# Dialog SDK 5.0.x/6.0.x Tutorial

## Pairing, Bonding and Security

2017 March

…personal
…portable
…connected

# BLE Security

## Let’s build a demo together …

- **Before we start, we recommend you to …**
  - Install the latest Smartsnippets studio from Dialog customer support website
  - Download the SDK as well
  - Link:
    - https://support.dialog-semiconductor.com/connectivity
  - Require to look at Dialog hands on tutorial 1, 2, 3 and 4

- **Consideration …**
  - All the changes are applicable in both the SDK 5.0.x (DA14580/1/2/3) and SDK 6.0.x (DA14585/6) if it is not mentioned specifically for a particular application

  - BLE 4.1 spec is supported by DA14580/1/2/3

  - BLE 4.2 and 5.0 spec is supported by DA14585/6

# BLE Security

## Let’s build a demo together …

- **What are you going to learn from this tutorial …**
  - Basic understanding of BLE Security
  - What is Pairing? What is Bonding?
  - ‘Just-Works’ pairing
  - Single-device bonding
  - Basic understanding of multi-device bonding
  - Small assignment to add pairing in the custom service database

# Contents

**BLE security**

Source code discussion WRT BLE security

What would you see as output

# BLE security

## Overview

- Protection of private information of a user is important for every wireless low energy device, from fitness band to payment systems. Privacy mechanisms prevent devices from being tracked by untrusted devices. Secure communications keep data safe while also preventing unauthorized devices from injecting data to trigger unintended operation of the system.

- In **Bluetooth Low Energy (BLE)** devices connected in a link can pass sensitive data by setting up a secure encrypted link.

- In BLE the confidential payload includes a **Message Identification Code (MIC)** that is encrypted with the data.

- In BLE the secure link is more vulnerable to passive eavesdropping, however because of the short transmission periods this vulnerability is considered a low risk.

- Therefore data encryption is used to prevent passive and active **Man-In-The-Middle (MITM)** – eavesdropping attacks on a Bluetooth low energy link.

# BLE security

## Overview

- Encryption is the means to make the data unreadable to all but the Bluetooth master and slave devices forming a link.

- Because eavesdropping attacks are directed on the over-the-air transmissions between the BLE devices, so data encryption is accomplished prior to transmission using a shared, secret key.

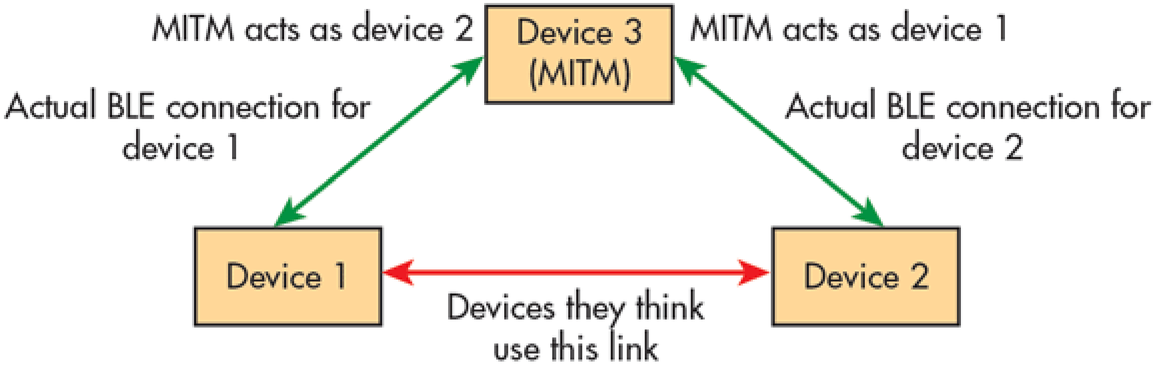
- **Common attacks on wireless communication protocols:**
  - **Man-in-the-Middle (MITM)**
  - **Passive Eavesdropping**
  - **Privacy or Identity tracking**

- To protect communications from unauthorized access, wireless systems must prevent passive eavesdropping and man-in-the-middle (MITM) attacks.

# BLE security

## Overview – MITM

- MITM is an attacking method where as two devices try to communicate with each other, a third ghost device inserts in the communication model between the actual two devices and emulates a behavior to both actual devices that those devices directly communicating to each other. This is also known as active eavesdropping.



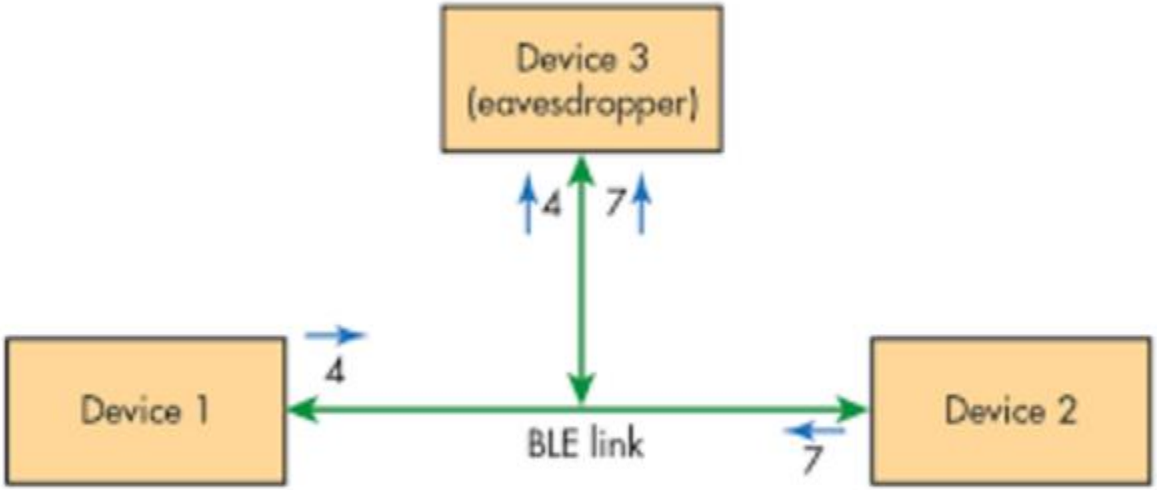
- Authentication protects against MITM by ensuring that the device is communicating with actually the intended device and not an unauthorized ghost device emulating as the intended one.

# BLE security

## Overview – Passive Eavesdropping

- Passive eavesdropping is a third device silently listens to the private communication between two devices.

- Protection against this security hole is important in applications such as payment system where the confidentiality of information is very import.



# BLE security

## Overview – Passive Eavesdropping

- Systems can protect against passive eavesdropping by using a key to encrypt data.

- LE Secure Connections, introduced in BLE 4.2 uses the **Federal Information Processing Standard (FIPS)** compliant **Elliptic Curve Diffie-Hellman (ECDH)** algorithm. It generates **DHKey (Diffe-Hellman Key)** as well which is never shared over the air.

- This DHKey key is used to generate other keys such as **Long Term Keys (LTK)**.

- As the DHKey is never exchanged over the air, it becomes very difficult for a third device to guess the encryption key.

- Earlier versions of BLE (Bluetooth 4.1 or older) devices used easy-to-guess **Temporary Keys (TK)** to encrypt the link for the first time. **Long Term Keys (LTK)** along with other keys, were then exchanged between devices over this encrypted but potentially compromised link.

# BLE security

## Overview – Privacy or Device Identity Tracking

- BLE supports the privacy feature that reduces the ability to track an LE device over a period of time by changing the Bluetooth device address frequently. The frequently changing address is called the **Resolvable Private Address** (RPA) and only the trusted devices can resolve it.

# BLE security

## Overview – BLE solutions to protect device attacks

- In Bluetooth, an **association model** is a mechanism that two devices use to **authenticate** each other and then securely exchange data. These are used to remove the risk of BLE device attacks called MITM and passive eavesdropping.

- **Which model to use -** Ask the designed system**:**

  - Input/Output capabilities of the devices: Does the device receive data from a user (such as a keyboard) or output data to the user (such as an LCD with 6 digit number display capability)?

  - Requirement of MITM protection

  - OOB data availability: Does the device communicate with other devices using Out-of-Band (OOB)? For example, if part of the security key can be transferred between the two devices over Near-Field Communication (NFC), an eavesdropper will not be able to make sense of the final data.

# BLE security

## Overview – BLE solutions to protect device attacks

- **There are two variants of the privacy feature to resolve identity tracking attack:**
  - First variant: Private addresses are resolved and generated by the Host. This is used in the before **4.2 Bluetooth stacks**.
  - Second variant: Private addresses are resolved and generated by the Controller without involving the Host after the Host provides the controller device identity information. This is used by Bluetooth 4.2 compliant devices.

# BLE security

## Overview – BLE Association models

- Four association models are available in Bluetooth 4.2 for Bluetooth Low Energy:
  - Numeric Comparison
  - Just Works
  - Passkey Entry
  - Out of Band (OOB)

# BLE security

## Overview – BLE Association models

- **Four association models are available in Bluetooth 4.2 for Bluetooth Low Energy**:

  - **Numeric Comparison –**

    In this model both devices display a six-digit number and the user authenticates by selecting YES if both devices are displaying the same number.

  - **Just Works –**

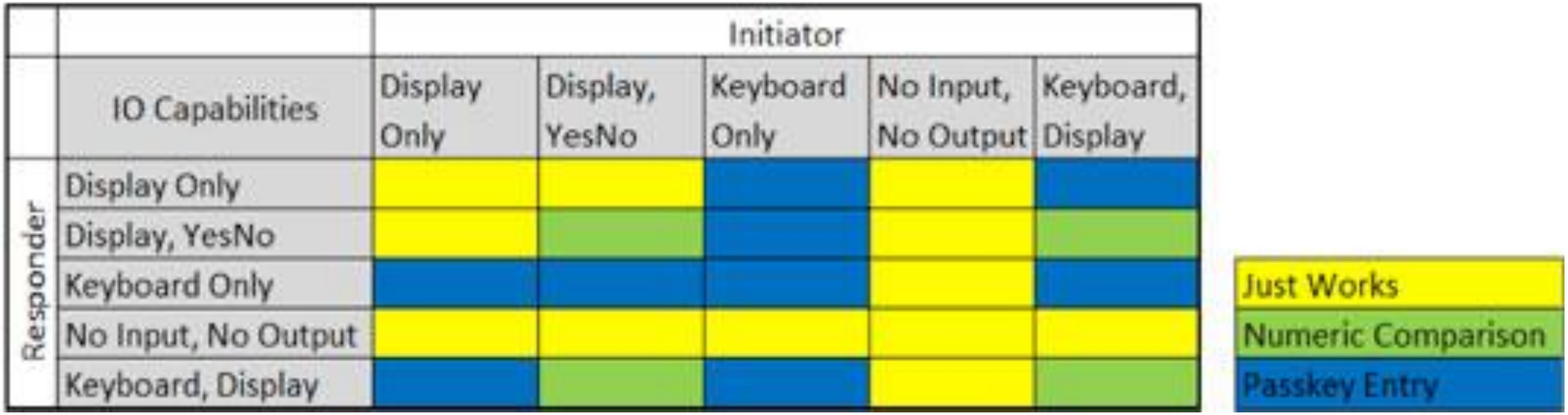
    This model is used when either MITM protection is not needed or devices have I/O capabilities shown in the page 15. The Just Works association model follows the same steps as mentioned in Numeric Comparison. However, a six-digit number is not generated or displayed.

# BLE security

## Overview – BLE Association models

- **Passkey Entry –**

  The user either inputs an identical passkey into both devices, or one device displays the passkey and the user enters that passkey into the other device. Exchange of the passkey one bit at a time in Bluetooth 4.2 is an important enhancement over the legacy passkey entry model (Bluetooth 4.1 or older) where the whole passkey is exchanged in a single confirm operation. This has enhanced the passkey exchange mechanism and now it is very difficult to guess the passkey in 4.2.



| | | Initiator | | | | |
|---|---|---|---|---|---|---|
| | IO Capabilities | Display Only | Display, YesNo | Keyboard Only | No Input, No Output | Keyboard, Display |
| Responder | Display Only | 🟨 | 🟨 | 🟦 | 🟨 | 🟦 |
| | Display, YesNo | 🟨 | 🟩 | 🟦 | 🟨 | 🟩 |
| | Keyboard Only | 🟦 | 🟦 | 🟦 | 🟨 | 🟦 |
| | No Input, No Output | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 |
| | Keyboard, Display | 🟦 | 🟩 | 🟦 | 🟨 | 🟩 |

| | |
|---|---|
| 🟨 | Just Works |
| 🟩 | Numeric Comparison |
| 🟦 | Passkey Entry |

