()`:

It is a callback function for the `SWTIM_IRQn` interrupt that handles the mild alert melody. This function is similar to the `high_alert_pwm_callback()` function, described in the previous section, with one extra parameter:

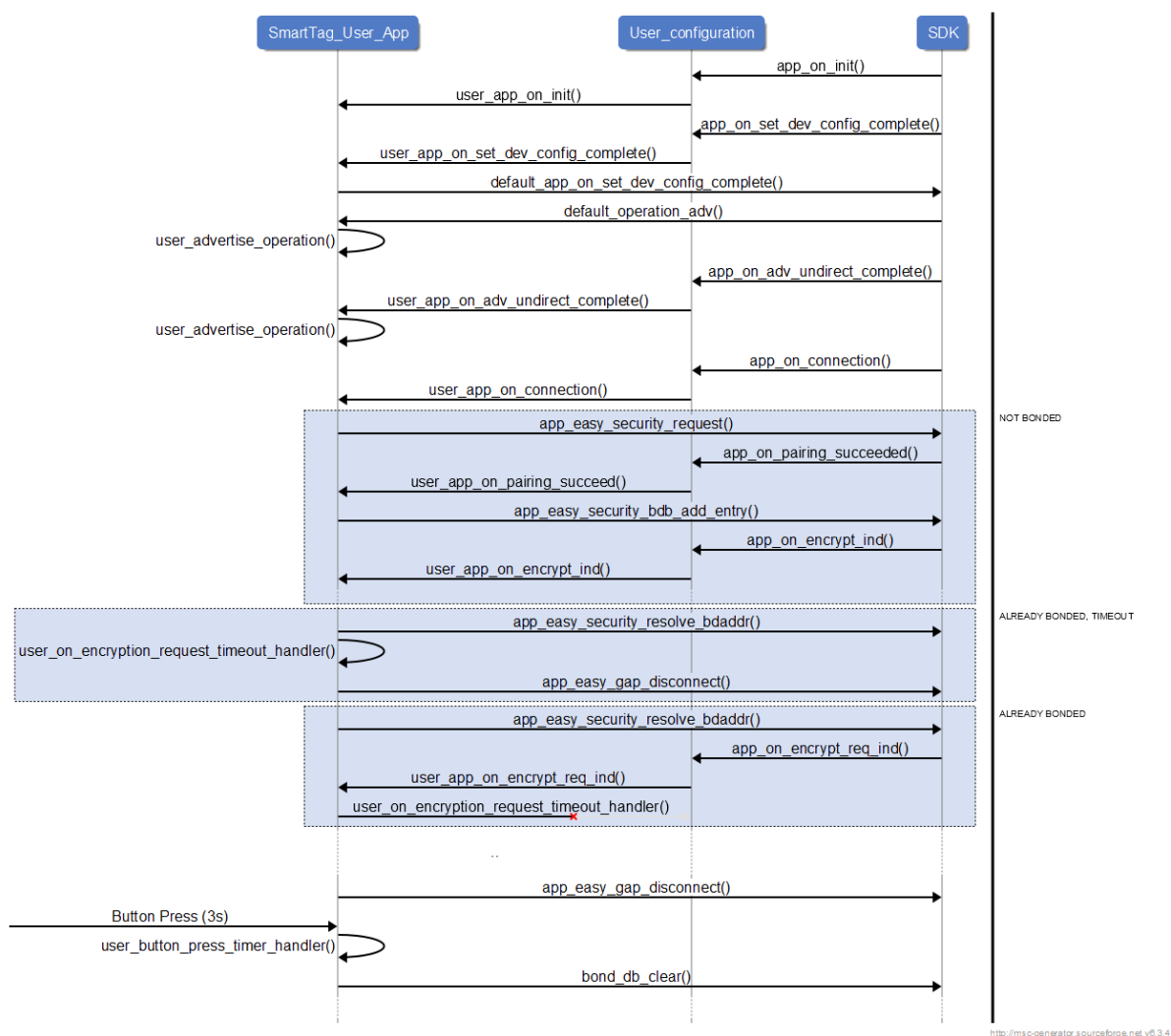
 - `ALERT_MILD_EXTRA_DELAY`: This parameter determines how many times the function will be called without programming a note/pause. This is needed in case that a note/pause duration needs to be quite long and cannot be set by the `timer0_set_pwm_on_counter()` range.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- The notes/pauses for the mild alert melody are defined in the constant array `alert_mild_notes[]`. The duration of the note/pause is determined by the parameter value `ALERT_MILD_DURATION`.
- `button_pwm_callback()`:
It is a callback function for the `SWTIM_IRQn` interrupt that handles the long button press alert. This alert is a single tone alert (`button_press_notes = PWM_TONE_A_5TH`) and the TIMER 0 'on' counter is not programmed again, because no further interrupts are needed to program more notes/pauses.

5.4.10 SmartTag Sequence Diagram



6 Beacon Reference Applications

6.1 Introduction

This section describes the Bluetooth® Low Energy Beacon reference application design based on DA14585. It serves as a developer's guide to customize the beacon for any desired purposes.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

Dialog Semiconductor is a member of the iBeacon™ program. Please contact your local sales office to learn more about the possibilities we can offer relating to this standard. The Dialog Beacon reference application also supports the AltBeacon protocol as well as the Eddystone™ protocol, with supported modes being the Eddystone-UID, the Eddystone-URL, and the Eddystone-TLM (unencrypted).



Figure 17: Beacon Protocol Logos

The Beacon reference software can be downloaded from the [Dialog support portal](#) and run on the DA14585 IoT MSK reference hardware. It also runs on Dialog's DA14585/14586 Development kit (Expert/Pro/Basic).

6.2 What is a Beacon?

Beacons are battery powered devices that advertise a particular Bluetooth low energy payload with identifying information. In short, it is a device that just says [Figure 18](#).



Figure 18: Bluetooth Low Energy Beacon

Although [\[17\]](#) contains a startup guide that explains how to run the Beacon reference software out of the box and how to configure the main parameters, this section provides to software developers all design details to customize the Dialog Beacon reference application for more advanced use cases, such as:

- Adaptive modification of advertising data
- Choosing from various beacon formats
- Interleaving connectable advertising events
- Software Updates Over The Air (SUOTA)

6.3 Beacon Example

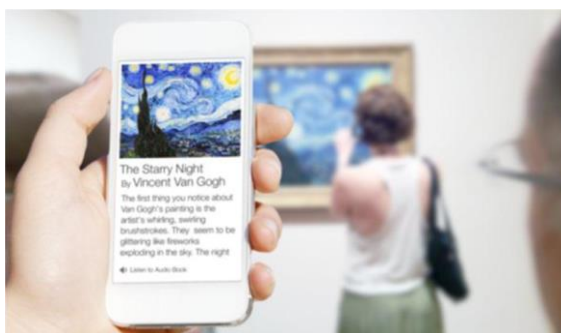
[Table 69](#) shows a beacon configuration that is chosen for an international museum application, where each exhibit has its own unique beacon. The specific advertising data transmitted by each beacon is picked up by mobile phones near the exhibit. The mobile phone will then direct users to additional information regarding the exhibit.

In this museum example, the organization uses a Universal Unique Identifier (UUID). The museum location is indicated by the Major Field and the exhibit number within the museum by the Minor field.

Table 69: Example of Advertising Data from a Museum Beacon

Museum Location		London	Paris	Amsterdam
UUID		1234A567-3854-ABED-8FAC-56E783159AE2		
Major		10	20	30
Minor	Exhibit #1	1	1	1
	Exhibit #2	2	2	2
	Exhibit #3	3	3	3

When the beacon sends '20' as a major value and '1' as a minor value, the application on smartphones/tablets will guide visitors to additional information on exhibit #1 from the Paris museum. This additional information might come from the smartphone application or from the Internet. As a result, visitors will receive only the specific information that is of interest when standing close to the exhibit ([Figure 19](#)).

**Figure 19: Description of the Exhibit on a Smartphone**

The smartphone application can also provide a distance indication to the beacon using the RSSI value.

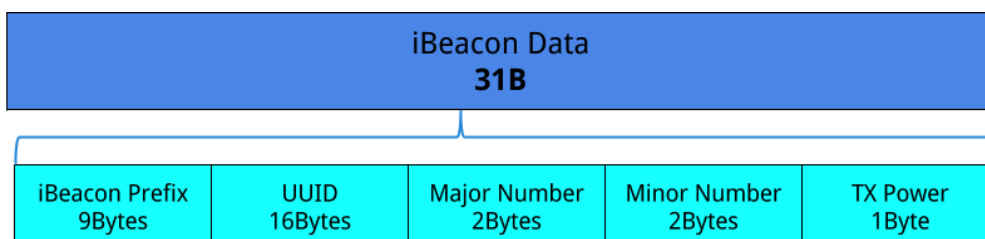
There are endless other applications using Beacon apart from the museum application, for example, in retail, advertising, and sports events.

6.4 Beacon Formats

There are various beacon formats. The beacon formats supported by DA14585 IoT MSK are presented in the following subsections.

6.4.1 iBeacon

iBeacon™ is a protocol developed by Apple but used by various vendors. It is a closed format that exposes a UUID and other configurable values. An iBeacon frame is presented in [Figure 20](#).

**Figure 20: iBeacon Frame**

An iBeacon frame consists of:

- **iBeacon Prefix:** A 9-byte constant preamble identifying iBeacon
- **UUID:** A 16-byte string used to differentiate a large group of related beacons.
- **Major:** A 2-byte string meant to distinguish a smaller subset of beacons within the larger group.
- **Minor:** A 2-byte string meant to identify individual beacons.
- **Tx Power:** A 1-byte value representing the RSSI at 1 m from the advertiser.

6.4.2 AltBeacon

AltBeacon is an open beacon format for proximity beacons. The emitted message contains information that the receiving device can use to identify the beacon and to compute its relative distance to the beacon. The receiving device may use this information as a contextual trigger to execute procedures and implement behaviors that are relevant to being in proximity to the transmitting beacon (see [13]). Figure 21 presents the AltBeacon frame structure.



Figure 21: AltBeacon Frame

Table 70 provides more detailed information on the various field of an AltBeacon frame.

Table 70: AltBeacon Protocol Fields

Field Name	Description	Accepted Values
AD LENGTH [MFG SPECIFIC]	Length of the type and data portion of the Manufacturer Specific advertising data structure.	0x1B
AD TYPE [MFG SPECIFIC]	Type representing the Manufacturer Specific advertising data structure.	0xFF
MFG ID	The beacon device manufacturer's company identifier code.	The little-endian representation of the beacon device manufacturer's company code as maintained by the Bluetooth SIG assigned numbers database.
BEACON CODE	The AltBeacon advertisement code.	The big-endian representation of the value 0xBEAC.
BEACON ID	A 20-byte value uniquely identifying the beacon.	The big-endian representation of the beacon identifier. For interoperability purposes, the first 16 bytes of the beacon identifier should be unique to the advertiser's organizational unit. Any remaining bytes of the beacon identifier may be subdivided as needed for the use case (Note 1).
REFERENCE RSSI	A 1-byte value representing the average received signal strength at 1m from the advertiser.	A signed 1-byte value from 0 to -127.
MFG RESERVED	Reserved for use by the manufacturer to implement special features.	A 1-byte value from 0x00 to 0xFF. Interpretation of this value is to be defined by the manufacturer and is to be evaluated based on the MFG ID value.

Note 1 In Dialog's beacon reference design, the Beacon ID is divided into a 16-byte UUID, a 2-byte ALT_val1, and a 2-byte ALT_val2.

6.4.3 Eddystone

Eddystone™ is a protocol specification that defines a BLE message format for proximity beacon messages. It describes several different frame types that may be used individually or in combination to create beacons suitable for a variety of applications (see [9]). Figure 22 shows the Eddystone modes supported by the Dialog Beacon reference design.

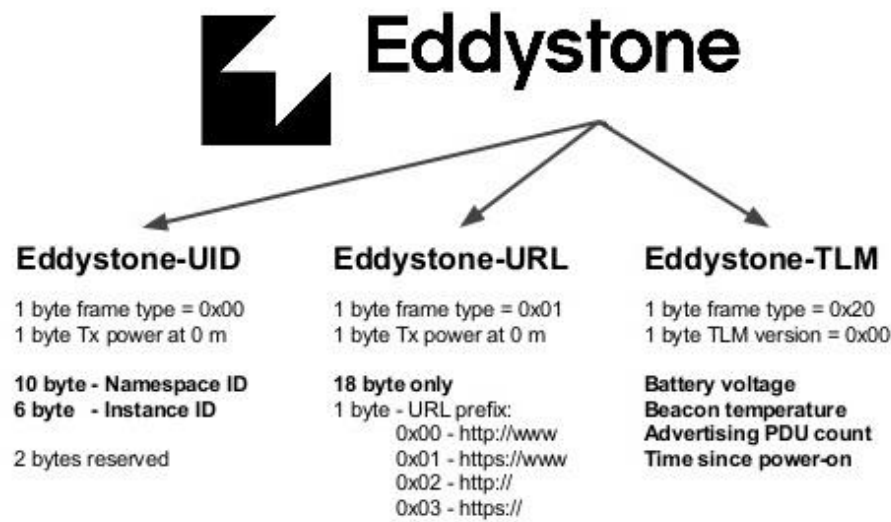


Figure 22: Eddystone Modes Supported by Dialog's Beacon Reference Design

Figure 23 describes the various fields of the different Eddystone Beacon modes.