# BLE security

## Overview – BLE Association models

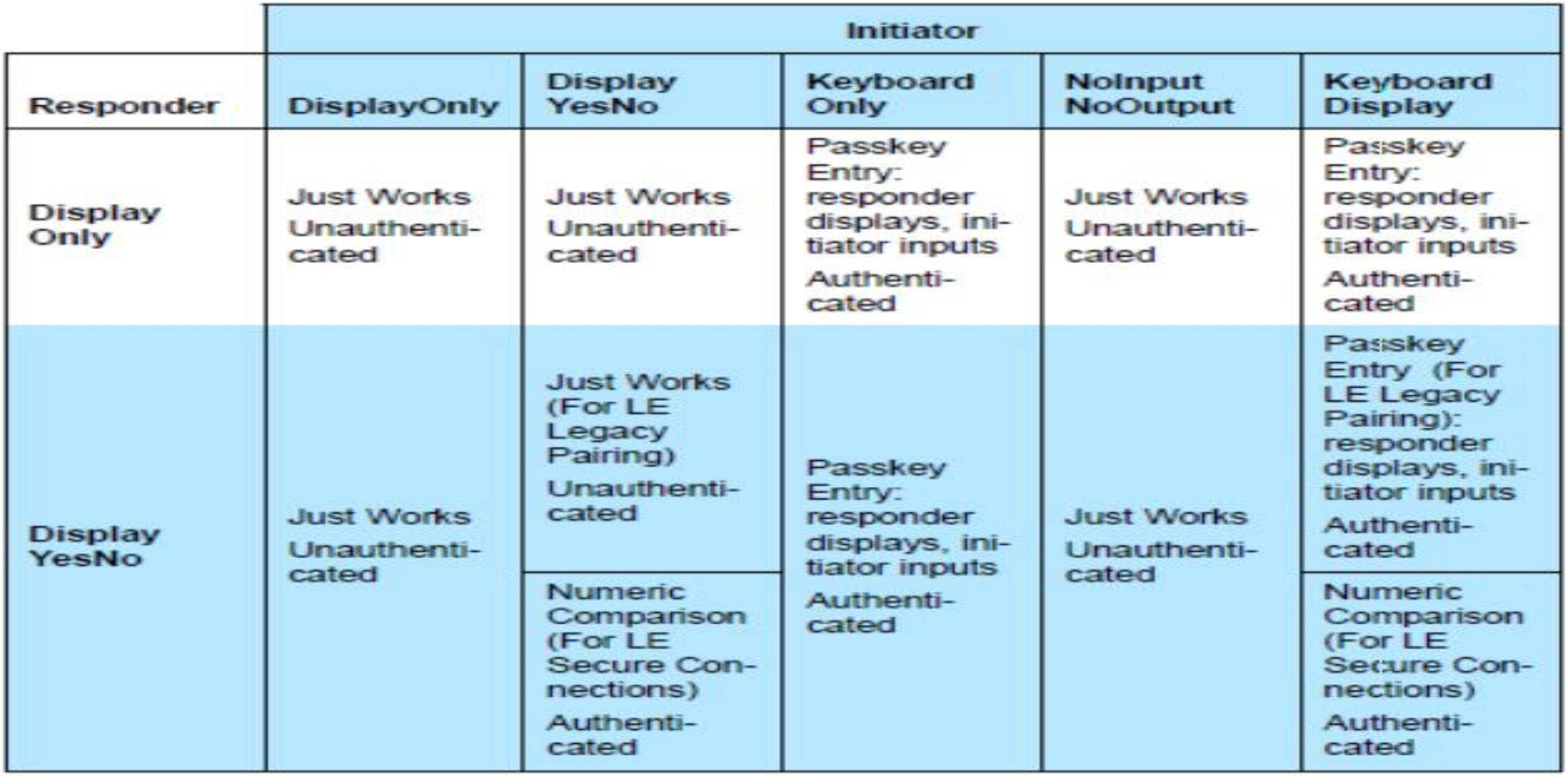
- **Out Of Band OOB –**

The OOB association model is the model to use if at least one device with OOB capability already has cryptographic information exchanged out of band. Here, protection against MITM depends on the MITM resistance of the OOB protocol used for sharing the information.

# BLE security

## Overview – Association model

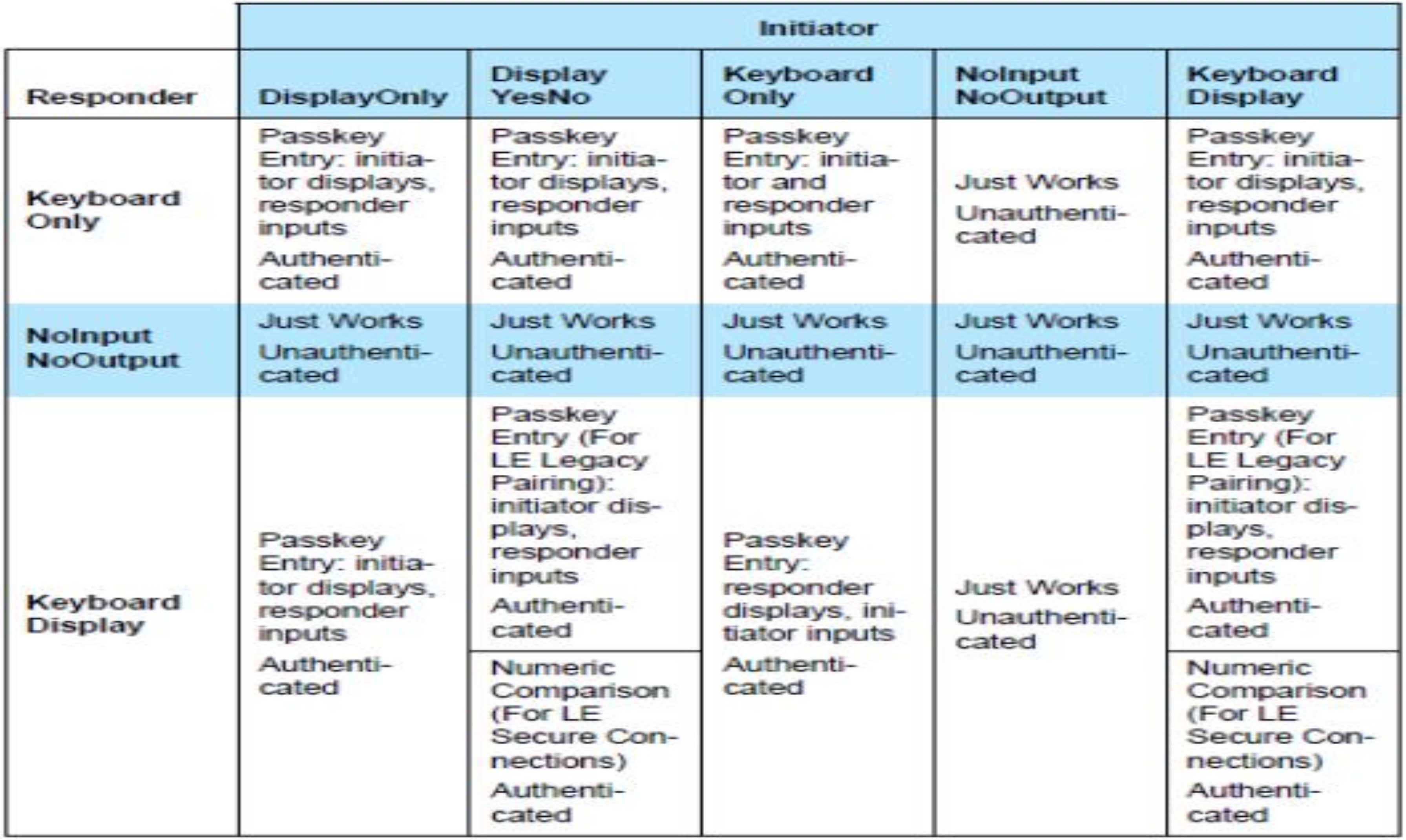
- The use of each association model is based on the I/O capabilities of the devices. The best pairing method can be chosen based on the following table:



| | **Initiator** | | | | |
|---|---|---|---|---|---|
| **Responder** | **DisplayOnly** | **Display YesNo** | **Keyboard Only** | **NoInput NoOutput** | **Keyboard Display** |
| **Display Only** | Just Works<br>Unauthenticated | Just Works<br>Unauthenticated | Passkey Entry: responder displays, initiator inputs<br>Authenticated | Just Works<br>Unauthenticated | Passkey Entry: responder displays, initiator inputs<br>Authenticated |
| **Display YesNo** | Just Works<br>Unauthenticated | Just Works (For LE Legacy Pairing)<br>Unauthenticated<br>Numeric Comparison (For LE Secure Connections)<br>Authenticated | Passkey Entry: responder displays, initiator inputs<br>Authenticated | Just Works<br>Unauthenticated | Passkey Entry (For LE Legacy Pairing): responder displays, initiator inputs<br>Authenticated<br>Numeric Comparison (For LE Secure Connections)<br>Authenticated |

# BLE security

## Overview – Association model



| | Initiator | | | | |
|---|---|---|---|---|---|
| **Responder** | **DisplayOnly** | **Display YesNo** | **Keyboard Only** | **NoInput NoOutput** | **Keyboard Display** |
| **Keyboard Only** | Passkey Entry: initiator displays, responder inputs<br>Authenticated | Passkey Entry: initiator displays, responder inputs<br>Authenticated | Passkey Entry: initiator and responder inputs<br>Authenticated | Just Works<br>Unauthenticated | Passkey Entry: initiator displays, responder inputs<br>Authenticated |
| **NoInput NoOutput** | Just Works<br>Unauthenticated | Just Works<br>Unauthenticated | Just Works<br>Unauthenticated | Just Works<br>Unauthenticated | Just Works<br>Unauthenticated |
| **Keyboard Display** | Passkey Entry: initiator displays, responder inputs<br>Authenticated | Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs<br>Authenticated<br>Numeric Comparison (For LE Secure Connections)<br>Authenticated | Passkey Entry: responder displays, initiator inputs<br>Authenticated | Just Works<br>Unauthenticated | Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs<br>Authenticated<br>Numeric Comparison (For LE Secure Connections)<br>Authenticated |

# BLE security

## Overview – Pairing and bonding to resolve attacking issue

- **Pairing** is the process of key exchange and authentication.
- **Bonding** means storing a set of secure device information in the memory. When the same peripheral device will try to connect to the peer device then peripheral device will need not to go through the pairing process again as long as the secure information is stored in the peer device.
- There are two types of paring base on BLE version:
  - LE Legacy Pairing (supported in Bluetooth 4.0 and 4.1)
  - LE Secure Connections (introduced in Bluetooth 4.2)
- Before going further ahead we need to understand a few terms used in pairing and authentication.

# BLE security

## Overview – Pairing and bonding to resolve attacking issue

- A BLE device uses a **shared secret key** with the trusted peer device. This key is known as **Identity Resolving Key** (IRK).

- **IRK** is used to **generate and resolve** an **RPA**.

- **IRK** is shared with peer devices during the time of **pairing** process between a BLE peripheral and a peer master device.

- The private address RPA is generated using the devices IRK exchanged during the previous pairing/bonding procedure.

- Depending on the application requirement and the capability of the devices, Bluetooth has several options for pairing.

# BLE security

## Overview – Pairing and bonding to resolve attacking issue

- In version 4.0 and 4.1 of the core specification, BLE functionality uses **the secure simple pairing model** (now known as **LE Legacy**), in which devices choose one method from **Just Works, Passkey Entry and Out Of Band (OOB)** based on the input/output capability of the devices.

- In version 4.2, security is enhanced by **the new LE secure connections pairing model**. In this model, the numeric comparison is added to the LE Legacy methods and the **Elliptical Curve Diffie-Hellman (ECDH)** algorithm is introduced for key exchange in this process.

- If you use **LE legacy** pairing **Just Works and Passkey Entry** do not provide any passive eavesdropping protection.

# BLE security

## Overview – Pairing

- A BLE device that wants to share secure data with another device must first pair with that device. The **Security Manager Protocol (SMP)** carries out the pairing in **three steps**:

  - The two connected BLE devices announce their input and output capabilities based on association model and from that information the BLE stack determine a suitable method for step 2.

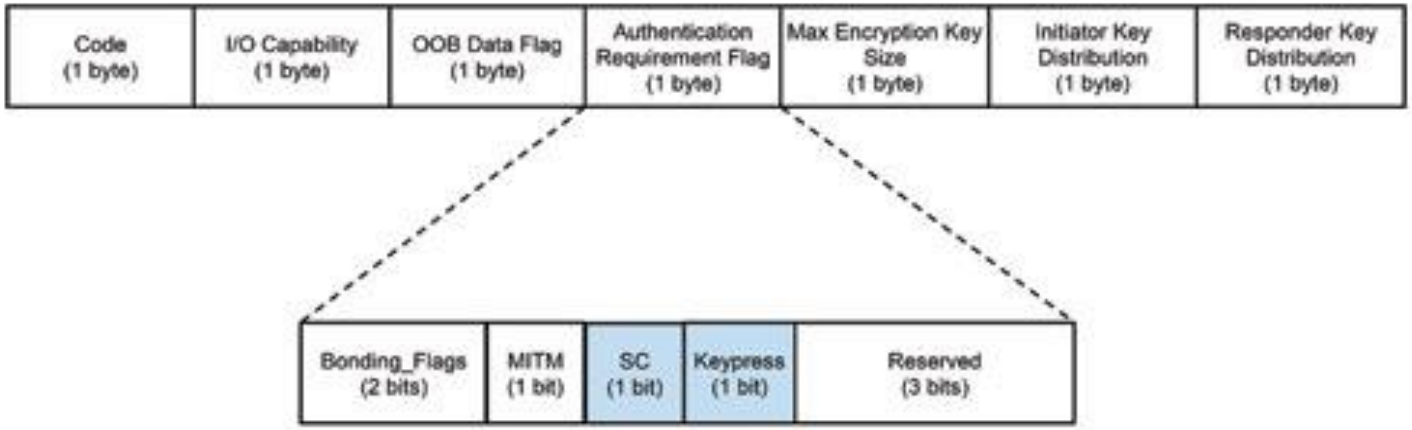
  - The purpose of this step 2 is to generate the **Short Term Key (STK)** used in the third step to secure key distribution. The devices agree on a **Temporary Key (TK)** that along with some random numbers creates the **STK**.

  - In this step 3 each device may distribute to the other device up to **three keys**:
    - The **Long Term Key (LTK)** used for Link Layer encryption and authentication,
    - The **Connection Signature Resolving Key (CSRK)** used for data signing at the ATT layer, and
    - The **Identity Resolving Key (IRK)** used to generate a private address.