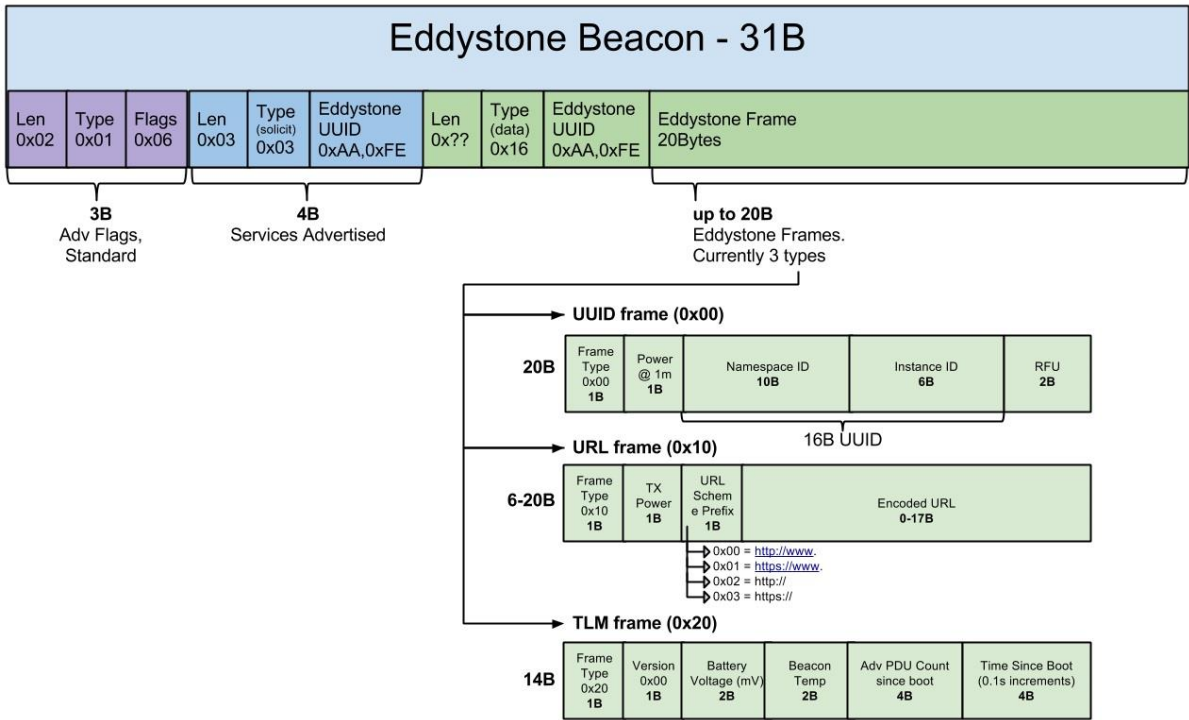


Figure 23: Eddystone Different Mode Frames Analyzed

The specific type of Eddystone frame is encoded in the high-order four bits of the first octet in the Service Data associated with the Service UUID. Permissible values are shown in Table 71.

Table 71: Eddystone Frame Types

Frame Type	High Order 4 bits	Byte Value
UID	0000	0x00
URL	0001	0x10
TLM	0010	0x20
EID	0011	0x30
RESERVED	0100	0x40

Note 1 The four low-order bits are reserved for future use and shall be 0000.

Note 2 Although the core Bluetooth data types are defined in the standard as little-endian, Eddystone's multi-value bytes defined in the Service Data are all big-endian.

6.4.3.1 Eddystone-UID

The Eddystone-UID frame broadcasts an opaque and unique 16-byte Beacon ID composed of a 10-byte namespace and a 6-byte instance. The Beacon ID is useful in mapping a device to a record in external storage. The namespace of the ID can be used to group a particular set of beacons, while the instance of the ID identifies individual devices in the group. The division of the ID into a namespace and an instance can also be used to optimize BLE scanning strategies, for example, by filtering only the namespace.

The UID frame is encoded in the advertisement as a Service Data block associated with the Eddystone service UUID. The frame layout is shown in [Table 72](#).

Table 72: Eddystone UID Frame

Byte Offset	Field	Description
0	Frame Type	Value = 0x00
1	Ranging Data	Calibrated Tx power at 0 m
2	NID[0]	10-byte Namespace
3	NID[1]	
4	NID[2]	
5	NID[3]	
6	NID[4]	
7	NID[5]	
8	NID[6]	
9	NID[7]	
10	NID[8]	
11	NID[9]	
12	BID[0]	6-byte Instance
13	BID[1]	
14	BID[2]	
15	BID[3]	
16	BID[4]	
17	BID[5]	
18	RFU	Reserved for future use, must be 0x00
19	RFU	Reserved for future use, must be 0x00

More on information on the Eddystone-UID specification can be found in [10].

6.4.3.2 Eddystone-URL

The Eddystone-URL frame broadcasts a URL using a compressed encoding format in order to fit more within the limited advertisement packet. Once decoded, the URL can be used by any client with access to the internet.

Table 73: Frame Specification

Byte Offset	Field	Description
0	Frame Type	Value = 0x10
1	TX Power	Calibrated Tx power at 0 m
2	URL Scheme	Encoded Scheme Prefix
3+	Encoded URL	Length 1 to 17

For URLs longer than 17 bytes, a [URL shortener](#) is recommended.

Table 74 URL Scheme Prefix

Decimal	Hex	Expansion
0	0x00	http://www.
1	0x01	https://www.
2	0x02	http://
3	0x03	https://

The HTTP URL scheme is defined by [RFC 1738](#). The encoding consists of a sequence of characters. Character codes excluded from the URL encoding are used as text expansion codes. When a user agent receives the Eddystone-URL, the byte codes in the URL identifier are replaced by the expansion text according to [Table 75](#).

Table 75: Eddystone-URL HTTP URL Encoding

Byte Offset	Field	Description
0	0x00	.com/
1	0x01	.org/
2	0x02	.edu/
3	0x03	.net/
4	0x04	.info/
5	0x05	.biz/
6	0x06	.gov/
7	0x07	.com
8	0x08	.org
9	0x09	.edu
10	0x0A	.net
11	0x0B	.info
12	0x0C	.biz
13	0x0D	.gov

Byte Offset	Field	Description
14..32	0x0E...0x20	Reserved for future use
127..255	0x7F...0xFF	Reserved for future use

URLs are written only with the graphic printable characters of the US-ASCII coded character set. The octets 00 to 20 and 7F to FF hexadecimal are not used. See “Excluded US-ASCII Characters” in [RFC 3986](#).

IMPORTANT NOTE

In short, the URL-prefix is represented by one byte (see [Table 74](#)), followed by the URL in ASCII, and succeeded by the extension (see [Table 75](#)), if applicable.

Below an example of a URL frame is presented:

```
uint8_t url_adv_data[] =
{
    0x03, // Length of Service List
    0x03, // Param: Service List
    0xAA, 0xFE, // Eddystone ID
    0x0E, // Length of Service Data
    0x16, // Service Data
    0xAA, 0xFE, // Eddystone ID
    0x10, // Frame type: URL
    0xC5, // Power
    0x00, // http://www.
    'd', 'i', 'a', 's', 'e', 'm', 'i',
    0x07, // .com
};
```

During the development of Dialog's Beacon reference design, an Eddystone-URL generator has been used, however it is not essential to create a URL frame whether one follows the specification. More information on the Eddystone-URL specification can be found in [\[11\]](#).

6.4.3.3 Unencrypted Eddystone-TLM

Eddystone Beacons may transmit data about their own operation to clients. This data is called telemetry and is useful for monitoring the health and operation of a fleet of beacons. Since the Eddystone-TLM frame does not contain a beacon ID, **it must be paired with an identifying frame which provides the ID, either of type Eddystone-UID or Eddystone-URL.**

Table 76: Eddystone-TLM Frame Specification

Byte offset	Field	Description
0	Frame Type	Value = 0x20
1	Version	TLM version, value = 0x00
2	VBATT[0]	Battery voltage, 1 mV/bit
3	VBATT[1]	
4	TEMP[0]	Beacon temperature
5	TEMP[1]	
6	ADV_CNT[0]	Advertising PDU count
7	ADV_CNT[1]	
8	ADV_CNT[2]	
9	ADV_CNT[3]	
10	SEC_CNT[0]	Time since power-on or reboot (up timer)

Byte offset	Field	Description
11	SEC_CNT[1]	
12	SEC_CNT[2]	
13	SEC_CNT[3]	

More information on the unencrypted Eddystone-TLM specification can be found in [\[12\]](#).

6.5 Software Features

This section explains the advanced software features of Dialog's BLE Beacon. The following configurations are supported:

- **Non-connectable advertising (Beacon mode)**
 - Allows users to advertise data with the lowest power consumption.
- **Connectable advertising (Peripheral mode)**
 - Allows users to connect to a central device to run SUOTA and use official BLE and custom 128-bit profiles.
- **Dynamically change beacon data**
 - Values `major_ALT_value1` and `minor_ALT_value2` are periodically updated with measured data. In the "altbeacon_dynamic" project, these are data from the environmental sensor.
 - TLM data (up timer, temperature, battery readings) are updated.
- **Support of SPI Flash memory**
 - Power off/on for power saving.
 - Storage of the beacon configuration (product header, configuration structure).
 - Storage of an updated image received through SUOTA.
 - Storage of boot loader.
- **SUOTA**
 - After a connection has been established, the firmware can be updated.
 - Dual image boot loader.
- **Custom 128-bit profiles**
 - Environmental Data Notifications proprietary profile. This service makes data from the environmental sensor (temperature, humidity, pressure) available to the user.
 - Device configuration proprietary profile. The beacon configuration can be read and modified by the central device.

6.6 Beacon Parameters

The following subsections describe how to modify the basic parameters of the beacon: advertising data and advertising interval.

6.6.1 Advertising Data

The data to be advertised is derived from a structure by the name of `user_beacon_config_tag`. The struct contains fields that can serve any beacon format or mode. In [Table 77](#), the `user_beacon_config_tag` struct is analyzed.

Table 77: Format of Struct `user_beacon_config_tag`

Type	Name	Size (B)	Description
uint8_t	uuid[16]	16	UUID value if iBeacon, AltBeacon or Eddystone-UID

Type	Name	Size (B)	Description
uint16_t	major_ALT_val1	2	iBeacon Major value or AltBeacon value 1
uint16_t	minor_ALT_val2	2	iBeacon Minor value or AltBeacon value 2
uint16_t	company_id	2	Manufacturer ID
uint16_t	adv_int	2	Advertising Interval
uint8_t	power	1	Reference value of the received signal strength (RSSI) measured at 1 m. The value is represented in signed format. (Note 1)
uint8_t	beacon_type	1	Beacon Type (iBeacon, AltBeacon, Eddystone UID/URL)
uint8_t	url_prefix	1	http://www. or https://www. or http:// or https://
uint8_t	url[19]	19	The url preceded by the length of service data and succeeded by the extension (.com, .net and others)
uint8_t	TLM_version	1	The TLM version of the TLM service (if TLM is used). (Note 2)
uint8_t	TLM_used	1	Flag that shows if TLM is used or not. (Note 2)

Note 1 To estimate the distance to a transmitting beacon, the receiving device uses and calibrates RSSI by the *measured power* parameter, which is included in the advertising data. The measured power parameter is a 1-byte value in signed representation. The value depends on the RF transmit power and antenna implementation of the hardware. For Dialog's Beacon, the RSSI level measured at 1 m is -59 dBm (see Note 3)

Note 2 Eddystone-TLM is not an "independent" Eddystone beacon mode. It must be paired with either Eddystone-UID or Eddystone-URL (see section 6.4.3.3).

Note 3 Follow these steps to convert the RSSI level (-59 dBm) into signed format:

1. Take positive value: $(59)_{DECIMAL} = (0011\ 1011)_{BINARY}$
2. Reverse all bits: $(0011\ 1011)_{BINARY} \Rightarrow (1100\ 0100)_{BINARY\ REVERSE}$
3. Take 1's complement: $(1100\ 0100)_{BINARY\ REVERSE} \Rightarrow (1100\ 0101)_{1'S\ COMPL}$
4. Convert to hexadecimal: $(1100\ 0101)_{1'S\ COMPL} = (C5)_{HEX}$

Two methods are used to populate the contents of the struct:

- Using the `user_default_beacon_config` struct (defined in the code) and programming the desired values.
- Reading the contents of a device configuration struct in the Flash memory.

The two methods are explained in detail in section 6.6.1.1 and 6.6.1.2.

6.6.1.1 Using the `user_default_beacon_config` Struct

If a Flash memory is not available or not to be used, the contents of the `user_beacon_config_tag` struct are populated with the contents of the `user_default_beacon_config` struct. Below an example of a populated `user_default_beacon_config` struct is presented:

```
const struct user_beacon_config_tag user_default_beacon_config = {
    .uuid = {0x58, 0x5C, 0xDE, 0x93, 0x1B, 0x01, 0x42, 0xCC, 0x9A, 0x13, // 10-byte Namespace
             0x25, 0x00, 0x9B, 0xED, 0xC6, 0x5E}, // 6-byte Instance
    .major_ALT_val1 = 0x0800, // Major/Alt1 Value
    .minor_ALT_val2 = 0x0400, // Minor/Alt2 Value
    .company_id = DIALOG_COMP_ID, // Beacon company ID
    .adv_int = BEACON_ADVERTISING_INTERVAL, // Advertising interval
    .power = 0xC5, // Tx Power
    .beacon_type = xxx_BEACON_TYPE, // Reserved for future use
};
```

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

```
.url_prefix = HTTPWWW, // Populated if used
.url = {0x0E, 'd','i','a','s','e','m','i',DOTCOM},// Populated if used
.TLM_version = 0x00, // Populated if used
.TLM_used = 0x01 // Populated if used
}
```

NOTE

The fields `major_ALT_val1` and `minor_ALT_val2` are in big endian format.

This example shows how users can use the same struct to alternate between different beacon types. Depending on the application, some of the fields of the struct are used and some are not. For example, if the application uses the Eddystone-URL beacon type (see 6.4.3.2), the fields `".uuid"`, `".major_ALT_val1"`, and `".minor_ALT_val2"` are not used. If the beacon type is different, for example, Eddystone-UID (see 6.4.3.1), the fields `".url_prefix"` and `".url"` would not be used. The advertised data packet is synthesized according to the beacon type.

6.6.1.2 Reading Advertising Data from Flash

If a Flash memory is used, the program reads the advertising data values from a device configuration struct that is written in Flash memory. The device configuration struct is by default located at address 0x30000, but this can easily be configured by the value set in the product header (see Appendix D.1.1). The data are written in Flash in the order of the `user_beacon_config_tag` struct. Figure 24 shows an example of the contents of a configuration struct in a Flash memory.

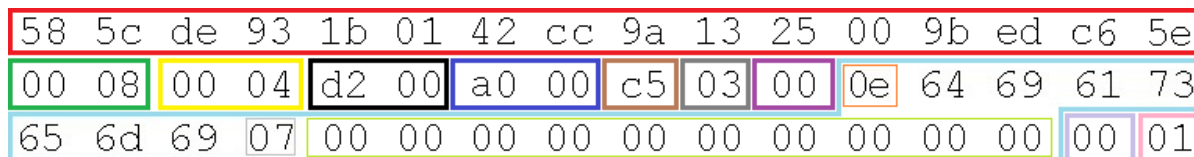


Figure 24: Example of a Device Configuration Struct in Flash Memory

In Figure 24, different colors from left to right represents different components in the format of the `user_beacon_config_tag` struct (see Table 77 for more information):

- Red is the `uuid`
- Green is the `major_ALT_val1` value
- Yellow is the `minor_ALT_val2` value
- Black is the `company_id`
- Blue is the `adv_int`
- Brown is the `Tx power`
- Grey is the `beacon_type`
- Purple is the `url_prefix`
- Light blue is the `url` in ASCII (here it is "diasemi") with:
 - Orange is the length of the service data
 - Light grey is the URL postfix (here it is the code for ".com")
 - Light green bytes are not needed (because this specific URL is shorter) and are set to 0x00
- Light purple is the `TLM_version`
- Pink is the `TLM_used` flag

NOTE

The values are in opposite endianness compared to the code.

6.6.2 Advertising Interval

The BLE Beacon sends advertising packets at a certain time interval. This is called the advertising interval. According to the BLE specification, the advertising interval can be set from 20 ms up to 10.24 s.

At a shorter advertising interval, the radio of the beacon will be enabled more often, resulting in higher average power consumption. A shorter advertising interval will also lead to an increased number of packets per second at the receiver and therefore result in a more accurate RSSI reading.

If the advertising data derive from Flash, the advertising interval takes the value of the `adv_int` field of the device configuration struct written in flash (Figure 24, blue field).

If no Flash memory is used, the advertising interval parameter can be changed at the location shown in Table 78.

Table 78: Advertising Interval Location

Parameter	Macro	Project	File Name
Beacon advertising interval	BEACON_ADVERTISING_INTERVAL	All beacons	user_config.h
SUOTA advertising interval	SUOTA_ADVERTISING_INTERVAL	All beacons	user_config.h

6.7 Software Architecture

Figure 25 presents how the Beacon reference applications are structured.

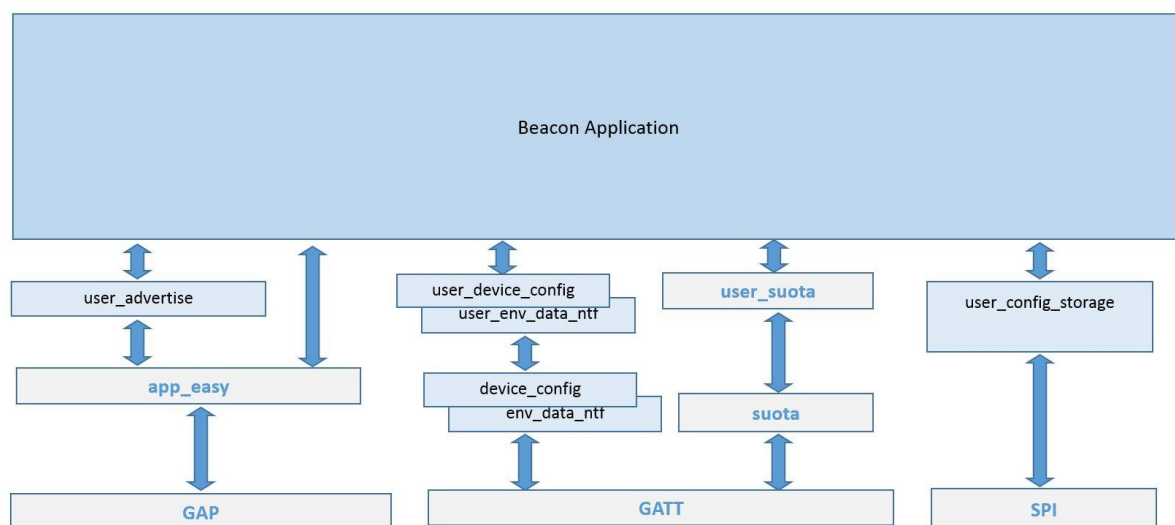


Figure 25: Beacon SW System Overview

Table 79 shows the source files of the Beacon reference applications.

Table 79: Source Files of Beacon Reference Applications

Group	File Name	Project	Description
user_platform	user_periph_setup.c	All beacons	Peripheral modules initialization, GPIO pins assignment
user_config_storage	user_config_storage.c	All beacons	Application configuration data storage API
device_config	device_config.c	Eddystone,	Device Configuration profile

Group	File Name	Project	Description
	device_config_task.c user_device_config.c user_device_config_task.c	iBeacon	
env_data_ntf	env_data_ntf.c user_env_data_ntf.c env_data_ntf_task.c user_env_data_ntf_task.c	Eddystone	Environmental Data Notifications Profile
user_drivers	i2c_gpio_extender.c user_iot_dk_utils.c battery.c	All beacons (battery.c not in AltBeacon)	User drivers for the I2C GPIO extender, battery and MSK HW peripherals
bme680_drivers	bme680.c bme680_implc.c	Eddystone, AltBeacon	Environmental Sensor Drivers
sensor_inteface	sensors_inteface.c sensors_inteface_api.c environmental_bme680.c	Eddystone, AltBeacon	Sensors interface API
utilities	wkup_adapter.c sensors_periph_interface.cc rc32.c	All beacons	Various utilities for wakeup, crc and peripherals
user_beacon	user_eddy_uid_url_tlm.c	Eddystone	Application Code
	user_altbeacon_dynamic.c	AltBeacon	
	user_ibeacon_suota_button.c	iBeacon	
user_adv_api	user_advertise.c	All beacons	User Advertise SW module

6.8 Operation Overview

Figure 26 presents the system operation.

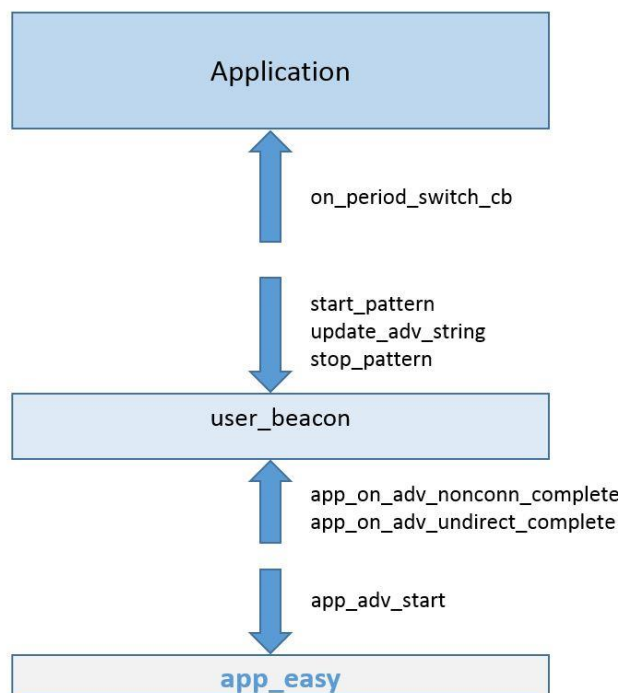


Figure 26: Operation Overview

6.8.1 Configuration Switches

The code contains various configuration switches which include or exclude functionalities from the code. They are divided into two categories:

- Software configuration switches
- Profile configuration switches

The various configuration switches are outlined in [Table 80](#) and [Table 81](#).

Table 80: List of Software Configuration Switches

Switch	Used in	Description
CFG_CONFIG_STORAGE	AltBeacon, Eddystone, iBeacon	Read/Write configuration from/to Flash memory enabled.
CFG_DYNAMIC_BEACON_DATA	AltBeacon	If defined, the major/minor values of the beacon are updated dynamically with data from the Environmental sensor.
CFG_DEV_CNF_HDR_CRC32_SUPPORT	AltBeacon, Eddystone, iBeacon	If defined, a CRC32 value of the configuration struct data is calculated and compared to the CRC_word included in the configuration struct header.
USE_EDDYSTONE_UID/USE_EDDYSTONE_URL	Eddystone	Eddystone-UID or Eddystone-URL packets will be advertised depending on which will be defined.

Table 81: List of Profile Configuration Switches

Switch	Used in	Description
CFG_PRF_DEVICE_CONFIG	Eddystone, iBeacon	Custom 128-bit "Device Configuration" profile is enabled.

Switch	Used in	Description
CFG_PRF_ENV_DATA_NTF	Eddystone	Enables custom 128-bit "Environmental Data Notifications" profile.
CFG_PRF_SUOTAR	iBeacon	Dialog SUOTA profile is enabled. CFG_SPI_FLASH switch must be defined.
CFG_PRF_DISS	Eddystone, iBeacon	Device Information Service Server (DISS) profile.
CFG_PRF_BASS	Eddystone, iBeacon	Battery Service profile. Battery readings are updated using an advanced battery reporting mechanism.

6.9 User Advertise SW Module

The user advertise SW module provides users with an easy API to create configurable advertising strings with easily configurable advertising parameters (for example, advertising interval). Below the structural components of the user advertise module are outlined.

6.9.1 Style

The main block of the user advertise SW module is called a **style**. Below, the style type is presented:

```
/// Advertising Style
typedef struct user_adv_style
{
    /// Advertising interval
    uint16_t adv_int;
    /// Advertising length
    uint8_t adv_len;
    /// Counter of advertising events
    uint16_t cnt_upd_adv_string;
    /// Advertising mode
    uint8_t adv_mode;
    /// Advertising data
    uint8_t adv_data[31];
} user_adv_style_t;
```

An advertising style contains information about:

- Advertising interval
- Length of the advertising string
- Amount of advertising events of that specific style before switching to another style
- Advertising mode (undirected or non-connectable)
- A 31-byte array containing the actual data to be advertised.

6.9.2 Pattern

An array of styles is called a **pattern**. The maximum number of styles allowed per pattern is configurable and is defined by the `MAX_STYLES_PER_PATTERN` macro.

The value of the `cnt_upd_adv_string` field of each style shows how many advertising events will take place before a style switches to the next. Below an example of a pattern is presented.

```
user_adv_style_t ibeacon_suota_pattern[2] =
{
    {
        .adv_int = iBEACON_ADV_INT,
        .adv_len = iBEACON_ADV_LEN,
        .cnt_upd_adv_string = iBEACON_EVENTS,
        .adv_mode = NON_CONNECTABLE_MODE,
```



```

        .adv_data = <pointer to a data array>
    },
    {
        .adv_int = SUOTA_ADVERTISING_INTERVAL,
        .adv_len = USER_ADVERTISE_DATA_LEN,
        .cnt_upd_adv_string = SUOTA_EVENTS_TO_SWITCH,
        .adv_mode = UNDIRECTED_MODE,
        .adv_data = USER_ADVERTISE_SUOTA,
    }
};

```

In this example, a pattern is declared, consisting of two styles. The data for the second style in the pattern is derived from the array shown below:

```

#define USER_ADVERTISE_DATA    "\x05"\
                                ADV_TYPE_COMPLETE_LIST_16BIT_SERVICE_IDS\
                                ADV_UUID_DEVICE_INFORMATION_SERVICE\
                                ADV_UUID_SUOTAR_SERVICE\
                                "\x0E"\
                                ADV_TYPE_COMPLETE_LOCAL_NAME\
                                USER_DEVICE_NAME

```

Figure 27 presents how to use a user advertise SW block.