# BLE security

## Overview – Pairing

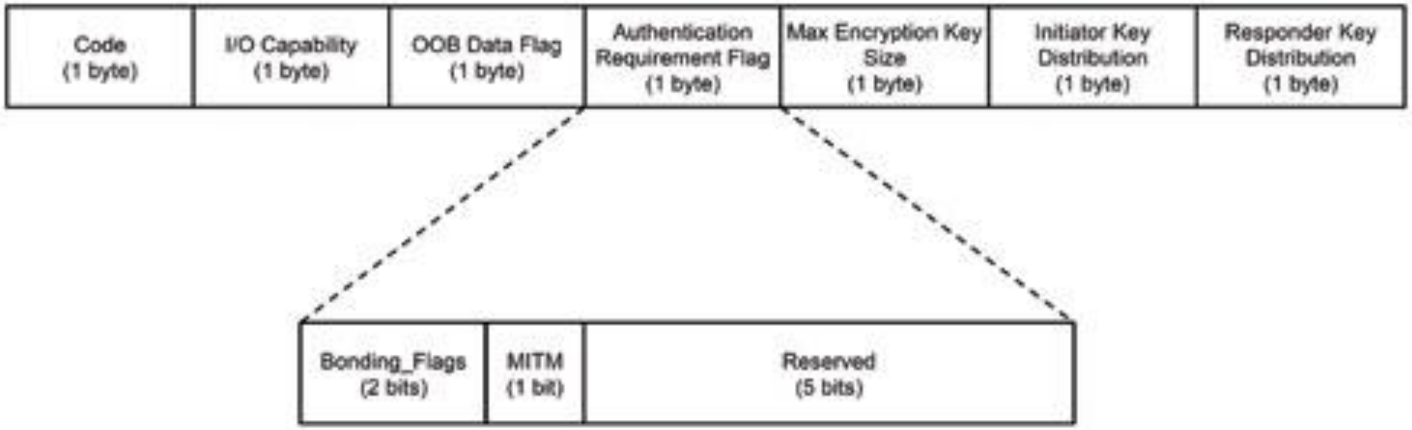
Pairing Parameters in Bluetooth 4.2



| Code (1 byte) | I/O Capability (1 byte) | OOB Data Flag (1 byte) | Authentication Requirement Flag (1 byte) | Max Encryption Key Size (1 byte) | Initiator Key Distribution (1 byte) | Responder Key Distribution (1 byte) |
|---|---|---|---|---|---|---|

| Bonding_Flags (2 bits) | MITM (1 bit) | SC (1 bit) | Keypress (1 bit) | Reserved (3 bits) |
|---|---|---|---|---|

Pairing Parameters in Bluetooth 4.1



| Code (1 byte) | I/O Capability (1 byte) | OOB Data Flag (1 byte) | Authentication Requirement Flag (1 byte) | Max Encryption Key Size (1 byte) | Initiator Key Distribution (1 byte) | Responder Key Distribution (1 byte) |
|---|---|---|---|---|---|---|

| Bonding_Flags (2 bits) | MITM (1 bit) | Reserved (5 bits) |
|---|---|---|

# BLE security

## Overview – Pairing

- A **Pairing Request message** is transmitted from the initiator containing the IO capabilities, authentication data availability, authentication requirements, key size requirements, and other data.

- A **Pairing Response message** is transmitted from the responder and contains much of the same information as the initiators Pairing Request message thus confirming that a pairing is successfully negotiated.

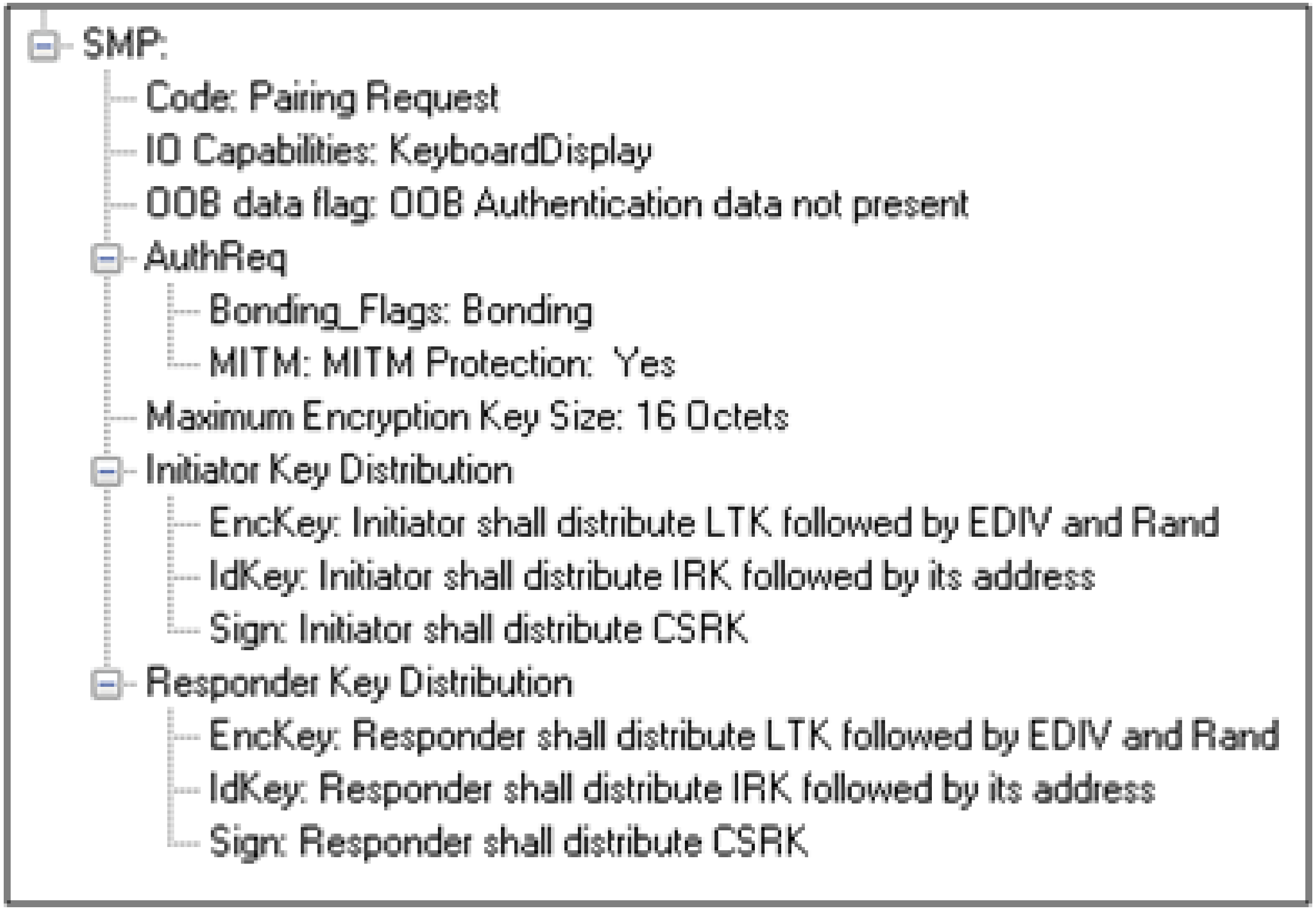
- Sharing a sample SMP decode in the next slide page, please note the **key** identified.

- Creating a shared, secret key is an evolutionary process that involves several intermediary keys.

Dialog Semiconductor © 2017

# BLE security

## Overview – Security Manager Protocol



- SMP:
  - Code: Pairing Request
  - IO Capabilities: KeyboardDisplay
  - OOB data flag: OOB Authentication data not present
  - AuthReq
    - Bonding_Flags: Bonding
    - MITM: MITM Protection:  Yes
  - Maximum Encryption Key Size: 16 Octets
  - Initiator Key Distribution
    - EncKey: Initiator shall distribute LTK followed by EDIV and Rand
    - IdKey: Initiator shall distribute IRK followed by its address
    - Sign: Initiator shall distribute CSRK
  - Responder Key Distribution
    - EncKey: Responder shall distribute LTK followed by EDIV and Rand
    - IdKey: Responder shall distribute IRK followed by its address
    - Sign: Responder shall distribute CSRK

# BLE security

## Overview – Pairing

- The resulting several intermediary keys include:
  - **IRK**: 128-bit key used to generate and resolve random address.
  - **CSRK**: 128-bit key used to sign data and verify signatures on the receiving device.
  - **LTK**: 128-bit key used to generate the session key for an encrypted connection.
  - **Encrypted Diversifier (EDIV)**: 16-bit stored value used to identify the LTK. A new EDIV is generated each time a new LTK is distributed.
  - **Random Number (RAND)**: 64-bit stored value used to identify the LTK. A new RAND is generated each time a unique LTK is distributed.
  - Note that, particular importance to decrypting the encrypted data on a BLE link is LTK, EDIV, and RAND.

# BLE security

## Overview – Pairing

- Note that, IRK and CSRK are passed in an encrypted link along with LTK and EDIV.
- The use of the IRK and CSRK attempt to place an identity on devices operating in a piconet. The probability that two devices will have the same IRK and generate the same random number is low.

- **IRK:**

**BLE** has a feature that reduces the ability of an attacker to track a device over a long period buy by frequently and randomly changing an advertising device’s address. This is the privacy feature. This feature is not used in the discovery mode and procedures but is used in the connection mode and procedures.

If the advertising device was previously discovered and has returned to an advertising state, the device must be identifiable by trusted devices in future connections without going through discovery procedure again. The IRK stored in the trusted device will overcome the problem of maintaining privacy while saving discovery computational load and connection time. The advertising devices **IRK** was passed to the master device during initial bonding. The a master device will use the IRK to identify the advertiser as a trusted device.

# BLE security

## Overview – Pairing

- **CSRK:**

BLE supports the ability to authenticate data sent over an unencrypted ATT bearer between two devices in a trust relationship. If authenticated pairing has occurred and encryption is not required (security mode 2) data signing is used if CSRK has been exchanged. The sending device attaches a digital signature after the data in the packet that includes a counter and a message authentication code (MAC). The key used to generate MAC is CSRK. Each peer device in a Piconet will have a unique CSRK.

The receiving device will authenticate the message from the trusted sending device using the CSRK exchanged from the sending device. The counter is initialized to zero when the CSRK is generated and is incremented with each message signed with a given CSRK. The combination of the CSRK and counter mitigates replay attacks.

# BLE security

## Overview – Anatomy of Pairing Methods

- The two devices in the link use the IO capabilities from Pairing Request and Pairing Response packet data to determine which of two pairing methods to use for generation of the **Temporary Key (TK)**.

- The two methods are **Just Works** and **Passkey Entry**.

- Example when **Just Works** method is appropriate is when the IO capability input = None and output = None.

- Example when **Passkey Entry** would be appropriate would be if input= Keyboard and output = Display.

# BLE security

## Overview – Anatomy of Pairing Methods

- In **Just Works** the TK = 0.

- In the **Passkey Entry** method

  TK = {

  6 numeric digits, Input = Keyboard

  6 random digits, Input = Display

  }

- **Mechanism:**

  The initiating device will generate a 128-bit random number that is combined with – TK,

  Pairing Request command,

  Pairing Response command,

  Initiating device address and address type,

  and responding device address and address type.

# BLE security

## Overview – Anatomy of Pairing Methods

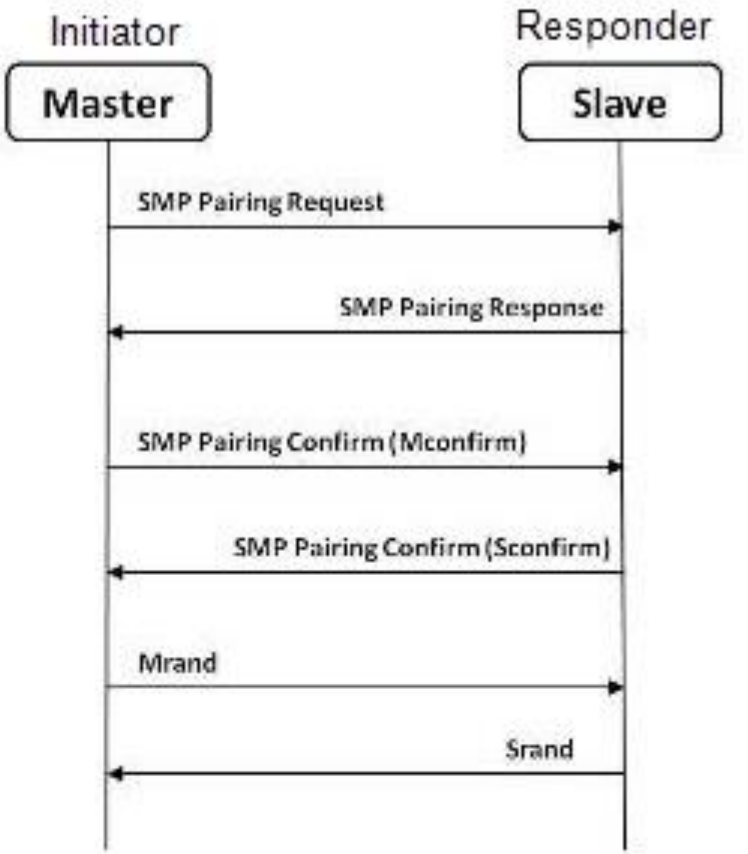
- The resulting value is a random number **Mconfirm** that is sent to the responding device by the pairing confirm command.

```
SMP:
    Code: Pairing Confirm
    Confirm Value: 0xfadc394940947c6edb6ffee9f399c9d5
```

- The responding device will validate the responding device data in the Pairing Confirm command and if it is correct will generate a **Sconfirm** value using the same methods as used to generate **Mconfirm** only with different 128-bit random number and TK.

```
SMP:
    Code: Pairing Confirm
    Confirm Value: 0x7fc2569e13e92125798a45a64256208a
```

- The responding device will send a Pairing Confirm command to the initiator and if accepted the authentication process is complete.



# BLE security

## Overview – Anatomy of Pairing Methods

- **Mrand** is the random number in **Mconfirm**.
- **Srand** is the random number in **Sconfirm**.
- **Mrand** and **Srand** have a key role in setting encrypting the link.
- Finally the master and slave devices exchange **Mrand** and **Srand** so that the slave can calculate and verify **Mconfirm** and the master can likewise calculate and verify **Sconfirm**.
- The **Short Term Key (STK)** is used for encrypting the link the first time the two devices pair. **STK** remains in each device on the link and is not transmitted between devices. STK is formed by combining **Mrand** and **Srand** which were formed using device information and TKs exchanged with Pairing Confirmation (**Pairing Confirm**).