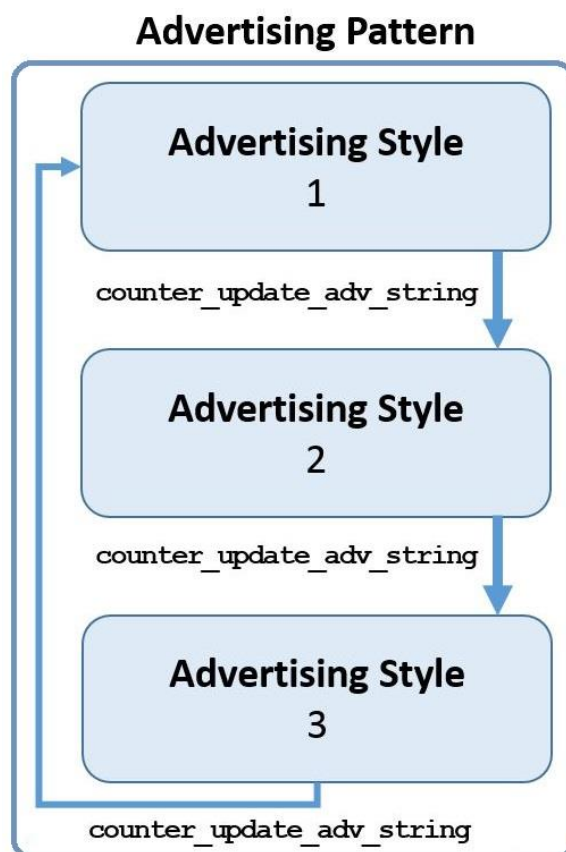


Figure 27: User Advertise Usage Example

In this example, the pattern contains three styles. When `cnt_upd_adv_string` of the first style is reached, in other words, `<cnt_upd_adv_string>` events have been advertised, the application switches to the second style. The same happens when the `cnt_upd_adv_string` of the second style is reached. However, when `< cnt_upd_adv_string >` events of the third advertising style have been

advertised, the application starts advertising again from the first style. The row of events, or the times a style is advertised, depends on the application.

6.9.3 User Advertise SW Module Callbacks

The User Advertise SW Module makes two callbacks to the application and users can use these callbacks for any desired usage. In the provided examples, a callback is used to inform the application of advertising events and the other callback is called every time the `user_on_ble_powered` system callback is called. More information on the User Advertise SW module API is included in the doxygen documentation that is included in the release.

NOTE

The system callback `user_on_ble_powered` updates the advertising string.

6.10 Device Configuration Service

The Device Configuration Service is a custom 128-bit profile developed by Dialog Semiconductor. The profile provides a generic interface to a peripheral device for reading and writing configuration parameters of the application, irrespective of the number, type, and size of the parameters.

6.10.1 Device Configuration Service Specification

The Device Configuration Service provides four characteristics outlined in [Table 82](#).

Table 82: Characteristics of the Device Configuration Service

Characteristic Name	Qualifier	Properties	Size (B)
Configuration structure version	Mandatory	Read	66
Write configuration	Mandatory	Write	1
Read command	Mandatory	Write	1
Read response	Mandatory	Indicate	67

- **Configuration structure version:**

- It identifies the type and version of the configuration structure of the application.
- It is used as a convention between device configuration server and client for the format of the configuration data.
- The Device configuration client should be aware of the ID and size of each configuration parameter.

- **Write configuration:**

- The Device configuration client writes the configuration data into this characteristic.
- The data format is shown in [Figure 28](#):

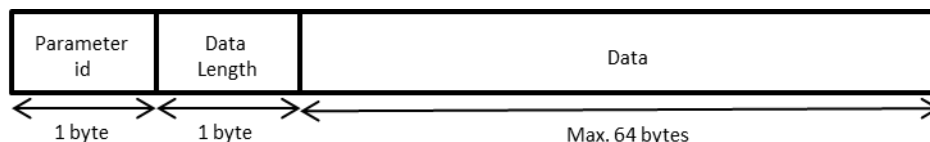


Figure 28: Data Format in Write Configuration

- The maximum configuration data size supported by the Device configuration service is 64 bytes. Up to 256 parameters can be supported. The client can read the parameter data after the end of a write operation.

- **Read Command:**

- The Device configuration client writes a parameter ID into this characteristic, requesting the command server to return the current values of the parameters.
- **Read Response:**
 - The Device configuration server sends an indication including the configuration data of a parameter, whenever the client requests it by writing the parameter ID to the read command characteristic.
 - The format of the indication data is shown in [Figure 29](#).

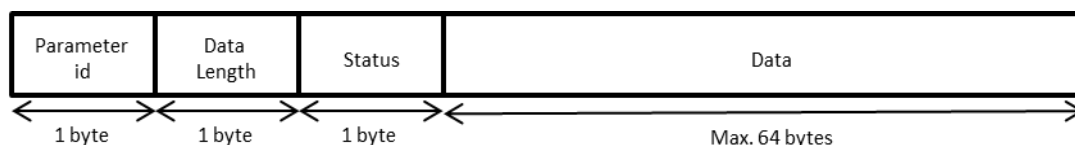


Figure 29: Indication Data Format in Read Response

6.11 Environmental Data Notifications Service

The Environmental Data Notifications Service is a custom 128-bit profile developed by Dialog Semiconductor. The profile provides a generic interface to a peripheral device providing data from the environmental sensor (temperature, humidity, and pressure).

6.11.1 Environmental Data Notifications Service Specification

The Environmental Data Notifications Service provides five characteristics outlined in [Table 83](#).

Table 83: Characteristics of the Environmental Data Notifications Service

Characteristic Name	Qualifier	Properties	Size (B)
Temperature	Mandatory	Read	2
Pressure	Mandatory	Read	4
Humidity	Mandatory	Read	4
External Event	Mandatory	Read	1
Sampling Interval	Mandatory	Read/Write	2

- **Temperature:**
 - The "Temperature" characteristic carries the temperature value read by the environmental sensor.
 - Executing a read command, the current temperature value from the sensor will be transmitted. The same will happen upon a button press, if characteristic notifications are enabled.
- **Pressure:**
 - The "Pressure" characteristic carries the pressure value read by the environmental sensor.
 - Executing a read command, the current pressure value from the sensor will be transmitted. The same will happen upon a button press, if characteristic notifications are enabled.
- **Humidity:**
 - The "Humidity" characteristic carries the humidity value read by the environmental sensor.
 - Executing a read command, the current humidity value from the sensor will be transmitted. The same will happen upon a button press, if characteristic notifications are enabled.
- **External Event:**
 - The "External Event" characteristic carries the external event state (in our case, the state of the button).

- Executing a read command, the current button state will be transmitted. The same will happen upon a button press, if characteristic notifications are enabled.
- **Sampling Interval:**
 - The "Sampling Interval" characteristic carries the value of the timer interval during which the environmental sensor samples environmental data.
 - These data are used to update the values of the aforementioned characteristics (temperature, pressure, and humidity).

6.12 Beacon Configuration

Depending on the defined software configuration switches, the configuration data can be modified by a central device over the Device Configuration Service or written into SPI Flash memory during the production phase.

The Beacon application reference software uses a configuration storage module, implemented in file `user_config_storage.c`, to fetch and store configuration data from and into the non-volatile memory. The configuration storage module provides a list of common API functions to any DA14585 application. The current version only supports SPI Flash memory.

6.12.1 Beacon Configuration Memory Map

Configuration storage uses the memory map of the dual image boot loader. More specifically, it uses the first four bytes of the 'Reserved' field (byte offset 12) in the Product Header, as described in [Appendix D.1.1](#), to define the memory address of the configuration area. The address shall point to the start of an SPI Flash sector and no other information shall be stored in the same sector.

Device configuration data are stored at the start of the configuration area and consist of the device configuration header, followed by the device configuration struct. The size of the configuration data is 112 bytes and the format is outlined in [Table 84](#).

Table 84: Configuration Data Format

Byte #	Field
Device Configuration Header	
0, 1	Signature (0x70, 0x53)
2	Valid flag
3	Number of items
4 to 7	CRC
8 to 23	Version
24, 25	Data size
26	Encryption flag
27 to 63	Reserved
Device Configuration Struct	
64 to 79	UUID
80, 81	Major_ALT_val1
82, 83	Minor_ALT_val2
84, 85	Company ID
86, 87	Advertising Interval
88	Tx Power
89	Beacon Type (iBeacon/AltBeacon/Eddystone (UUID/URL))

Byte #	Field
90	URL prefix
91 to 109	URL
110	TLM version
111	TLM_used flag

The fields included in the header are:

- **Signature:**
 - A "magic" number identifying the configuration header.
 - Value: 0x70, 0x53.
- **Valid flag:**
 - A value of 0xAA denotes a valid image.
- **Number of items:**
 - The number of configuration parameters in the configuration data.
- **CRC (If used):**
 - The checksum calculated over the configuration data.
- **Version:**
 - Determines the configuration structure type and version.
 - Can be checked by the application to confirm that the expected data are stored.
- **Data size:**
 - Size of the configuration data (in bytes).
- **Encryption flag:**
 - Indicates whether the configuration data have been encrypted.
 - Encryption of the configuration in memory is not supported in this software release.
- **UID:** The value in this field serves different uses depending on the mode used.
 - If iBeacon is selected, this field contains the 16-byte UUID value.
 - If AltBeacon is selected, this field contains the 16-byte Beacon ID value ([Note 1](#)).
 - If Eddystone-UID is selected, this field contains the 10-byte namespace value followed by the 6-byte instance value.
- **Major_ALT_val1:**
 - Contains the Major value for iBeacon or the Alt_val1 for AltBeacon.
- **Minor_ALT_val2:**
 - Contains the Minor value for iBeacon or the Alt_val2 for AltBeacon.
- **Company ID:**
 - The beacon device manufacturer's company identifier code, see [\[7\]](#).
- **Advertising Interval:**
 - The advertising interval for the beacon (see section [6.6.2](#)).
- **Tx Power:**
 - A 1-byte value representing the average RSSI at 1 m from the advertiser.
- **Beacon Type:**
 - The beacon type included in the Configuration Struct.
- **URL Prefix:**
 - If Eddystone-URL beacon type is selected, this field defines the URL prefix (see [Table 74](#)).
- **URL:**

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- If Eddystone-URL beacon type is selected, this field contains the URL in ASCII, preceded by the length of service data and succeeded by the extension (.com, .net, and others).
- **TLM_version:**
 - TLM version is reserved for future development of this frame type. At present the value must be 0x00.
- **TLM_used flag:**
 - This flag indicates whether the TLM service is used or not.

6.13 Battery Level Sampling

If `BATTERY_SAMPLING_ENABLED` is defined, a battery level and voltage averaging mechanism is enabled.

A user driver has been developed to achieve more accurate battery readings. The main idea is to periodically capture an ADC sample from the VBAT3V signal immediately after the device wakes up to avoid any possible battery consuming activities. These samples allow the driver to calculate average values for the battery voltage and battery level. The calculated average level and voltage values can then be used by the application.

Upon connection, if the BASS service is enabled, `app_batt_set_level(battery_return_avg_lvl())` is called and sets the battery level in the BASS service, making the averaged battery level available when the device is connected.

6.14 Beacon Examples for DA14585 IoT MSK

The Beacon reference applications based on DA14585 IoT MSK HW reference design comes with three distinct beacon examples that make use of all the different beacon types and features supported by Dialog Semiconductor. Below these examples are outlined.

6.14.1 AltBeacon

In this example a non-connectable AltBeacon string is advertised containing a 16-byte UID followed by four bytes (two bytes for `ALT_Val1` and two bytes for `ALT_Val2`). If the software-built flag `DYNAMIC_BEACON_DATA` is not enabled, `ALT_Val1` and `ALT_Val2` are populated by the values set in the configuration struct in flash (if `CONFIG_STORAGE` is enabled) or by the default values hardcoded in the default configuration struct. If `DYNAMIC_BEACON_DATA` is enabled, the 2-byte values are populated by environmental sensor data, refreshed every `periodic_read_interval` set by users when initializing the environmental sensor.

Figure 30 presents the advertising transition diagram of the AltBeacon example.