# Contents

**Source code discussion WRT Security**

What would you see as output

# Custom service

## Custom service profile example

- **This example demonstrates:**
  - Simple bonding based on custom profile database
  - This tutorial covers a step by step procedure how to enable security during the process of device connection between a master and a slave.

- **Software you need:**
  - Dialog Smartsnippets studio
  - Dialog SDK
  - Project location:
    - ..\projects\target_apps\ble_examples\ble_app_security

# Code

**target_apps\ble_examples\ble_app_security project covers**

- Configuring security parameters using passkey procedure of association model
- Applying security in custom profile

# Code

## Custom service profile basic message flow

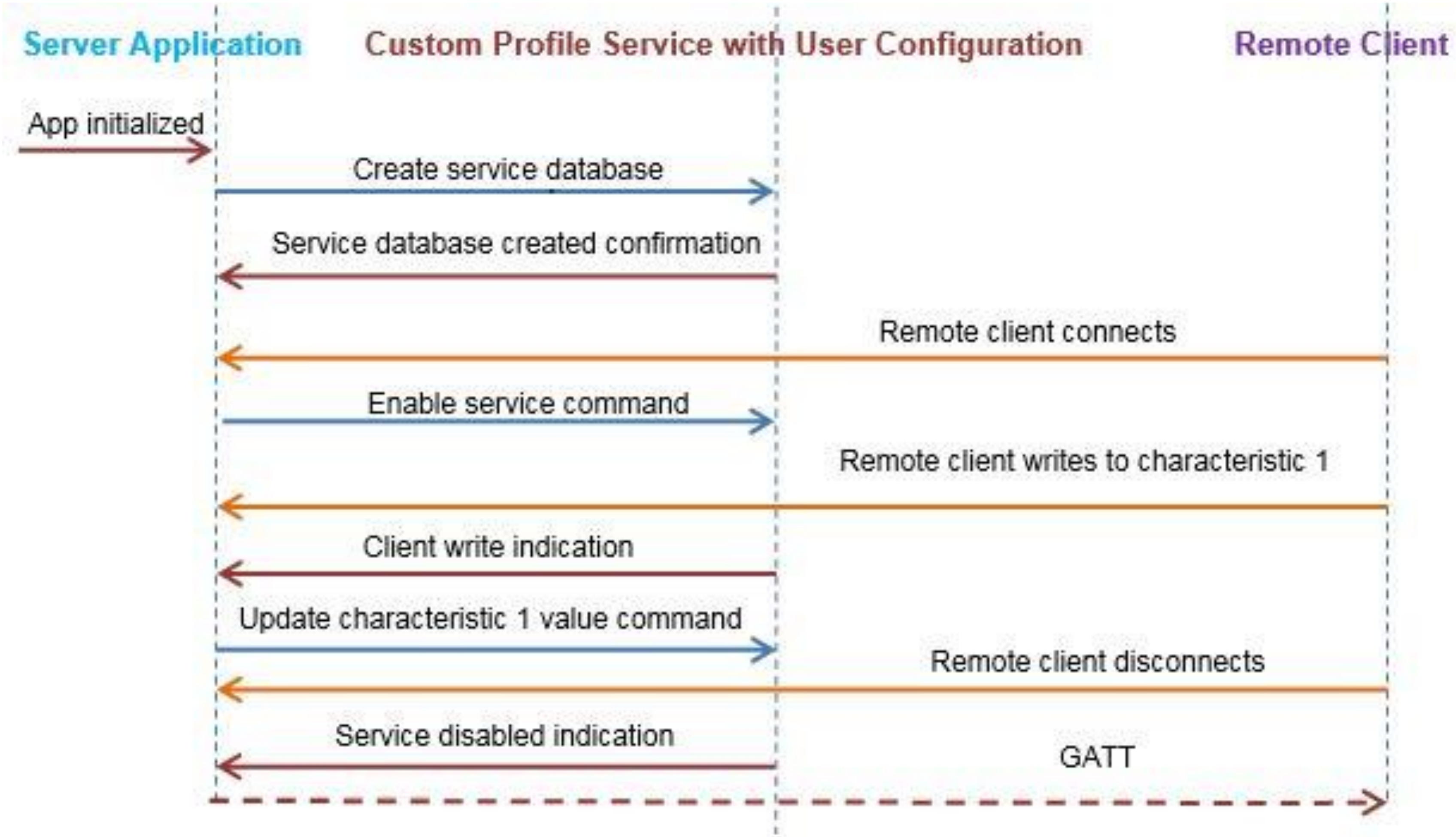
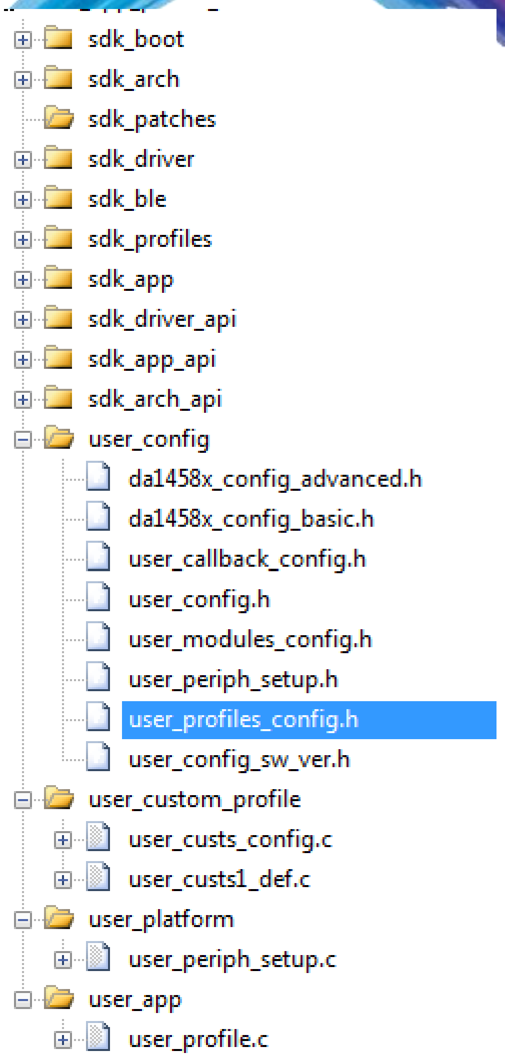


Figure: Message flow diagram

# Code

## ble_app_security.uvprojx project layout

- Group ***user_config***, ***user_platform*** and ***user_app***.
- These groups contain the user configuration files.



- sdk_boot
- sdk_arch
- sdk_patches
- sdk_driver
- sdk_ble
- sdk_profiles
- sdk_app
- sdk_driver_api
- sdk_app_api
- sdk_arch_api
- user_config
  - da1458x_config_advanced.h
  - da1458x_config_basic.h
  - user_callback_config.h
  - user_config.h
  - user_modules_config.h
  - user_periph_setup.h
  - user_profiles_config.h
  - user_config_sw_ver.h
- user_custom_profile
  - user_custs_config.c
  - user_custs1_def.c
- user_platform
  - user_periph_setup.c
- user_app
  - user_profile.c

# Code

## Description of some important files

```
/* Holds DA1458x basic configuration settings.    */
da1458x_config_basic.h

/* Holds DA1458x advanced configuration settings. */
da1458x_config_advanced.h

/* Holds user specific information about software version. */
user_config_sw_ver.h

/* Defines which application modules are included or excluded from the user’s application. */
user_modules_config.h

            /* The Device information application profile is excluded. */
            #define EXCLUDE_DLG_PROXR         (1)
            /* The Device information application profile is included. */
            #define EXCLUDE_DLG_CUSTS1        (0)

            /* Note:                                                                            */
            /*      This setting has no effect if the respective module is a BLE Profile        */
            /*      that is not used in the user's application.                                 */


/* Callback functions that handle various events or operations. */
user_callback_config.h


/* Holds advertising parameters, connection parameters, and compile time security parameters etc. */
user_config.h
```

# Code

## Description of some important files

```
/* Defines which BLE profiles (Bluetooth SIG adopted or custom ones) will be included in user’s application.
   each header file denotes the respective BLE profile*/
user_profiles_config.h

            #inlucde "diss.h"        // Includes Device Information Service.
            #include "custs1.h"      // Includes Custom service.
```

Note: SDK6 has provided a robust interface so the above implementation is done by MACRO flags

```
            #define CFG_PRF_DISS
            #define CFG_PRF_CUST1
```

```
/* Defines the structure of the Custom profile database structure and
   cust_prf_funcs[] array, which contains the Custom profile API functions calls.*/
user_custs_config.h
```

Note: SDK6 uses the following file for the same purpose

```
user_custs_config.c
```

```
/* Holds hardware related settings relative to the used Development Kit. */
user_periph_setup.h
```

```
/* Source code file that handles peripheral (GPIO, UART, SPI, etc.)
   configuration and initialization relative to the Development Kit.*/
user_periph_setup.c
```

# Code

## Security step by step

**TODO 1** - Change the default **BD_ADDRESS, this address has to be unique in a BLE network.**

```
/* @file da1458x_config_advanced.h */
```

```
/* copy and paste in code step 1 change the BLE device address */
#define CFG_NVDS_TAG_BD_ADDRESS                {0x01, 0x01, 0x01, 0x01, 0x01, 0x01}
```

**TODO 2** - Check and define **DLG_CUST1** module in your application code

```
/* @file user_modules_config.h */
```

```
#define EXCLUDE_DLG_SPOTAR              (1)            /* excluded */
/* copy and paste in code step 2 define DLG_CUST1 module in your application code */
#define EXCLUDE_DLG_CUSTS1              (0)            /* included */
```

**TODO 3** - Check and include **cust1.h** in your application code to activate custom profile

```
/* @file user_profiles_config.h */
```

```
#include "diss.h"
/* copy and paste in code step 3 add custs1.h NOTE: For SDK6 check the MACRO flags mentioned in slide 14 */
#include "custs1.h"
```

# Code

## Security step by step

**TODO 4** - Information and change your advertising device name

```
/* @file user_config.h */
```

```
/* default sleep mode. Possible values ARCH_SLEEP_OFF, ARCH_EXT_SLEEP_ON, ARCH_DEEP_SLEEP_ON
   ARCH_EXT_SLEEP_ON, ARCH_DEEP_SLEEP_ON – You cannot debug in these modes
*/
const static sleep_state_t app_default_sleep_mode = ARCH_SLEEP_OFF;
//-------------NON-CONNECTABLE & UNDIRECTED ADVERTISE RELATED COMMON -- //
/// Advertising service data
/// dev step 5 explanation of the following 3 items

#define USER_ADVERTISE_DATA ("\x03"\
            ADV_TYPE_COMPLETE_LIST_16BIT_SERVICE_IDS\
            ADV_UUID_DEVICE_INFORMATION_SERVICE\
            "\x11"\                                  /// The next section takes hex x11 = decimal 17 bytes
            ADV_TYPE_COMPLETE_LIST_128BIT_SERVICE_IDS\      /// Shows complete list of 128 bit Service IDs
            "\x2F\x2A\x93\xA6\xBD\xD8\x41\x52\xAC\x0B\x10\x99\x2E\xC6\xFE\xED") /// Your Custom Service UUID
/// Note- Custom service UUID is shown from right to left <-- EDFEC6...2F in the client LightBlue iOS app GUI
/* copy and paste in code step 4 change your advertising device name */
#define USER_DEVICE_NAME    ("B-SEC1")
```

# Code

## Security step by step

**TODO 5** – Overview of existing BLE Profile custom service characteristic values and properties

| **NAME** | **PROPERTIES** | **LENGTH** | **DESCRIPTION** |
|---|---|---|---|
| Control Point | WRITE | 1 | Accept commands from peer |
| LED State | WRITE NO RESPONSE | 1 | Toggles a LED connected to a GPIO |
| ADC Value 1 | READ, NOTIFY | 2 | Reads sample from an ADC channel |
| ADC Value 2 | READ | 2 | Reads sample from an ADC channel |
| Button State | READ, NOTIFY | 1 | Reads the current state of a push button connected a GPIO |
| Indicate able | READ, INDICATE | 20 | Demonstrate indications |
| Long Value | READ, WRITE. NOTIFY | 50 | Demonstrate writes to long characteristic value |

# Code

## Security step by step

**TODO 6** - Now define or enable the application security flag

`/* @file` **`da1458x_config_basic.h`** `*/`

```
/**********************************************************************************************************************************************************/
/* Enables the BLE security functionality in TASK_APP. If not defined BLE security related code is compiled out.*/
/*********************************************************************************************************************************************************/
#define CFG_APP_SECURITY
```

**TODO 7 –** Now define or enable compile time security feature wrt association model

`/* @file` **`user_config.h`** `*/`

```
/****************************************************************************************************************************************************
 * Pairing Methods: -              JUST WORKS  (#define USER_CFG_PAIR_METHOD_JUST_WORKS)
 *                                 PASSKEY     (#define USER_CFG_PAIR_METHOD_PASSKEY)
 *                                 OOB         (#define USER_CFG_PAIR_METHOD_OOB)
 * Select only one option.
 ******************************************************************************************************************************************************/
#define USER_CFG_PAIR_METHOD_PASSKEY
```