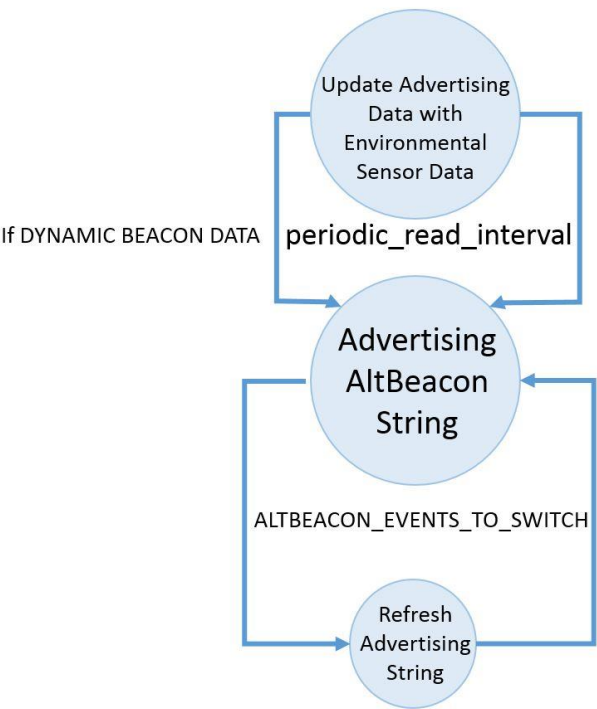


Figure 30: AltBeacon Example Transition Diagram

6.14.1.1 AltBeacon Example Sequence Diagram

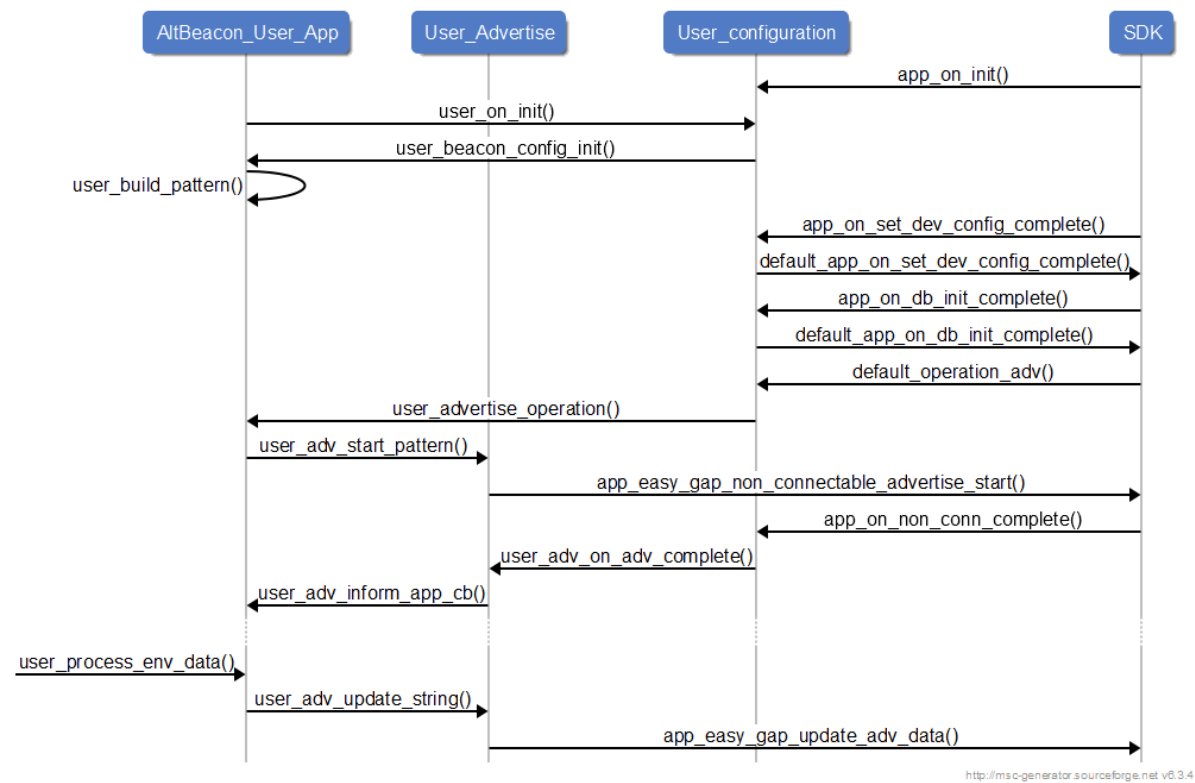


Figure 31: Altbeacon Sequence Diagram

6.14.2 Eddystone

In this example, a connectable Eddystone-UID or Eddystone URL string is advertised, depending on the corresponding build flags `USE_EDDYSTONE_UID` or `USE_EDDYSTONE_URL`.

The advertising string contains a 16-byte UID if the flag `USE_EDDYSTONE_UID` is used.

If the flag `USE_EDDYSTONE_URL` is used, an encoded URL with a length ranging from 1 to 17 bytes is contained in the advertising string.

Every time when `user_adv_on_adv_complete` is called, the application advertises an EDDYSTONE-TLM advertising string and then returns to advertising Eddystone-UID or Eddystone-URL strings. As explained in 6.4.3.3, the Eddystone-TLM packet contains information about the battery voltage of the device. The application uses the battery averaging mechanism described in 6.13 to provide more accurate voltage values to the Eddystone-TLM advertising string.

When connected to a peripheral, the device provides four different GATT services: DISS and BASS which are official BLE GATT services and two Dialog proprietary GATT services, the `env_data_ntf` and `device_config`. The `device_config` and `env_data_ntf` services are described in 6.10 and 6.11.

Figure 32 presents the advertising transition diagram of the Eddystone example.

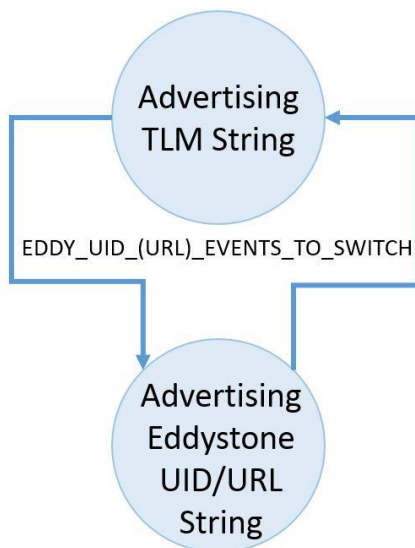


Figure 32: Eddystone UID/URL/TLM Example Transition Diagram

6.14.2.1 Eddystone Example Sequence Diagram

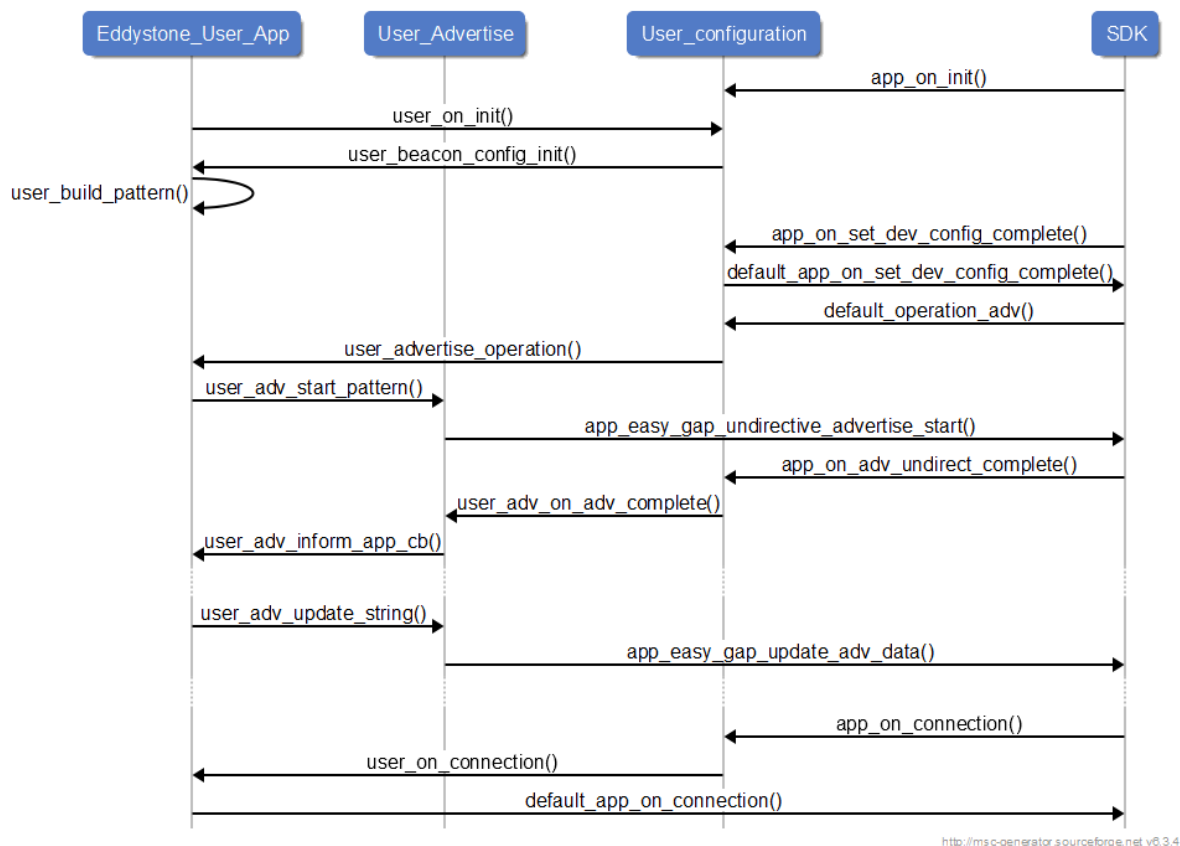


Figure 33: Eddystone Sequence Diagram

6.14.3 iBeacon

In this example, a non-connectable iBeacon string is advertised. The Major and Minor values are populated by the values set in the configuration struct in flash (if `CONFIG_STORAGE` is enabled) or by the default values hardcoded in the default configuration struct. On button press the device starts advertising a connectable SUOTA string for a duration of `ADV_SUOTA_TIMEOUT` set in `ibeacon_suota_button.h`. However, if during this time period there is another button press, the application returns to advertising the non-connectable iBeacon string.

Figure 34 presents the advertising transition diagram of the iBeacon example:

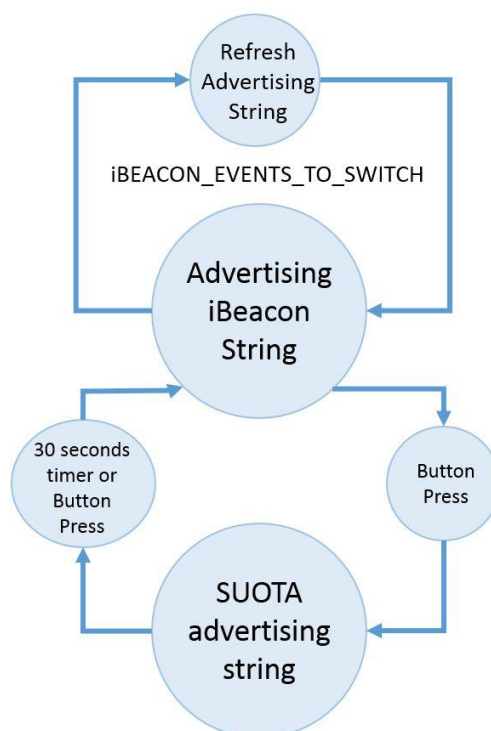


Figure 34: iBeacon Example Transition Diagram

6.14.3.1 iBeacon Example Sequence Diagram

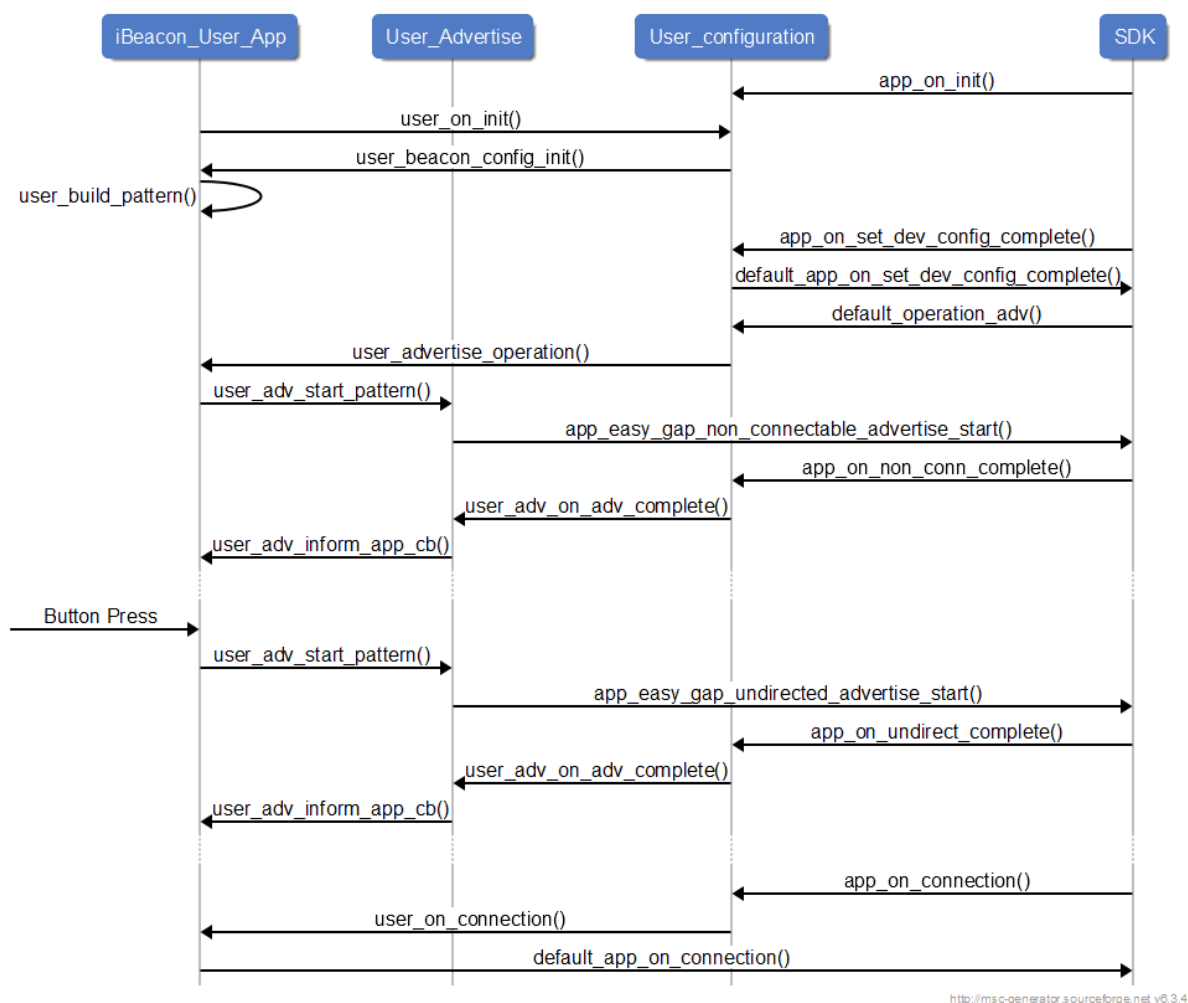


Figure 35: iBeacon Sequence Diagram

7 Memory Footprint and Power Measurements

7.1 Memory Footprint

The SYSRAM footprint of the reference applications in the DA14585 IoT MSK reference design are depicted in [Table 85](#).

Table 85: Memory Footprint

Application	Code	RO Data	RW Data	ZI Initialized
IoT Sensors	66416	6420	416	14764
IoT Sensors without Air Quality library	46124	4556	400	13608
Smart Tag	28332	2952	8	7960
iBeacon	27056	4004	344	15756
Eddystone	30140	3936	748	15524
AltBeacon dynamic	21096	2656	60	15416

Besides the application memory footprint, there is a memory area of around 9.5 Kbytes, depending on the application, reserved for the exchange memory between BLE controller layer and BLE Core.

7.2 Power consumption

Table 86 presents the average power consumption of the most commonly used operation modes of the reference applications of DA14585 IoT MSK.

Table 86: Power Consumption

Mode	Current (avg)
IoT Sensors	
Advertising: Interval at 100 ms and LED blink	313.596 μ A
Idle (waiting for motion to advertise)	20.679 μ A
All sensors: Accelerometer and Gyro at 100 Hz Sensor Fusion and Magneto 10 Hz Temperature, Humidity, Pressure, and Air quality at 0.3 Hz Ambient Light and proximity at 0.5 Hz	2.008 μ A
Motion sensors: Accelerometer and Gyro at 100 Hz Sensor Fusion and Magneto 10 Hz	1371.032 μ A
Environmental sensor Temperature, Humidity, Pressure, and Air quality at 0.3 Hz	1054.313 μ A
Optical sensor Ambient Light and proximity at 0.5 Hz	488.423 μ A
Smart Tag	
Advertising	295.977 μ A
Connected	158.757 μ A
Beacons	
iBeacon Advertising	37.17 μ A
iBeacon Connected	124.148 μ A
Eddystone Advertising: Environmental sampling at 0.25 Hz	49.198 μ A
Eddystone Connected Environmental sampling at 0.25 Hz	138.226 μ A
AltBeacon Advertising	41.146 μ A

Appendix A MSK Boot Sequence

The MSK boot sequence consists of the following stages:

- BootROM sequence (*sections 4.4, 4.4.3, [5]*).
- Secondary Bootloader (*Appendix H, [2]*) is part of the released code inside folder `utilities\secondary bootloader`.
- Application Code.

For the application to be initialized correctly, an architecture that uses Secondary Bootloader with mirrored images is used. The secondary bootloader serves four purposes:

- Initializes the IMU (ICM42605 or BMI160) device to operate in SPI mode.
 - If offset 0xFE20 in OTP (user area) is not programmed as 0x0, the IMU initialization is skipped.
 - Please be aware that if this bit is set, it cannot be undone, and the device may be unable to boot.
- Scans UART during UART boot.
- Selects and copies the most recent application image from the flash memory to SRAM.
- Passes control to application

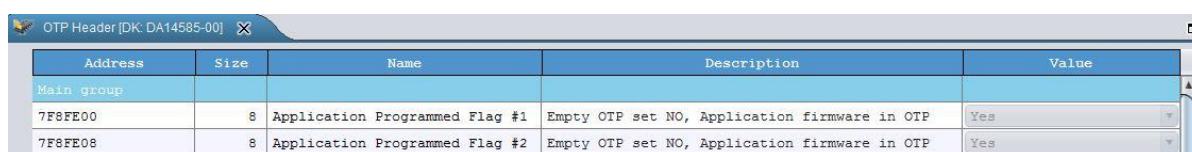
The Secondary Bootloader is copied from OTP offset 0x0000 to the end of SRAM, so it is crucial to keep the footprint of the Secondary Bootloader as small as possible.

In non-MSK boards, in order to use the Secondary Bootloader, users should follow the following steps:

1. Burn the generated `secondary_bootloader_585.bin` into offset 0x0000 of OTP.
2. Program the two application flags located in offsets 0xFE00 and 0xFE08 of OTP memory using [SmartSnippets](#) as in [Figure 36](#).

NOTE

The MSK boards have the Secondary Bootloader and the OTP application flags written into OTP when shipped.



Address	Size	Name	Description	Value
Main group				
7F8FE00	8	Application Programmed Flag #1	Empty OTP set NO, Application firmware in OTP	Yes
7F8FE08	8	Application Programmed Flag #2	Empty OTP set NO, Application firmware in OTP	Yes

Figure 36: Application Programmed in OTP Flags

Appendix B Memory Map

[Figure 37](#) shows the default memory locations of the different parts of the various images for all projects. The figure also shows that the product header, among other information, contains the offsets of the images and the configuration struct. The offsets of the two images and the configuration struct (in beacon projects) can be modified in the product header, thus enabling users to write the images and the struct in those offsets. The secondary bootloader will “look” for those offsets when booting the device. See [Appendix D.1](#). Please pay attention to the distance of the memory locations to fit the size of the corresponding parts (product header, application image, and others).

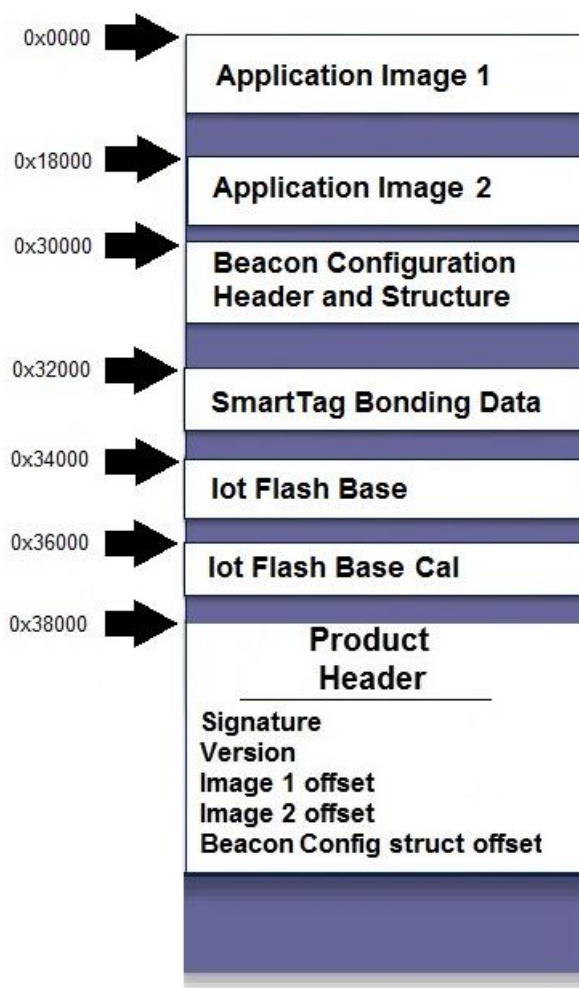


Figure 37: Analyzing a Flash Memory Image

At this point, it is important to note that all these different parts **do not exist all at once**. [Table 87](#) shows which part exists and at which location depending on the application.

Table 87: Parts of the Image Depending on the Application

	All Beacons	Smart Tag	IoT Sensors
Application Image 1	X	X	X
Application Image 2	X	X	X
Beacon Configuration Header and Struct	X		
SmartTag Bonding Data		X	
IoT Flash Base (Configuration Data)			X
IoT Flash Base Cal (Sensor Calibration Data)			X
Product Header	X	X	X

Appendix C Using the mkimage Application

C.1 mkimage Scripts

The `mkimage` scripts that run the `mkimage` modes and create the desired multi-image for the Smart Tag and IoT Sensors reference applications or the whole images for the Beacon reference application has been created to help users.

The `mkimage` modes are analyzed in Appendix C.2. The multi-images or whole images are in `.bin` format and are written in Flash. The multi-images contain two alternative images of the application as well as a product header at the end. The whole images are essentially multi-images (two images with a product header) but also include a configuration struct for the Beacon.

The available `mkimage` scripts as well as the various files needed for the scripts to work are located (or should be placed) in "...\\utilities\\mkimage_utils_scripts" and are shown in Table 88.

Please note that the ".hex" extension for all images is added by the script automatically and should be:

- C:>iot_image_folder>make_image_beacon.bat altbeacon_dynamic
- C:>iot_image_folder>make_image_iot.bat io585_585
- C:>iot_image_folder>make_image_tag.bat smart_tag_585
- C:>iot_image_folder>make_all_images.bat (no parameters, builds everything if all the needed .hex files exist in the build folder).

Table 88 Available mkimage Scripts

Script	Purpose	Needed Files
make_image_beacon.bat	Creates a whole image for the beacon applications	<beacon_project>.hex, dev_conf_struct_default.cfg, user_config_sw_ver.h (struct version), beacon_sw_version.h (Note 1)
make_image_iot.bat	Created a multi-image for the IoT sensors application	<iot_project>.hex, iot_sw_version.h (Note 2)
make_image_tag.bat	Creates a multi-image for the Smart Tag application	<tag_project>.hex, tag_sw_version.h (Note 3)
make_all_images.bat	Creates whole images for all beacon applications and multi images for the IoT sensors and Smart Tag application by calling all aforementioned bats.	This script only works when the .hex files are named as following: ibeacon_suota_button.hex altbeacon_dynamic.hex, eddy_uid_url_tlm.hex. smart_tag_585.hex, iot585_585.hex and the corresponding sw_version files and beacon configuration files are

		also present.
clean.bat	Removes all .img and .bin files from the folder	N/A

Note 1 If an alternative .hex file exists, it should go by the name <beacon_project>_1.hex and a file by the name "beacon_sw_version1.h" should be provided.

Note 2 If an alternative .hex file exists, it should go by the name <iot_project>_1.hex.

Note 3 If an alternative .hex file exists, it should go by the name <tag_project>_1.hex.

C.2 mkimage Modes

The mkimage application has different modes to create desired images.

- **Single:** creates an .img file from the .bin file of the Keil project.
- **Multi:** creates a .bin file from the .bin file of the Keil project that contains two alternative .img files that are needed when using the SUOTA functionality and the product header.
- **Whole_img:** creates an .img file containing two alternative .img files that are needed when using the SUOTA functionality, the config_struct.cfg file, the product header, and optionally the bootloader.bin file.
- **Multi_no_suota:** creates an .img file containing the config_struct.cfg file, the product header, and the .bin file of the Keil project, which is preceded by an AN-B-001 header [16].
- **cfg:** creates a .cfg file containing a device configuration struct preceded by its header.

IMPORTANT NOTE

For the mkimage app to work, all needed files should be brought in the mkimage folder where the mkimage.exe will be located after building the mkimage project.

Typing "mkimage" in the command console shows users instructions on the syntax needed to create an .img file for all modes of the application.

C.2.1 mkimage single

The "mkimage single" mode is used to create an .img file from the .bin file of the Keil project. This image contains the software version and the software version date. The .img files created in this mode are used for manually burning images one-by-one at specific addresses in Flash memory using the [SmartSnippets Studio](#) (see Appendix D.2).

Example: `mkimage single my_project_1.bin sw_version.h img_1.img`

Users should also make a second (different) image file so that during the SUOTA procedure the SUOTA application can find another image to load. Users should either include a different .bin file, a different sw_version.h file, or both.

Example: `mkimage single my_project_2.bin sw_version.h img_2.img`

C.2.2 mkimage multi

The "mkimage multi" mode is used to create a .bin file from the .bin file of the Keil project. This bin file contains two images created by the "mkimage single" mode and a product header at the end of the file. Optionally, the image can be created for an SPI Flash memory or an EEPROM Flash memory. The .bin files created by this mode are used for burning in Flash memory using the [SmartSnippets Studio](#) (see Appendix D.2).

Example:

`mkimage multi spi img_1.img 0x0 img2_1.img 0x18000 0x38000 multi_myproject.bin`

C.2.3 mkimage whole_img

The "mkimage whole_img" mode is used to create a complete .img file, containing two alternative .img files created by "mkimage single" mode that are needed when using the SUOTA functionality, the config_struct.cfg file and the product header.

Example:

```
mkimage whole_img img_1.img 0x0 img_2.img 0x18000 config.cfg 0x30000 0x38000
whole_%1.bin
```

The offsets 0x0, 0x18000, and 0x30000 correspond to the file that precedes them: 0x0 is the offset where img_1.img is written and so on. The final offset (in this example 0x38000) is the offset where the product header is written.

C.2.4 mkimage multi_no_suota

The "mkimage multi_no_suota" mode is used to create a whole .img file containing the .bin file of the Keil project preceded by an AN-B-001 header and the config_struct.cfg file. Optionally, the image can be created for an SPI Flash memory or an EEPROM Flash memory. The generated image will not include a SUOTA functionality.

In "mkimage multi_no_suota" mode, no ".img" file generated by the "mkimage single" mode is needed.

Example:

```
mkimage multi_no_suota spi out585.bin dev_conf_with_header.cfg 0x30000 0x38000 out.img
```

In this example the out585.bin file (preceded by an AN-B-001 header) is written at address 0x00. 0x38000 refers to the offset where the product header is written, whereas 0x30000 refers to the offset of the config_struct.cfg file.