# Code

## Security step by step

**TODO 8** - Now for simplicity use a public address to play around privacy feature

`/* @file user_config.h */`

```
/****************************************************************************************************
 * Privacy feature:
 * PRIV_GEN_STATIC_RND    (#define USER_CFG_PRIV_GEN_STATIC_RND)
 * PRIV_GEN_RSLV_RND      (#define USER_CFG_PRIV_GEN_RSLV_RND)
 * This configuration flags are used for selecting privacy feature of the peripheral device.
 * This feature allows the device to use random addresses to prevent peers from tracking it.
 * Privacy feature is selected through the following two flags.
 * Select only one option for random address. If none is selected, a public
 * address will be used.
 ****************************************************************************************************/

#undef USER_CFG_PRIV_GEN_STATIC_RND
#undef USER_CFG_PRIV_GEN_RSLV_RND
```

# Code

## Security step by step

**TODO 9** - Peer device’s bond data can be stored on an external SPI Flash or I2C EEPROM memory. Un-define both to store bonding information in sysRAM for application simplicity

```
/* @file user_config.h */


/*************************************************************
 * Select memory medium for bond data storage:
 * - SPI FLASH              (#define USER_CFG_APP_BOND_DB_USE_SPI_FLASH)
 * - I2C EEPROM             (#define USER_CFG_APP_BOND_DB_USE_I2C_EEPROM)
 * - SysRAM only            (define nothing)
 * Select only one option.
 **************************************************************/
#undef USER_CFG_APP_BOND_DB_USE_SPI_FLASH
#undef USER_CFG_APP_BOND_DB_USE_I2C_EEPROM
```

# Code

## Security step by step

**TODO 10 – BLE Security configuration, it should look like the following**

```
/* @file user_config.h */
```

```
/* ********************************************************************************
 * Security related configuration simplified view as the structure is very huge
 ********************************************************************************/
static const struct security_configuration user_security_conf = {
    /*******************************************************************************
     * IO capabilities (@see gap_io_cap)
     * - GAP_IO_CAP_NO_INPUT_NO_OUTPUT     No Input No Output
     *******************************************************************************/
    .iocap          = GAP_IO_CAP_NO_INPUT_NO_OUTPUT,

    /*******************************************************************************
     * OOB information (@see gap_oob)
     * - GAP_OOB_AUTH_DATA_NOT_PRESENT     OOB Data not present
     *******************************************************************************/
    .oob            = GAP_OOB_AUTH_DATA_NOT_PRESENT,
```

# Code

## Security step by step

**TODO 10 – BLE Security configuration, it should look like the following**

`/* @file user_config.h */`

```
/************************************************************************************
 * Authentication (@see gap_auth)
 * - GAP_AUTH_REQ_MITM_BOND          MITM and Bonding
 ************************************************************************************/
#if defined (USER_CFG_PAIR_METHOD_PASSKEY)
.auth         = GAP_AUTH_REQ_MITM_BOND,
#endif

/************************************************************************************
 * Device security requirements (minimum security level). (@see gap_sec_req)
 * - GAP_SEC1_AUTH_PAIR_ENC          Authenticated pairing with encryption
 ************************************************************************************/
#if defined (USER_CFG_PAIR_METHOD_PASSKEY)
.sec_req      = GAP_SEC1_AUTH_PAIR_ENC,
#endif
```

# Code

## Security step by step

**TODO 10 – BLE Security configuration, it should look like the following**

`/* @file user_config.h */`

```
  /// Encryption key size (7 to 16) - LTK Key Size
 .key_size       = KEY_LEN,
 /************************************************************************************
  * Initiator key distribution (@see gap_kdist)
  * - GAP_KDIST_IDKEY                 IRK (ID key)in distribution
  * - GAP_KDIST_SIGNKEY                CSRK (Signature key) in distribution
  ************************************************************************************/
 #if defined (USER_CFG_PAIR_METHOD_JUST_WORKS) || defined (USER_CFG_PAIR_METHOD_PASSKEY) || defined (USER_CFG_PAIR_METHOD_OOB)
 .ikey_dist      = GAP_KDIST_SIGNKEY | GAP_KDIST_IDKEY,
 #endif
 /************************************************************************************
  * Responder key distribution (@see gap_kdist)
  * - GAP_KDIST_ENCKEY                 LTK (Encryption key) in distribution
  ************************************************************************************/
 #if defined (USER_CFG_PAIR_METHOD_JUST_WORKS) || defined (USER_CFG_PAIR_METHOD_PASSKEY) || defined (USER_CFG_PAIR_METHOD_OOB)
     .rkey_dist      = GAP_KDIST_ENCKEY,
 #endif
};
```

# Code

## Security step by step

**TODO 11 – Apply your passkey**

`/* @file user_security.h */`

```
/// Passkey that is presented to the user and is entered on the peer device (MITM) <= 6 digit number
#define APP_SECURITY_MITM_PASSKEY_VAL    (321456)
```

# Code

## Security step by step

**TODO 12** - BLE events are processed using the following callbacks in Dialog SDK

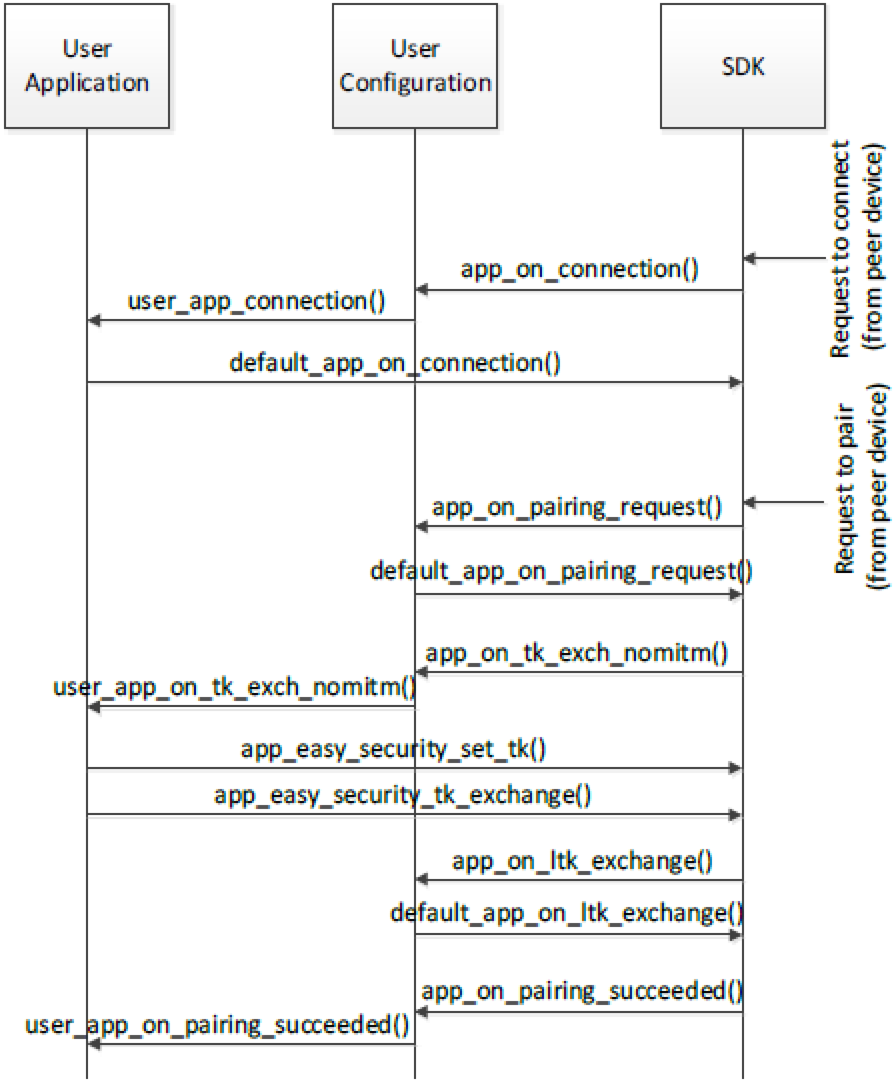
`/* @file user_callback_config.h */`

```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection                  = user_app_connection,
    .app_on_disconnect                  = user_app_disconnect,
    .app_on_set_dev_config_complete     = default_app_on_set_dev_config_complete,
    .app_on_adv_undirect_complete       = user_app_adv_undirect_complete,
    .app_on_db_init_complete            = default_app_on_db_init_complete,
    .app_on_get_dev_appearance          = default_app_on_get_dev_appearance,
    .app_on_get_dev_slv_pref_params     = default_app_on_get_dev_slv_pref_params,
    .app_on_set_dev_info                = default_app_on_set_dev_info,
    .app_on_update_params_request       = default_app_update_params_request,
#if (BLE_APP_SEC)
    .app_on_pairing_request             = default_app_on_pairing_request,
    .app_on_tk_exch_nomitm              = user_app_on_tk_exch_nomitm,
    .app_on_ltk_exch                    = default_app_on_ltk_exch,
    .app_on_pairing_succeded            = user_app_on_pairing_succeeded,
    .app_on_encrypt_req_ind             = user_app_on_encrypt_req_ind,
#endif // (BLE_APP_SEC)
};
```

# Code

## Abstract code flow

**Pairing using Passkey Entry**



User Application

User Configuration

SDK

Request to connect (from peer device)

app_on_connection()

user_app_connection()

default_app_on_connection()

Request to pair (from peer device)

app_on_pairing_request()

default_app_on_pairing_request()

app_on_tk_exch_nomitm()

user_app_on_tk_exch_nomitm()

app_easy_security_set_tk()

app_easy_security_tk_exchange()

app_on_ltk_exchange()

default_app_on_ltk_exchange()

app_on_pairing_succeeded()

user_app_on_pairing_succeeded()

# Code

## Security step by step

**TODO 13** - Apply Permission on a GATT characteristic value. This can be achieved by changing the permissions from UNAUTH to AUTH. Using this setting the following:

**`.security_request_scenario = DEF_SEC_REQ_ON_CONNECT`**

in **user_config.h** you can select when authorization is required, during connection or during read/write of a characteristic.

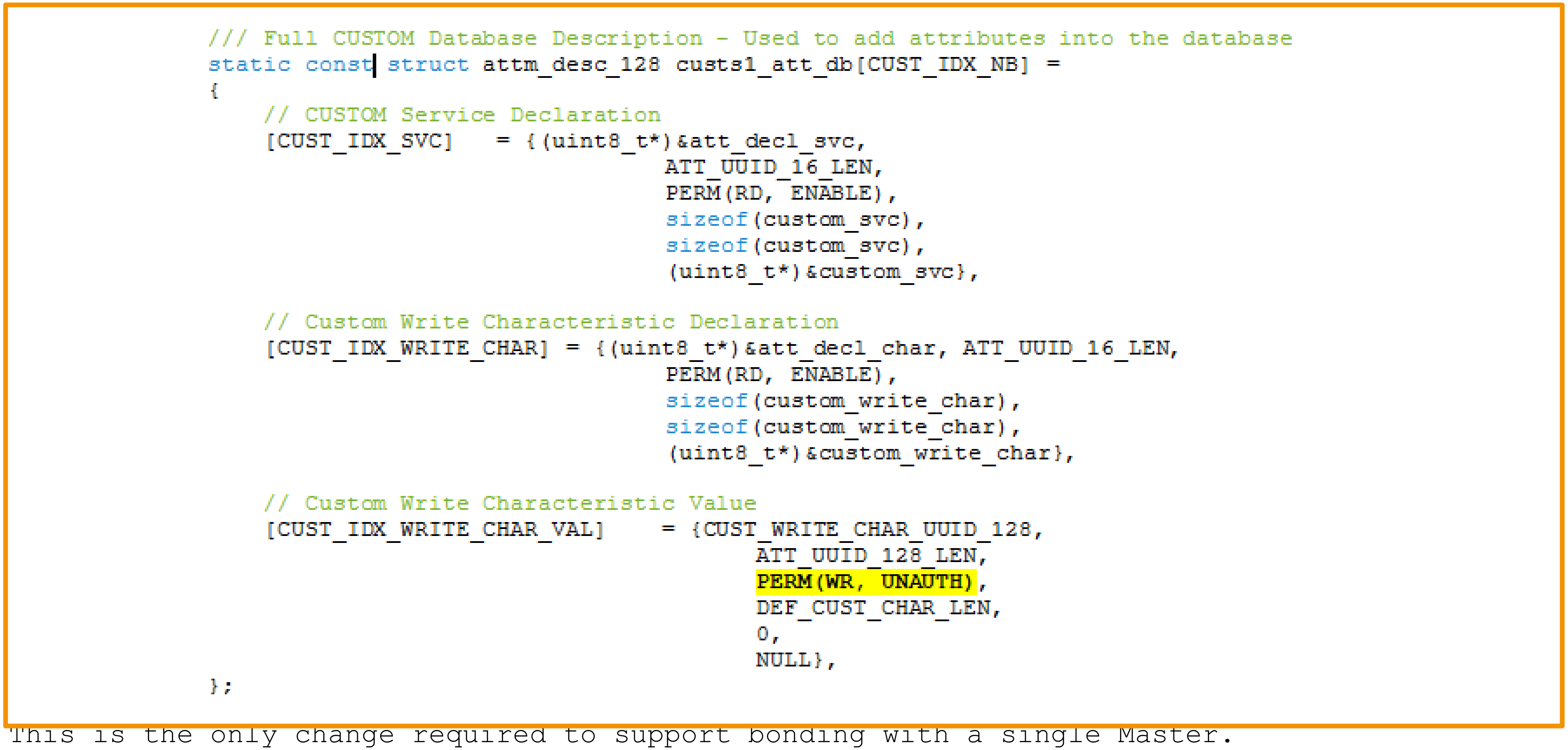
```
/* @file user_config.h */
```

```
static const struct default_handlers_configuration  user_default_hnd_conf = {
    //Configure the advertise operation used by the default handlers
    //Possible values:
    //  - DEF_ADV_FOREVER
    //  - DEF_ADV_WITH_TIMEOUT
    .adv_scenario = DEF_ADV_FOREVER,

    //Configure the advertise period in case of DEF_ADV_WITH_TIMEOUT.
    //It is measured in timer units (3 min). Use MS_TO_TIMERUNITS macro to convert
    //from milliseconds (ms) to timer units.
    .advertise_period = MS_TO_TIMERUNITS(180000),

    //Configure the security start operation of the default handlers
    //if the security is enabled (CFG_APP_SECURITY)
    .security_request_scenario = DEF_SEC_REQ_ON_CONNECT
};
```

# Code

## Single Device Bonding Example

**TODO 14 –**To convert an existing read or write characteristic to require pairing change the Characteristic Value permissions in the Database Description change the permission flag:



```
/// Full CUSTOM Database Description - Used to add attributes into the database
static const struct attm_desc_128 custs1_att_db[CUST_IDX_NB] =
{
    // CUSTOM Service Declaration
    [CUST_IDX_SVC]   = {(uint8_t*)&att_decl_svc,
                                 ATT_UUID_16_LEN,
                                 PERM(RD, ENABLE),
                                 sizeof(custom_svc),
                                 sizeof(custom_svc),
                                 (uint8_t*)&custom_svc},

    // Custom Write Characteristic Declaration
    [CUST_IDX_WRITE_CHAR] = {(uint8_t*)&att_decl_char, ATT_UUID_16_LEN,
                                 PERM(RD, ENABLE),
                                 sizeof(custom_write_char),
                                 sizeof(custom_write_char),
                                 (uint8_t*)&custom_write_char},

    // Custom Write Characteristic Value
    [CUST_IDX_WRITE_CHAR_VAL]    = {CUST_WRITE_CHAR_UUID_128,
                                        ATT_UUID_128_LEN,
                                        PERM(WR, UNAUTH),
                                        DEF_CUST_CHAR_LEN,
                                        0,
                                        NULL},
};
```

This is the only change required to support bonding with a single Master.

# Contents

## What would you see as output

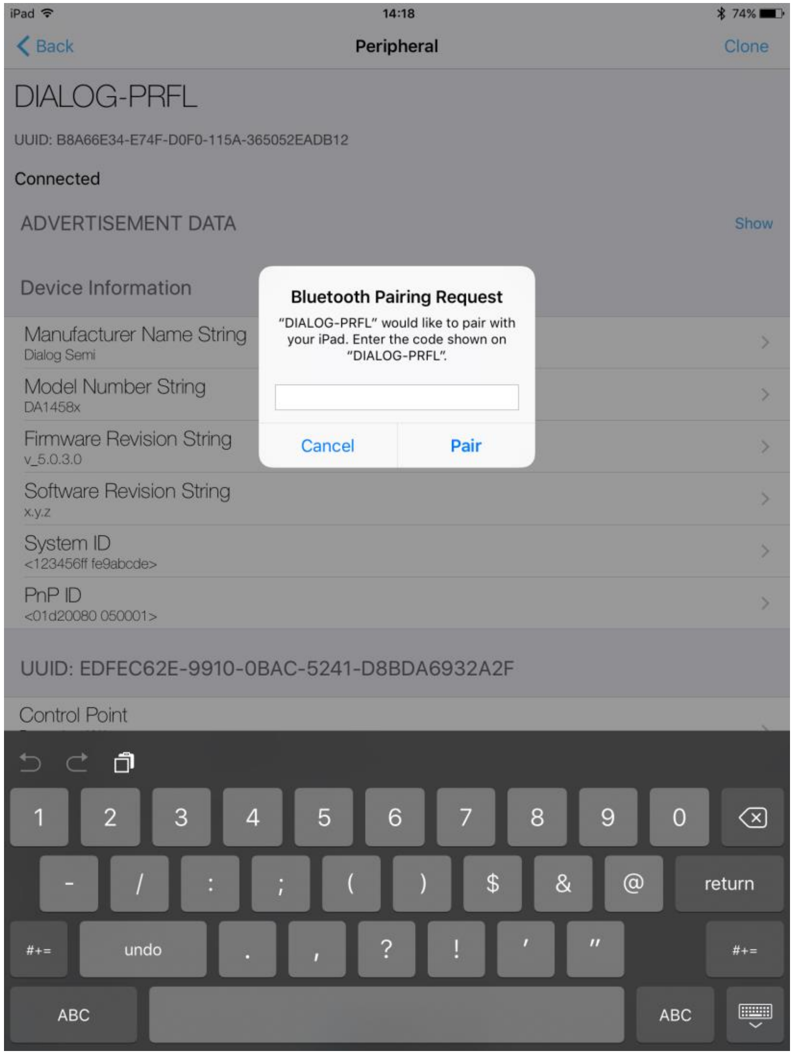
# Output

## Output

- The LightBlue iOS application can be used to connect an iPad/iPod/iPhone device to the application. In such a case the iPad/iPod/iPhone acts as a BLE Central and the application as a BLE Peripheral. It should be listed by the name given in the USER_DEVICE_NAME definition.
- One service should be listed – the Device Information Service. On some scanners, this will be listed either as a named service, or as a set of hex numbers (0A 18) as part of a list of 16-bit Service class UUIDs.
- On connecting to the device, the Characteristics should be retrieved.

# Output

## Passkey



DIALOG-PRFL

UUID: B8A66E34-E74F-D0F0-115A-365052EADB12

Connected

ADVERTISEMENT DATA

Device Information

Manufacturer Name String
Dialog Semi

Model Number String
DA1458x

Firmware Revision String
v_5.0.3.0

Software Revision String
x.y.z

System ID
<123456ff fe9abcde>

PnP ID
<01d20080 050001>

UUID: EDFEC62E-9910-0BAC-5241-D8BDA6932A2F

Control Point

**Bluetooth Pairing Request**

"DIALOG-PRFL" would like to pair with your iPad. Enter the code shown on "DIALOG-PRFL".

# Output

- **Note**: The devices will be connectable in this and future examples. Connecting to a device will mean that other scanners won't be able to locate the device – it is recommended that you only connect to your own device.

- **Note**: Some scanners (notably Apple devices) may not update the name of device if it is changed – to correct this, it is necessary to disable then re-enable Bluetooth.

# Output

## Multiple Device Bonding

- To support multiple devices, the bonding information must be stored for each device.

- The simplest way is to store the bonding information in retained memory, using the attribute `__attribute__((section("retention_mem_area0"),zero_init)).`

- The SDK **app_sec** module provides a structure `app_sec_env` in this retained memory in which bonding information may be stored.
  - This module also provides helper functions to generate PINs and the Long Term Key

# Output

## Multiple Device Bonding

- Bonding information may be stored and used using the following procedure:
  - On successful pairing, the callback `.app_on_pairing_succeded` will be called. At this point you may store the `app_sec_env.rand_nb`, `app_sec_env.ediv`, `app_sec_env.ltk` and `app_sec_env.key_size` values to your permanent store.

  - When the callback `.app_on_encrypt_req_ind` is called, do a lookup on the `app_sec_env.rand_nb` and `app_sec_env.ediv` variables stored previously. If a match is found, write the values to `app_sec_env.rand_nb`, `app_sec_env.ediv`, `app_sec_env.ltk` and `app_sec_env.key_size` and return true.

- Dialog's IoT Sensor and Keyboard reference designs (available through support.dialog-semiconductor.com)  include example code dealing with storing bonding information to EEPROM.

# Output

## Further Considerations

- Other pairing modes:
  - Other pairing methods like Pass Key Entry and Out Of Band are supported by the Dialog platform and SDK
- Private addresses:
  - Generally, a peripheral will broadcast its presence to all listeners using the same address every time. It is possible to obfuscate the identity of a peripheral using a ‘Private Address’
  - Only devices which are bonded to the client can resolve the address, using their stored keys.
- Bondable / non-bondable:
  - When a master connects to a BLE slave, it may only pair if the slave allows it.
  - Typically, bonding can be controlled using a user interaction on the device – for example, pressing a specific button will start the device advertising in a mode that allows bonding.

# Output

## Further Considerations

- The DA14580 does not store any bonding info after power cycling the device. Even if the PIN code is not changed, the LTK is changed every time.
  - It is recommended to remove the bonding info in the Smartphone/Tablet.
  - It is normal because when you reset the DA14580, the keys do not match anymore.
  - There is a random part in the key so the bonding information is not stored in the memory of DA14580.

# Information

## What would you see as output

- **Note**: The devices will be connectable in this and future examples. Connecting to a device will mean that other scanners won't be able to locate the device – it is recommended that you only connect to your own device.

- **Note**: Some scanners (notably Apple devices) may not update the name of device if it is changed – to correct this, it is necessary to disable then re-enable Bluetooth.

# Some more easy tasks

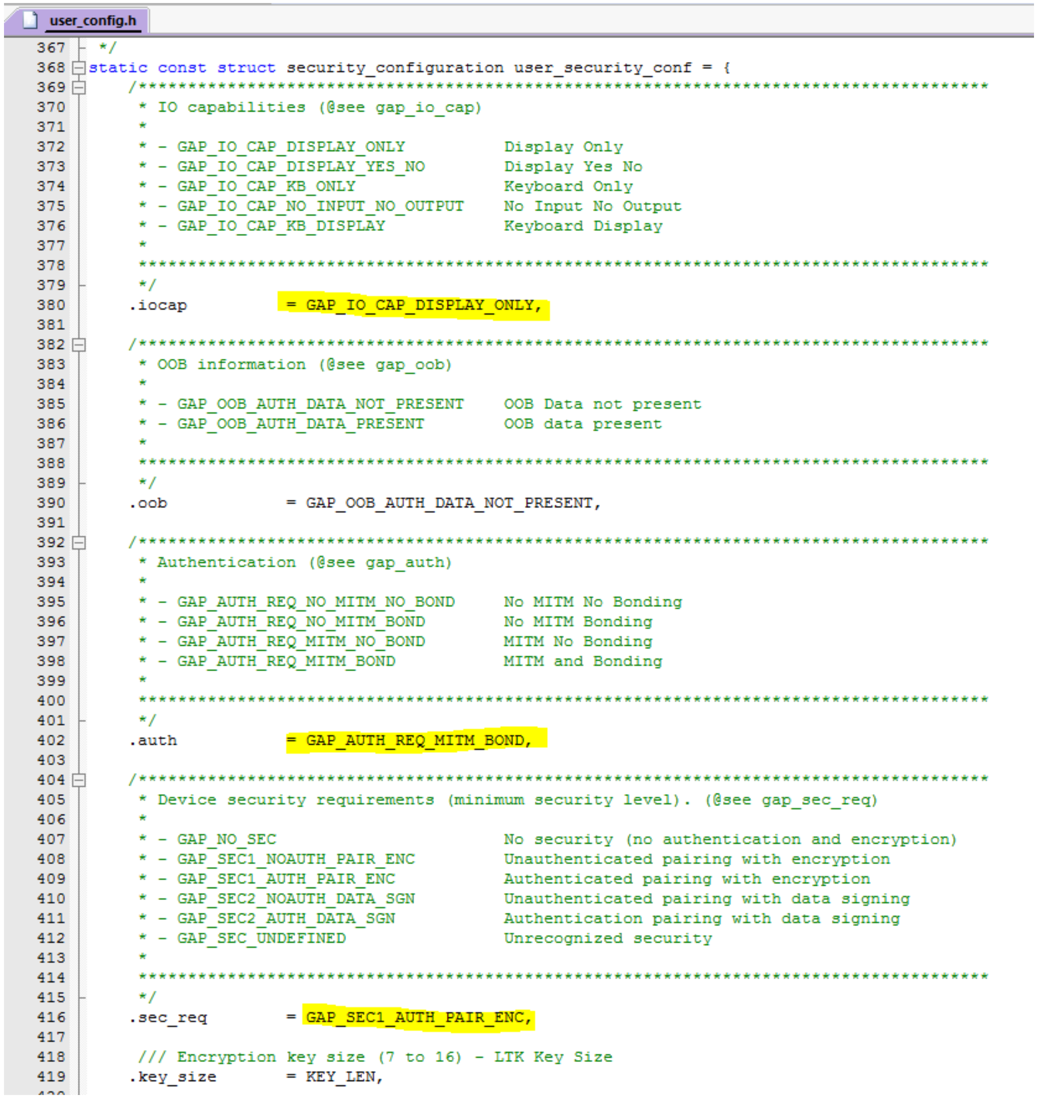
Small Do-it-youself assignement with code indication

- **Task 1:**
- Implement MITM security with access key provided out of band
- Use clean SDK5.0.4 empty_peripheral_template project as starting point
- Will trigger a pin code prompt upon connection establishment

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# Security configuration

In ‘user_config.h’, change to these settings:

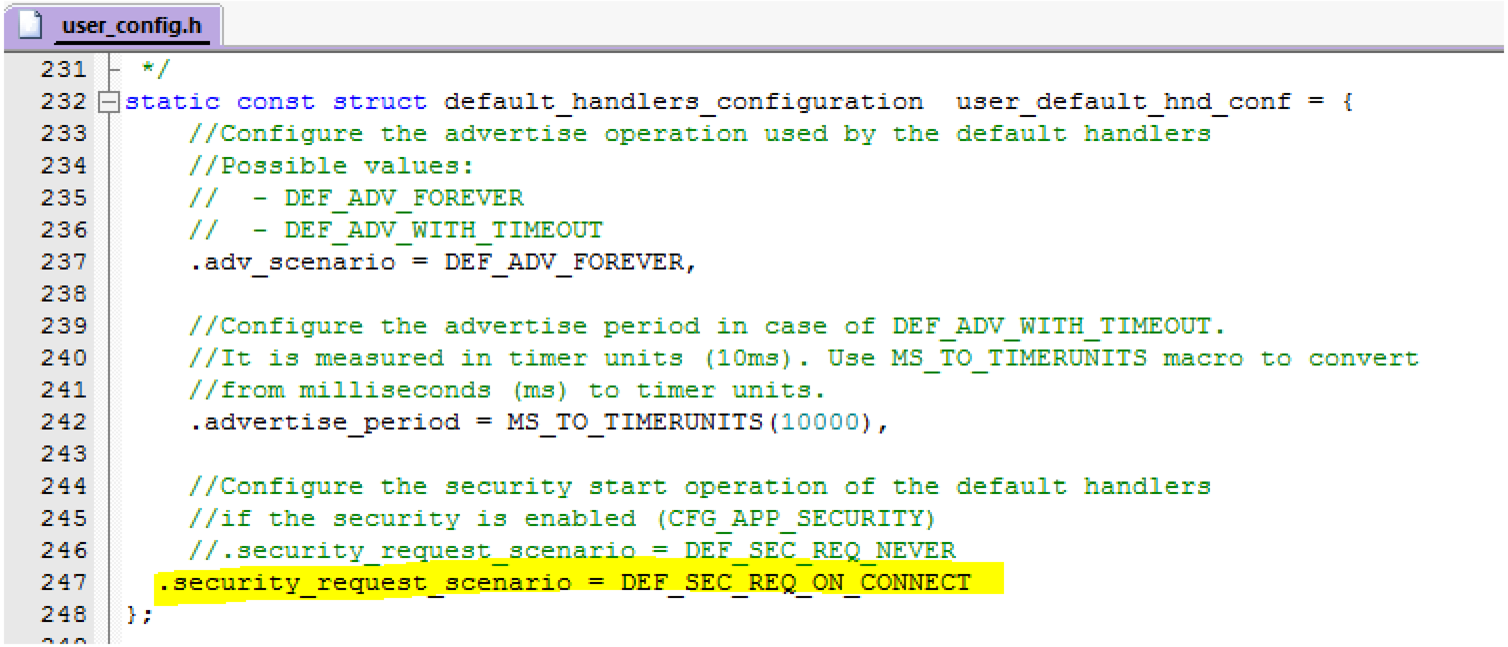


```
 */
static const struct security_configuration user_security_conf = {
    /**************************************************************************************
     * IO capabilities (@see gap_io_cap)
     *
     * - GAP_IO_CAP_DISPLAY_ONLY          Display Only
     * - GAP_IO_CAP_DISPLAY_YES_NO        Display Yes No
     * - GAP_IO_CAP_KB_ONLY               Keyboard Only
     * - GAP_IO_CAP_NO_INPUT_NO_OUTPUT    No Input No Output
     * - GAP_IO_CAP_KB_DISPLAY            Keyboard Display
     *
     **************************************************************************************
     */
    .iocap          = GAP_IO_CAP_DISPLAY_ONLY,

    /**************************************************************************************
     * OOB information (@see gap_oob)
     *
     * - GAP_OOB_AUTH_DATA_NOT_PRESENT    OOB Data not present
     * - GAP_OOB_AUTH_DATA_PRESENT        OOB data present
     *
     **************************************************************************************
     */
    .oob            = GAP_OOB_AUTH_DATA_NOT_PRESENT,

    /**************************************************************************************
     * Authentication (@see gap_auth)
     *
     * - GAP_AUTH_REQ_NO_MITM_NO_BOND     No MITM No Bonding
     * - GAP_AUTH_REQ_NO_MITM_BOND        No MITM Bonding
     * - GAP_AUTH_REQ_MITM_NO_BOND        MITM No Bonding
     * - GAP_AUTH_REQ_MITM_BOND           MITM and Bonding
     *
     **************************************************************************************
     */
    .auth           = GAP_AUTH_REQ_MITM_BOND,

    /**************************************************************************************
     * Device security requirements (minimum security level). (@see gap_sec_req)
     *
     * - GAP_NO_SEC                       No security (no authentication and encryption)
     * - GAP_SEC1_NOAUTH_PAIR_ENC         Unauthenticated pairing with encryption
     * - GAP_SEC1_AUTH_PAIR_ENC           Authenticated pairing with encryption
     * - GAP_SEC2_NOAUTH_DATA_SGN         Unauthenticated pairing with data signing
     * - GAP_SEC2_AUTH_DATA_SGN           Authentication pairing with data signing
     * - GAP_SEC_UNDEFINED                Unrecognized security
     *
     **************************************************************************************
     */
    .sec_req        = GAP_SEC1_AUTH_PAIR_ENC,

     /// Encryption key size (7 to 16) - LTK Key Size
    .key_size       = KEY_LEN,
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# Security configuration

In ‘user_config.h’, change to this setting



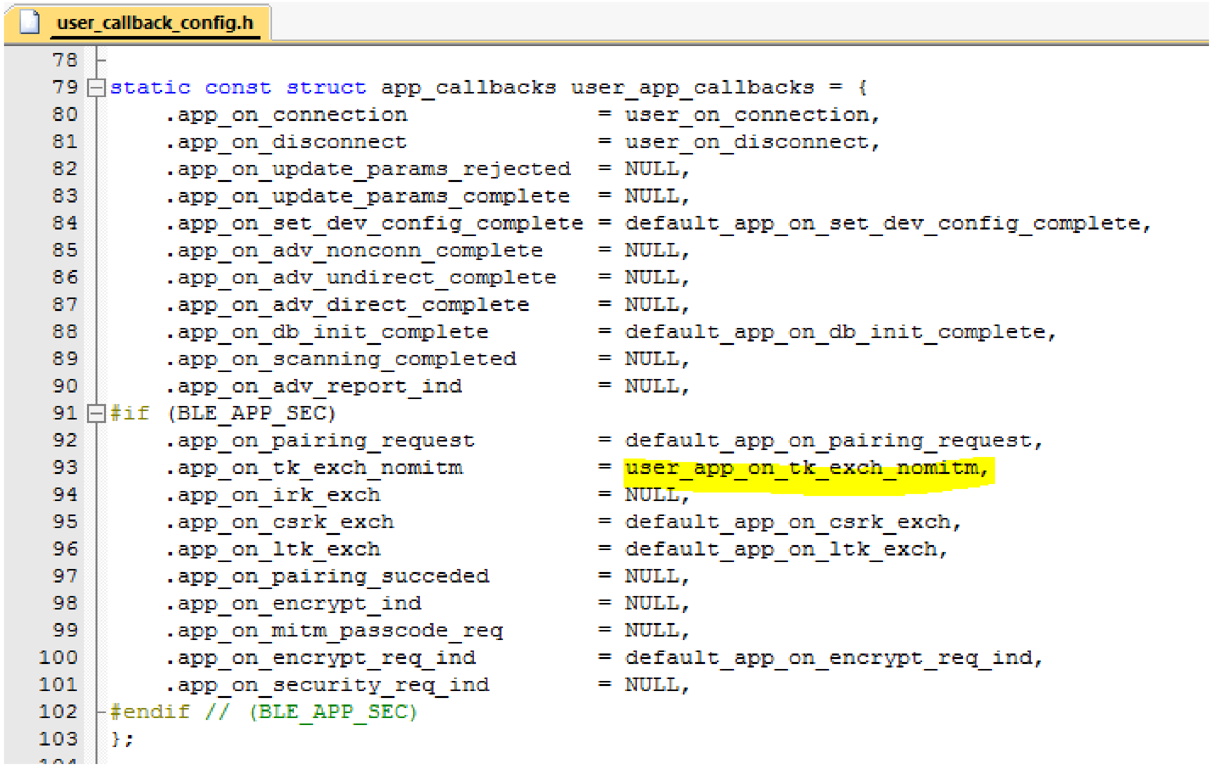
```
 */
static const struct default_handlers_configuration  user_default_hnd_conf = {
    //Configure the advertise operation used by the default handlers
    //Possible values:
    //  - DEF_ADV_FOREVER
    //  - DEF_ADV_WITH_TIMEOUT
    .adv_scenario = DEF_ADV_FOREVER,

    //Configure the advertise period in case of DEF_ADV_WITH_TIMEOUT.
    //It is measured in timer units (10ms). Use MS_TO_TIMERUNITS macro to convert
    //from milliseconds (ms) to timer units.
    .advertise_period = MS_TO_TIMERUNITS(10000),

    //Configure the security start operation of the default handlers
    //if the security is enabled (CFG_APP_SECURITY)
    //.security_request_scenario = DEF_SEC_REQ_NEVER
  .security_request_scenario = DEF_SEC_REQ_ON_CONNECT
};
```

```
.security_request_scenario = DEF_SEC_REQ_ON_CONNECT
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# Security configuration

In ‘user_callback_config.h’, route the tk exchange callback to user space:



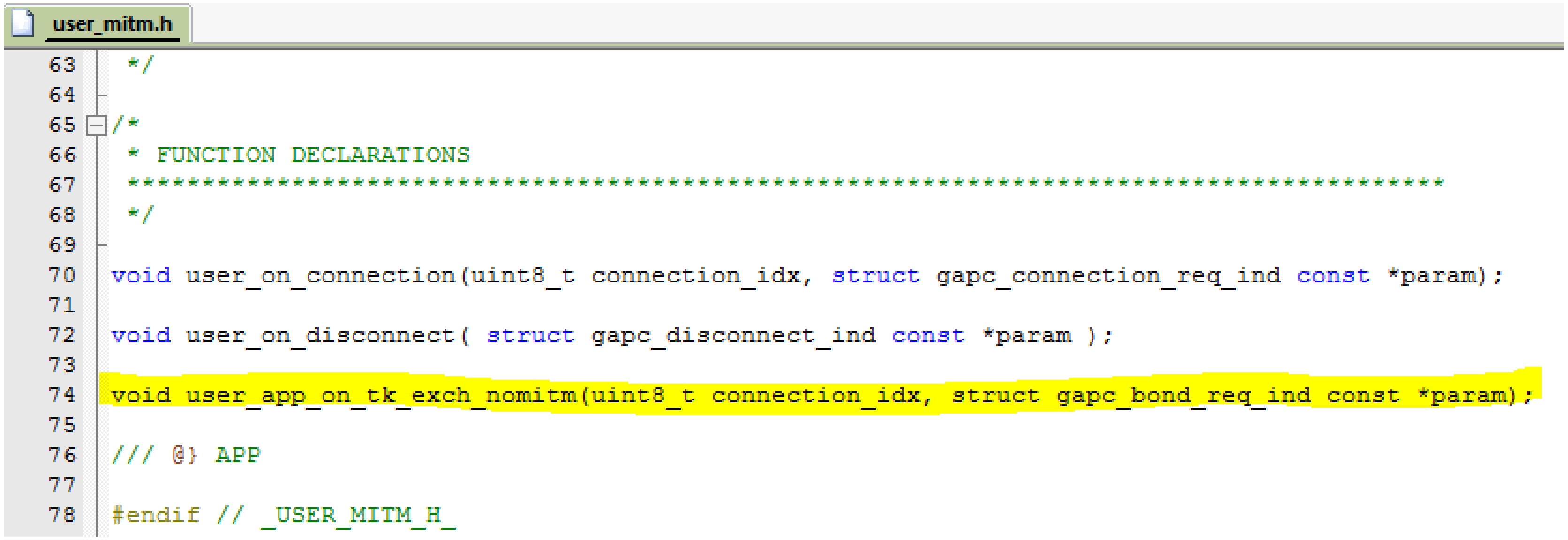
```
static const struct app_callbacks user_app_callbacks = {
    .app_on_connection                = user_on_connection,
    .app_on_disconnect                = user_on_disconnect,
    .app_on_update_params_rejected    = NULL,
    .app_on_update_params_complete    = NULL,
    .app_on_set_dev_config_complete = default_app_on_set_dev_config_complete,
    .app_on_adv_nonconn_complete      = NULL,
    .app_on_adv_undirect_complete     = NULL,
    .app_on_adv_direct_complete       = NULL,
    .app_on_db_init_complete          = default_app_on_db_init_complete,
    .app_on_scanning_completed        = NULL,
    .app_on_adv_report_ind            = NULL,
#if (BLE_APP_SEC)
    .app_on_pairing_request           = default_app_on_pairing_request,
    .app_on_tk_exch_nomitm            = user_app_on_tk_exch_nomitm,
    .app_on_irk_exch                  = NULL,
    .app_on_csrk_exch                 = default_app_on_csrk_exch,
    .app_on_ltk_exch                  = default_app_on_ltk_exch,
    .app_on_pairing_succeded          = NULL,
    .app_on_encrypt_ind               = NULL,
    .app_on_mitm_passcode_req         = NULL,
    .app_on_encrypt_req_ind           = default_app_on_encrypt_req_ind,
    .app_on_security_req_ind          = NULL,
#endif // (BLE_APP_SEC)
};
```

```
.app_on_tk_exch_nomitm  = user_app_on_tk_exch_nomitm,
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# Security configuration

In the main header file make a reference to the user function:



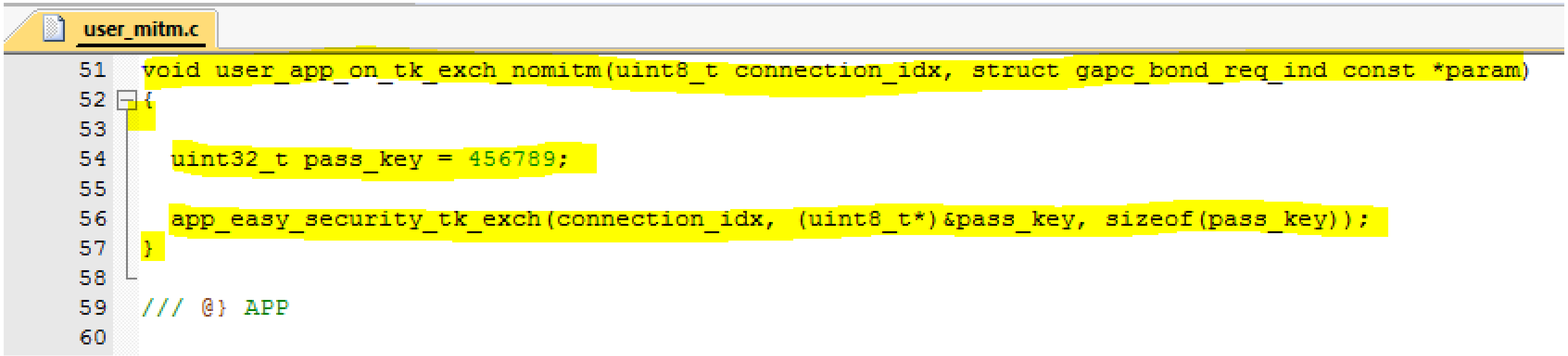
user_mitm.h