C.2.5 mkimage cfg

The "mkimage cfg" mode is used to create a .cfg file containing a device configuration struct preceded by its header. The device configuration struct header also contains a 4-byte CRC which is calculated from the fields of the configuration struct. The application also checks a software version file and includes the version in the header of the corresponding field.

Example: mkimage cfg dev_conf.bin sw_ver.h dev_conf_with_header.cfg

Appendix D Flash Programming in MSK Applications

D.1 Basic Information About the MSK Applications

The programmed devices come with the secondary bootloader already burned in the OTP memory. Upon booting, the secondary bootloader is expected to find the product header at address 0x38000, so that information on the signature, version, and the image offsets can be retrieved. [Appendix B](#) presents the different locations of MSK applications memory map. The following subsections present the formats of the product header, the image header, and the device configuration struct (for beacon projects).

D.1.1 Product Header

Table 89: Product Header Format

Byte #	Field	Description
0, 1	Signature	Product Header signature (0x70, 0x52)

Byte #	Field	Description
2, 3	Version	Version of the product header
4 to 7	Offset1	Start address of the first image
8 to 11	Offset2	Start address of the second image
12 to 17	RFU	Reserved for future use
18 to 21	Config_offset	Start address of the device config struct (for beacon projects)

D.1.2 Image Header

The Image Header format, which is common for any image created with `mkimage.exe`, is shown in [Table 90](#). The application checks the image header when attempting to write in Flash memory.

Table 90: Image Header Format

Byte	Field	Description
0, 1	Signature	Image Header signature (0x70, 0x51)
2	Valid flag	To be set to 0xAA (<code>STATUS_VALID_IMAGE</code>) at the end of the image burning
3	ImageID	Used to determine which image is newer
4 to 7	Code size	Image size
8 to 11	CRC	Image CRC (Not checked in current version)
12 to 27	Version	Version of the image
28 to 31	Timestamp	Time stamp
32	Encryption	Encryption flag
33 to 63	Reserved	For future use

D.1.3 Beacon Configuration Struct and Configuration Struct Header

The device configuration struct is preceded by the device configuration header in the Flash memory. [Table 91](#) and [Table 92](#) show the Device Configuration Header and the Device Configuration struct, respectively.

Table 91: Beacon Configuration Header

Byte #	Field	Description
0, 1	Signature	Device Config Signature (0x70, 0x53)
2	Valid flag	A value of 0xAA denotes a valid image
3	Number of items	The number of configuration parameters in the configuration data
4 to 7	CRC	CRC of the device configuration struct
8 to 23	Version	Determines the configuration structure type and version
24 to 25	Data size	Size of the configuration data
26	Encryption flag	(Not supported in current version)
27 to 63	Reserved	-

The Beacon Configuration struct appears in Flash memory in the same way as in the code (see section [6.6.1.1](#)).

Table 92: Beacon Configuration Struct Format

Byte #	Field	Description
0 to 15	UUID	Beacon Universally Unique ID
16, 17	Major	Major Value (MSB first)
18, 19	Minor	Minor Value (MSB first)
20, 21	Company_id	Beacon Company id (MSB first)
22, 23	Adv_int	Advertising interval (MSB first)
24	Power	Beacon output power at 1 m
25	beacon type	iBeacon/AltBeacon/Eddystone (not used in provided examples)
27	url prefix	URL prefix for Eddystone-URL
28 to 46	url[19]	The URL preceded by the length of service data and succeeded by the extension (.com, .net, and others)
47	TLM_version	TLM version
48	TLM_used	Flag to indicate whether Eddystone-TLM is used or not

D.1.4 Smart Tag Bonding Data, IoT Flash Base, IoT Flash Base Cal

Smart Tag bonding data, IoT Flash base, and IoT Flash base cal (Appendix B) are spaces used by the Smart Tag and IoT sensors applications. They are initially blank but are populated by the application for their needs. In that sense, they cannot be configured by users.

D.2 Flash Programming

D.2.1 Burning the Whole Image in Flash Memory

The procedure to make use of the provided `mkimage` scripts and files utilized by the scripts has been thoroughly explained in Appendix C.1.

With [SmartSnippets Studio](#), users can follow the steps below to burn the generated `.bin` file (multi-image for Smart Tag and IoT sensors applications and whole image for beacon applications) (see Appendix C):

1. Open **SmartSnippets Studio > SmartSnippets Toolbox** (Figure 38). Select **JTAG** (above), JTAG adapter (middle panel), a project name (left panel), and chip version (DA14585, right panel). If [SmartSnippets Toolbox](#) is being run for the first time, first define a new project by the **New** button.



Figure 38: Initial Window to Choose Device and Connection Type

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- Click the **Open** button.
- From the **Tools > Board Setup** (Figure 39), make sure that the correct board settings (Figure 40) are set.



Figure 39: Opening SmartSnippets Board Setup

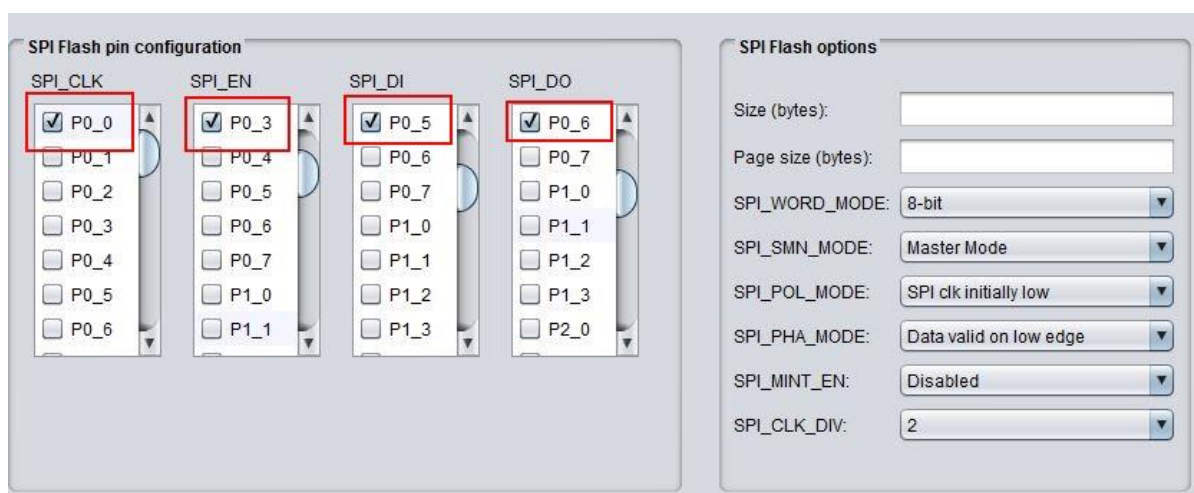



Figure 40: Smart Snippets Board Setup Window

- Click the  button on the left to open the **SPI Flash Programmer** (see Figure 41):

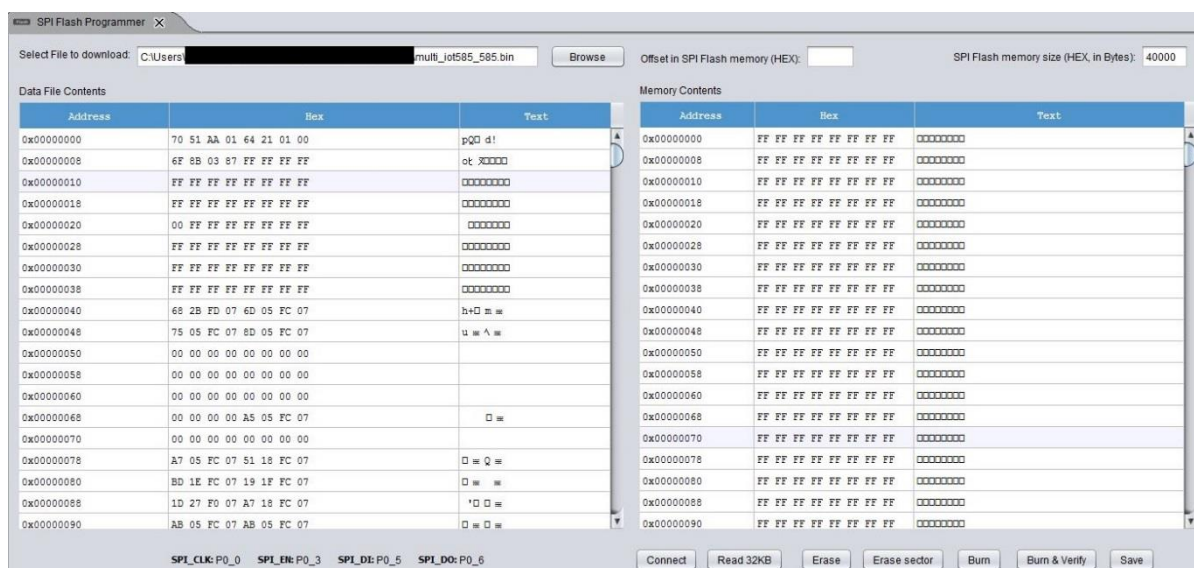


Figure 41: SPI Flash Programmer

- From **Select File to download**, browse for the `<multi/whole_app>.bin` file.
- Insert address `0x00` in the **Offset in SPI Flash memory (HEX)** field and follow the steps:

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- Press **Connect** and wait for confirmation from the Log window;
- Press **Erase** and wait for confirmation from the Log window;
- Press **Burn & Verify** and wait for confirmation from the Log window.

IMPORTANT NOTES

- When burning Flash memory with [SmartSnippets Studio](#), click **NO** when the "Do you want SPI Flash memory to be bootable?" window appears.
- In [SmartSnippets Studio](#), type (for example) "30000" instead of "0x30000".

- The image is now burnt in flash and by pressing the Reset button on the CIB board, it will start working with the programmed application.

D.2.2 Preparing the Various .img and .bin Files Manually

The following files ([Table 93](#)) are needed to program the Flash memory so that an MSK application starts on a button or hard reset.


Table 93: Files Needed or Created During Flash Programming

File Name	Location	Description
<application>.hex	..\..\Keil_5\out_585	.hex file generated by Keil project
<application>.bin	mkimage folder	.bin file generated from .hex file
<sw_version>.h	..\..\sdk\platform\include	.h software version file needed to create .img
<file>.img	mkimage folder	.img file created from the .bin file and the software version file
device_config_all_beacons.txt	mkimage_utils_scripts folder	Contains the format of the device config struct
<device_config_all_beacons>.bin	mkimage_utils_scripts folder	.bin file created by SmartSnippets Toolbox containing the beacon configuration
<dev_conf_struct_default>.cfg	mkimage_utils_scripts folder	.cfg file containing a default configuration struct preceded by its header
user_config_sw_ver.h	sw_version folder	.h beacon SW version file used to create the device configuration field (struct and header)

In order to prepare the necessary files, follow the steps below:

- Open the `mkimage` app folder and open a cmd window (Shift + Left click > Open command window here).
- If there already is a .bin file of the Keil project, go directly to step 5.
- Copy and paste the .hex file generated by your Keil project in the `mkimage` folder.
- Create a .bin file from the .hex file (using the `hex2bin.exe` utility).
- Create two different .img files from alternative .bin files containing different images. Typing "mkimage" in the command console shows instructions on the syntax needed to create an .img file. Example: `mkimage single <generated_bin_file>.bin <version_file>.h <desired_name>.img`.
- If a `<dev_conf_struct_default>.bin` file is already provided in `mkimage` folder, skip to step 6.g.

Open [SmartSnippets Studio](#) > **SmartSnippets Toolbox**. The screen show in [Figure 38](#) appears.

 - Choose option **JTAG** and click button **Open**.
 - In the next screen, press button  on the left to open the Proprietary Header Programmer.

The `mkimage` folder should contain a `device_config_all_beacons.txt` file (see [Figure 42](#)).

```

1 16 String uuid uuid
2 2 Integer major major or ALT_val1
3 2 Integer minor minor or ALT_val2
4 2 Integer company_id company_id
5 2 Integer adv_int advertising interval
6 1 Integer power tx power
7 1 Integer beacon_type iBeacon/Alt/Eddystone (UUID/URL)
8 1 Integer url_prefix url_prefix
9 19 String url (first byte = Length)full url with extension
10 1 Integer TLM_version TLM version
11 1 Integer TLM_used TLM_used

```

Figure 42: Device Config Struct Format in .txt File

- c. Browse to the file `device_config_all_beacons.txt` and open it for editing.
- d. Enter the desired values at the corresponding blanks (see [Figure 43](#) and [Figure 44](#)):

Memory offset (HEX):

File:

Memory Type: ☐ EEPROM ☒ SPI Flash

Address	Size (bytes)	Type	Parameter	Description	Value
0x1F000	16	String	uuid	uuid	<input type="text" value="1"/>
0x1F010	2	Integer	major	major or ALT_val1	<input type="text"/>
0x1F012	2	Integer	minor	minor or ALT_val2	<input type="text"/>
0x1F014	2	Integer	company_id	company_id	<input type="text"/>
0x1F016	2	Integer	adv_int	advertising interval	<input type="text"/>
0x1F018	1	Integer	power	tx power	<input type="text"/>
0x1F019	1	Integer	beacon_type	iBeacon/Alt/Eddystone (UUID/URL)	<input type="text"/>
0x1F01A	1	Integer	url_prefix	url_prefix	<input type="text"/>
0x1F01B	19	String	url	(first byte = Length)full url with extension	<input type="text"/>
0x1F02E	1	Integer	TLM_version	TLM version	<input type="text"/>
0x1F02F	1	Integer	TLM_used	TLM_used	<input type="text"/>

Figure 43: Programming the Various Fields of the Device Configuration Struct

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

Memory offset (HEX):

File:

Memory Type: ☐ EEPROM ☒ SPI Flash

Address	Size (bytes)	Type	Parameter	Description	Value
0x1F000	16	String	uuid	uuid	2380C52068D211E498030800200C9A66
0x1F010	2	Integer	major	major or ALT_val1	0003
0x1F012	2	Integer	minor	minor or ALT_val2	0004
0x1F014	2	Integer	company_id	company_id	4C00
0x1F016	2	Integer	adv_int	advertising interval	E803
0x1F018	1	Integer	power	tx power	C5
0x1F019	1	Integer	beacon_type	iBeacon/Alt/Eddystone(UUID/URL)	00
0x1F01A	1	Integer	url_prefix	url_prefix	00
0x1F01B	19	String	url	(first byte = Length)full url with extension	000000000000000000000000E064696173656D6907
0x1F02E	1	Integer	TLM_version	TLM version	00
0x1F02F	1	Integer	TLM_used	TLM_used	00

Figure 44: Creating a Custom Dev_Conf_Struct .bin File

NOTE

Addresses and values are random.

- e. Click the **Export** button at the bottom of the screen to save the .bin file.
- f. To edit the generated .bin files:
 - i. Press button to open the Proprietary Header Programmer.
 - ii. Click the **Import** button at the bottom of the screen and browse for <filename>.bin. The **Value** column will now be populated with the programmed values.
 - iii. Modify the values as required.
 - iv. Click button **Export** to save the .bin file.
- g. To create the device configuration file to be burned into Flash memory, use the `mkimage` application in `cfg` mode (see Appendix C.2.5). This file includes the device configuration struct preceded by the device configuration struct header.
7. Use "mkimage multi" mode for Smart Tag and IoT sensors applications and "mkimage whole" mode for beacon applications to create the images to be burned in flash as described in Appendix D.2.1.

Appendix E Using the SUOTA Application for Android

This appendix describes how to use Dialog's SUOTA application to update the software programmed in the DA14585 MSK HW reference design. There are two variants of the SUOTA application: one for the Android operating system and one for the iOS. Users can find the SUOTA application by searching for 'Dialog SUOTA' in Google Play Store or Apple App Store. Since these applications have a similar user interface, only the application for Android operating system is described here.

STEP 1: Prepare the MSK Device

If your device is already programmed, skip this step and proceed with [STEP 2](#).

A dual image secondary boot loader needs to be programmed into the OTP memory of the DA14585. Moreover, the initial software image with the right header needs to be programmed into the DA14585 IoT MSK SPI Flash memory using the Flash Programmer (see [Appendix D](#)).

To verify that this step has been completed successfully, reset the device and check the expected behavior for the desired application:

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

- Green LED blinking for Smart Tag
- Yellow LED blinking for IoT sensors
- Beacon advertising for beacon examples

STEP 2: Install and Start the Application on the Android Device

After successful installation, the SUOTA icon appears under the installed applications menu. Click on the icon ([Figure 45](#)) to start the application.



Figure 45: SUOTA App Icon

STEP 3: Initial Menu, Scan for Advertising Devices

Assuming the MSK device advertises a SUOTA advertising string once the SUOTA APP is opened, the Bluetooth Device address and device name is displayed as shown in [Figure 46](#).

To re-initiate scanning, just press the “SCAN” button in the top right corner.

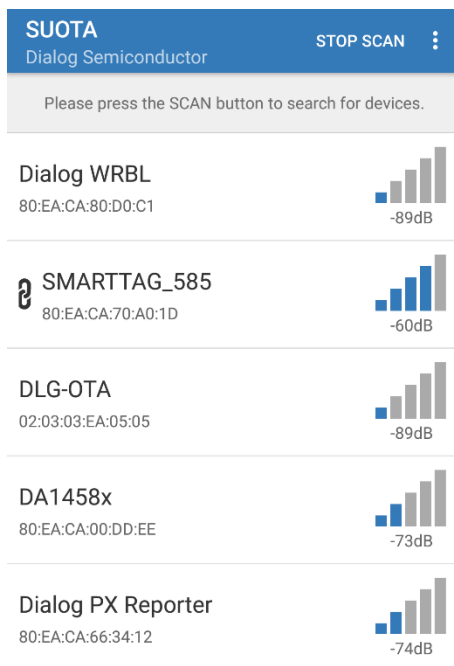


Figure 46: Device Selection Menu

STEP 4: Connect to the Smart Tag Device

Click on the desired device to connect. Once the APP is connected to the device successfully, the DISS information is displayed on the screen as shown in [Figure 47](#).

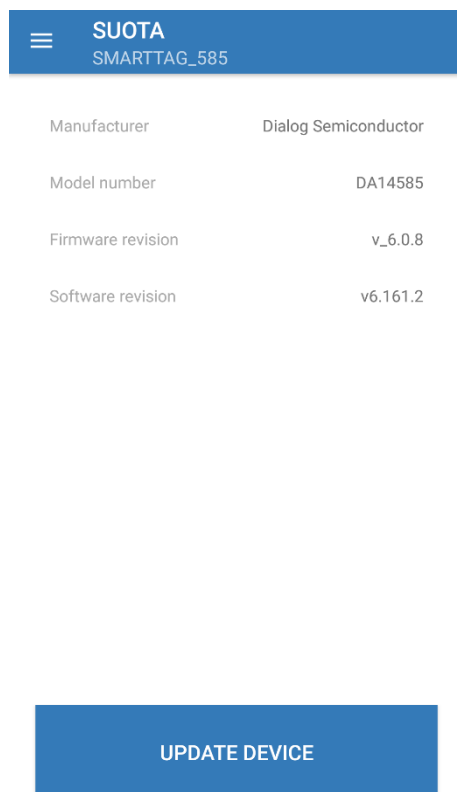


Figure 47: DIS Screen

STEP 5: Update SmartTag Software Image

Click on the **Update device** button and a list of files will appear on the screen. For the file to appear in this **File selection** screen (Figure 48), it has to be copied to the SUOTA directory of the Android device. Connect the Android device via USB to the PC where the images are created and copy the images under the SUOTA directory.

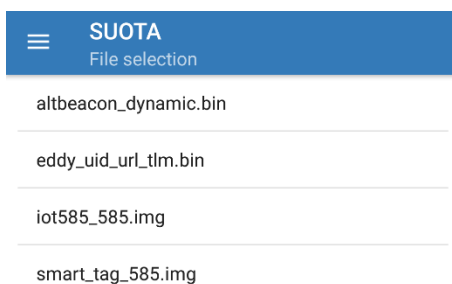
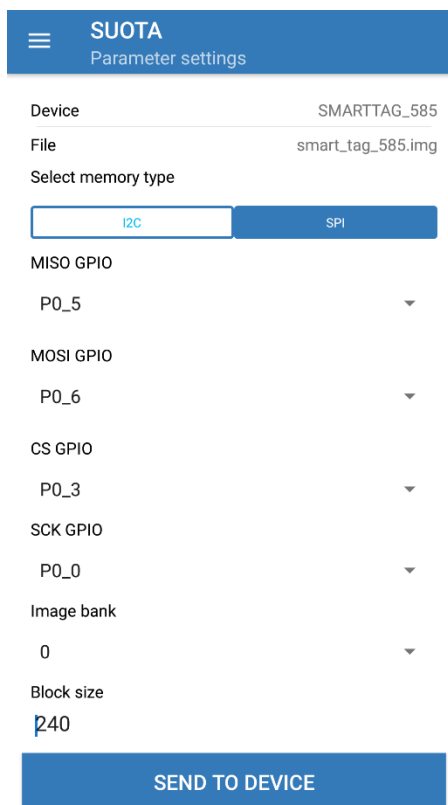


Figure 48: File Selection Screen

STEP 6: Set SUOTA Parameters

Once an image is selected, the **Parameter settings** (Figure 49) screen appears.



The screenshot shows the 'SUOTA Parameter settings' screen. At the top, there is a blue header with a hamburger menu icon, the title 'SUOTA', and the subtitle 'Parameter settings'. Below this, the settings are as follows:

- Device:** SMARTTAG_585
- File:** smart_tag_585.img
- Select memory type:** Two buttons, 'I2C' (highlighted in light blue) and 'SPI' (dark blue).
- MISO GPIO:** A dropdown menu showing 'P0_5'.
- MOSI GPIO:** A dropdown menu showing 'P0_6'.
- CS GPIO:** A dropdown menu showing 'P0_3'.
- SCK GPIO:** A dropdown menu showing 'P0_0'.
- Image bank:** A dropdown menu showing '0'.
- Block size:** A text input field containing '240'.

At the bottom of the screen is a large blue button labeled 'SEND TO DEVICE'.

Figure 49: SUOTA Parameter Settings

First set the memory type. The image update procedure is only supported for non-volatile memory types of SPI (Flash memory) and I2C (EEPROM). In this example SPI (Flash) is selected.

Then select the **Image (memory) bank** (Figure 49):

- 1: Use the first bank with start address indicated in the Product Header.
- 2: Use the second bank with start address indicated in the Product Header.
- 0: Burn the image into the bank that holds the oldest image.

Next, define the GPIO pins of the memory device. In the SmartTag device, the SPI Flash GPIO configuration is as follows:

- MISO P0_5
- MOSI P0_6
- CS P0_3
- SCK P0_0

Finally, scroll down to choose the block size. Take the following points into account when setting the block size.

- The block size must be larger than 64 bytes, which is the size of the image header.
- The block size must be a multiple of 20 bytes, which is the maximum amount of data that can be written at once in the `SPOTA_PATCH_DATA` characteristic.
- The block size must not be larger than the SRAM buffer in the SUOTA Receiver implementation, which holds the image data received over the BLE link before burning it into the non-volatile memory.

DA14585 IoT Multi Sensor Development Kit Developer's Guide

Company Confidential

This example uses a block size of 240 bytes.

After all the parameters are set, users can click on the **Send to device** button at the bottom of the screen.

STEP 7: Reboot the Device

Once the **Send to device** button is clicked, a load screen appears that shows the image data blocks being sent over the BLE link (Figure 50). In case an error occurs, a pop-up indication will inform the user.

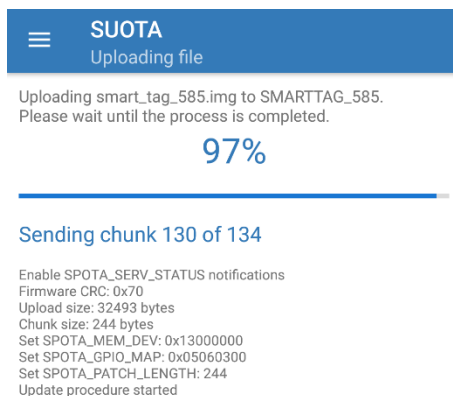


Figure 50: SUOTA Uploading Screen

When no error occurs and the SmartTag device has received and programmed the image successfully, the screen in Figure 51 will appear, asking the user to reboot the SmartTag device.

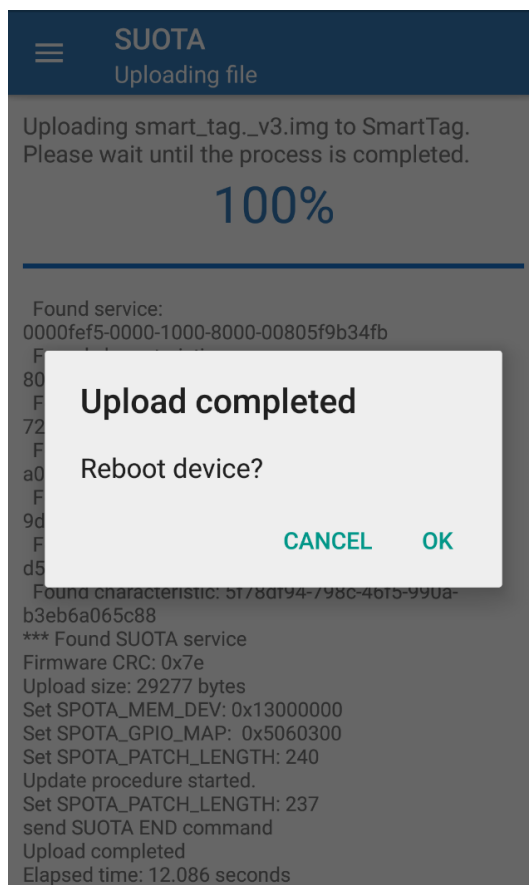


Figure 51: Successful Update Screen

STEP 8: Verify that the New Software is Running on the SmartTag Device

Repeat [STEP 3](#) and [STEP 4](#) to verify that the DIS screen shows the firmware and software version of the new software.

To re-initiate scanning just press the "**SCAN**" button in the top right corner.

Revision History

Revision	Date	Description
1.2	17-Jan-2022	Updated logo, disclaimer, copyright.
1.1	05-Feb-2019	Updates for 6.160.4
Change details: For changes: <ul style="list-style-type: none">● Section 4.2.4<ul style="list-style-type: none">○ Added this section to include details of how to compile including Bosch BSEC Library.		
1.0	03-Aug-2018	Initial version.

Status Definitions

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