```
 */

/*
 * FUNCTION DECLARATIONS
 *****************************************************************************************
 */

void user_on_connection(uint8_t connection_idx, struct gapc_connection_req_ind const *param);

void user_on_disconnect( struct gapc_disconnect_ind const *param );

void user_app_on_tk_exch_nomitm(uint8_t connection_idx, struct gapc_bond_req_ind const *param);

/// @} APP

#endif // _USER_MITM_H_
```

```
void user_app_on_tk_exch_nomitm(uint8_t connection_idx, struct gapc_bond_req_ind const *param);
```

# Security configuration

In the main file implement a function that returns the access key:



```
void user_app_on_tk_exch_nomitm(uint8_t connection_idx, struct gapc_bond_req_ind const *param)
{

   uint32_t pass_key = 456789;

   app_easy_security_tk_exch(connection_idx, (uint8_t*)&pass_key, sizeof(pass_key));
}

/// @} APP

```

```
void user_app_on_tk_exch_nomitm(uint8_t connection_idx, struct gapc_bond_req_ind const *param)
{
  uint32_t pass_key = 456789;
  app_easy_security_tk_exch(connection_idx, (uint8_t*)&pass_key, sizeof(pass_key));
}
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# Some more easy tasks

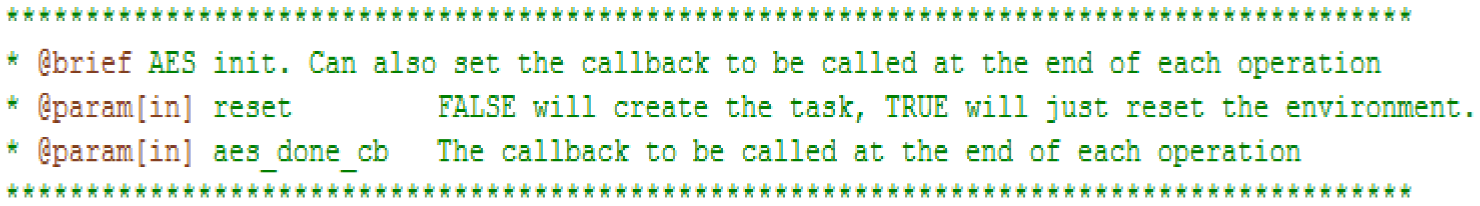
Small Do-it-youself assignement with code indication

- **Task 2:**
- AES 128 bit encryption / Decryption in DA1458x
- Demonstrates how to use AES library for data encryption and decryption
- Uses empty_peripheral_template as a starting point
- Assumes that you have arch_printf working
- Assumes that you have pointed the app_on_init callback to your user space
- AES encryption and decryption is achieved with only two function calls
- This tutorial only demonstrates synchronous mode does not utilize call-backs

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# The AES init function

aes_init():

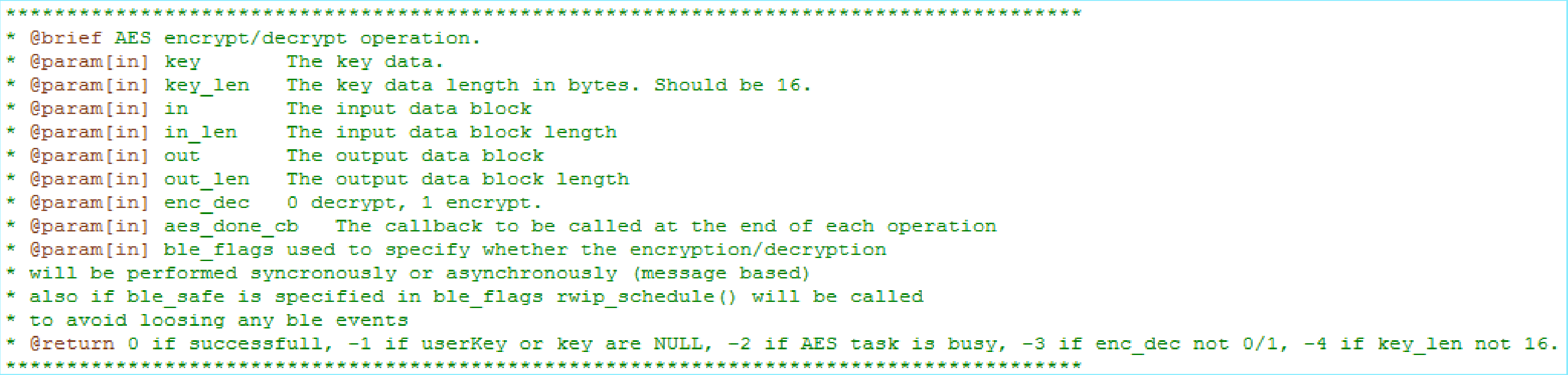


```
*********************************************************************************************
* @brief AES init. Can also set the callback to be called at the end of each operation
* @param[in] reset         FALSE will create the task, TRUE will just reset the environment.
* @param[in] aes_done_cb   The callback to be called at the end of each operation
*********************************************************************************************
```

```
void aes_init(bool reset, void (*aes_done_cb)(uint8_t status))
```

# The AES operation function

## aes_operation()



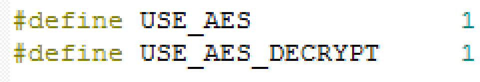
```
****************************************************************************************
* @brief AES encrypt/decrypt operation.
* @param[in] key        The key data.
* @param[in] key_len    The key data length in bytes. Should be 16.
* @param[in] in         The input data block
* @param[in] in_len     The input data block length
* @param[in] out        The output data block
* @param[in] out_len    The output data block length
* @param[in] enc_dec    0 decrypt, 1 encrypt.
* @param[in] aes_done_cb    The callback to be called at the end of each operation
* @param[in] ble_flags used to specify whether the encryption/decryption
* will be performed syncronously or asynchronously (message based)
* also if ble_safe is specified in ble_flags rwip_schedule() will be called
* to avoid loosing any ble events
* @return 0 if successfull, -1 if userKey or key are NULL, -2 if AES task is busy, -3 if enc_dec not 0/1, -4 if key_len not 16.
****************************************************************************************
```

```
  int aes_operation
  (
      unsigned char * key,                    // The AES encryption key
      int key_len,                            // The length of the key in number of octets
      unsigned char *in,                      // The input data
      int in_len,                             // The length of the input data in number of octets
      unsigned char *out,                     // The output data
      int out_len,                            // The length of the output data in number of octets
      int enc_dec,                            // 0 = Decryption, 1 = Encryption
      void(*aes_done_cb)(uint8_t status),     // Callback function called on completion (asynchronous use only)
      unsigned char ble_flags                 // Flags
)
```

© Copyright 2016 Dialog Semiconductor. All Rights Reserved

# Implementing AES support

In the user_config.h file add these two #defines:



```
#define USE_AES             1
#define USE_AES_DECRYPT     1
```

In the main user file, include the AES library:

```
/*
 * INCLUDE FILES
 ****************************************************************************************
 */

#include "aes.h"
```

```
#include "aes.h"
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# A helper function for visualization

In the main user file implement the following:

```
void user_serial_dump(char* label, uint8_t* data_ptr, uint8_t data_len, bool hex_format)
{
    // Print the label
    arch_printf("%s: ",label);

    // Iterate through the data and dump to serial port console in either hex or ascii format
    for(uint8_t i = 0; i < data_len; i++)
    {
        if(hex_format)
            arch_printf("0x%02X ",*data_ptr++);
        else
            arch_printf("%c",*data_ptr++);
    }

    // Add line-feed and carriage-return
    arch_puts("\n\r");
}
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# AES encryption and decryption function

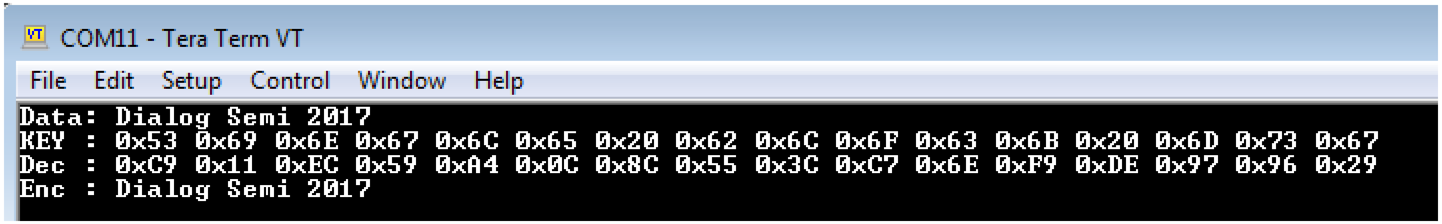
In the main user file implement the following:

```
void user_app_on_init(void)
{
    // Call the default handler
    default_app_on_init();

    // Set the AES initialization vector to all zeroes
    memset(aes_env.aes_key.iv, 0, KEY_LEN);

    // Initialize the AES environment
    aes_init(false, NULL);

    // Declare a result array
    uint8_t  aes_result[KEY_LEN];

    // Define some data and a key
    uint8_t aes_in[KEY_LEN]   = {'D', 'i', 'a', 'l', 'o', 'g', ' ', 'S', 'e', 'm', 'i', ' ', '2', '0', '1', '7'};
    uint8_t aes_key[KEY_LEN]  = {0x53, 0x69, 0x6e, 0x67, 0x6c, 0x65, 0x20, 0x62, 0x6c, 0x6f, 0x63, 0x6b, 0x20,
0x6d, 0x73, 0x67};

    // Dump cleartext data to console
    user_serial_dump("\n\rData", aes_in,  KEY_LEN, false);

    // Dump encryption key
    user_serial_dump("KEY ", aes_key, KEY_LEN, true);

    // Encrypt data using key
    aes_operation(aes_key, KEY_LEN, aes_in, KEY_LEN, aes_result, KEY_LEN, 1, NULL, 0);

    // Dump resulting encrypted data to console
    user_serial_dump("Dec ", aes_result, KEY_LEN, true);

    // Decrypt the previously encrypted data using key (to get back to original cleartext data)
    aes_operation(aes_key, KEY_LEN, aes_result, KEY_LEN, aes_in, KEY_LEN, 0, NULL, 0);

    // Dump decrypted data to console (will match original cleartext data)
    user_serial_dump("Enc ", aes_in, KEY_LEN, false);

}
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# AES Encryption demo

The application should dump the following on boot-up:



```
Data: Dialog Semi 2017
KEY : 0x53 0x69 0x6E 0x67 0x6C 0x65 0x20 0x62 0x6C 0x6F 0x63 0x6B 0x20 0x6D 0x73 0x67
Dec : 0xC9 0x11 0xEC 0x59 0xA4 0x0C 0x8C 0x55 0x3C 0xC7 0x6E 0xF9 0xDE 0x97 0x96 0x29
Enc : Dialog Semi 2017
```

© Copyright 2017 Dialog Semiconductor. All Rights Reserved

# Reference

**Reference**

- http://support.dialog-semiconductor.com/connectivity
- https://developer.bluetooth.org/gatt/Pages/default.aspx
- https://www.bluetooth.com/specifications/adopted-specifications
- https://www.wikiwand.com/en/Universally_unique_identifier

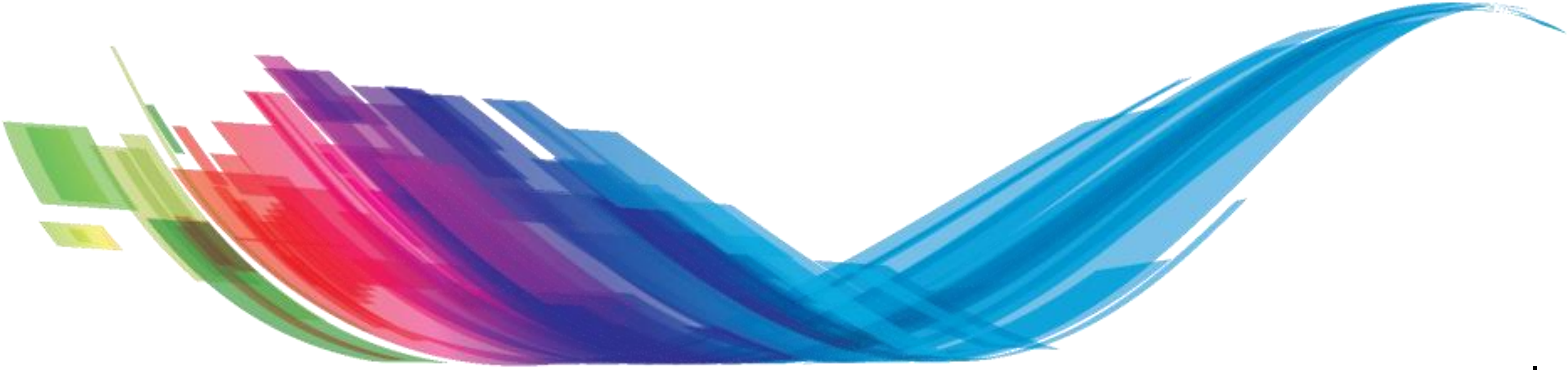
# What’s next

## For more …

- **What’s next …**
  - Please follow the other tutorials based on –
    - **SDK 5.0.x** for **DA14580/1/2/3** development OR
    - **SDK 6.0.x** for **DA14585/6** development
  - See **Reference** section of this training slide
  - Learn about Dialog BLE chip **differences** at a glance from –

    https://support.dialog-semiconductor.com/connectivity/products

# The Power To Be…



…personal
…portable
…connected