

# DemoKey 2.4



This product is subject to EU export restrictions according to Council Regulation (EC) No. 428/2009, dual-use control category 5D002.

## Data Sheet

**Doc version** 1.1
**Status** Approved
**Reference** IID-DK2-4-DS

**For internal use by Customer only**
**Confidential**

**www.intrinsic-id.com**

This product is subject to EU export restrictions according to Council Regulation (EC) No. 428/2009, dual-use control category 5D002.

This document contains information which is proprietary and confidential to Intrinsic ID B.V. and is intended for internal use only. The document is provided with the express understanding that the recipient will not divulge its content to other parties or otherwise misappropriate the information contained herein. Please destroy this document if you are not the intended recipient. Thank you.

Copyright in this document rests with Intrinsic ID B.V. Reproduction or publication in any medium of this document, in whole or in part, is expressly prohibited without the prior written permission of Intrinsic ID. Intrinsic ID reserves the right to make any changes to this document without prior notice. The contents of this document is provided AS-IS and without any warranties or guarantees as to accuracy or completeness. Receipt or possession of this document conveys no license under any patent or other intellectual property right of Intrinsic ID.

Intrinsic ID®, QuiddiKey®, QuiddiCard®, BroadKey™, DemoKey™, Citadel™, Spartan™, Confidentio™, Fuzzy ID™ and other designated brands included herein are trademarks of Intrinsic ID B.V. All other trademarks are the property of their respective owners.

This product contains code which is copyright 2014 Kenneth MacKay and licensed under the BSD license:

> Copyright (c) 2014, Kenneth MacKay
> All rights reserved.
>
> Redistribution and use in source and binary forms, with or without modification,
> are permitted provided that the following conditions are met:
> - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# History Information

| Version | Date | Change Description | Modified by: | Reviewed by: |
|---|---|---|---|---|
| 1.0 | 2018-07-17 | Initial version for DemoKey 2.4 | RS | DA, AS, PB |
| 1.1 | 2018-09-04 | Minor update | AC | RS |

# Table of Contents

# 1. Introduction

## 1.1. Document Objective

BroadKey is a software IP solution representing Intrinsic ID's flagship product line for secret key storage and cryptographic operations. DemoKey is a demo version of BroadKey. It has the same functionality and API's as BroadKey, but is intended for demonstration purposes only. Its security properties have been modified, to prevent it from being used in a actual product. See paragraph 1.3.1 for details.

This document explains the use and interface of the BroadKey software library and is mainly intended for SW developers deploying BroadKey in their application project. Targeted readers are expected to understand the basics of embedded SW programming.

This data sheet subsequently presents: the available configurations of BroadKey, and a description of the functionality which they offer (Section 2), the SW programming interface of the BroadKey module (Section 3), and some brief guidelines for integrating BroadKey in a product (Section 4). As a reference, Appendix A presents some code examples demonstrating typical deployments of BroadKey.

This data sheet is valid for the following product version: **DemoKey 2.4**, i.e. a call to the **bk_get_product_info** function (See Section 3.5.1.1) should return:

- product_id = 0x44 (character 'D' indicating DemoKey)
- major_version = 0x02
- minor_version = 0x04

NOTICE: This data sheet describes BroadKey in the sections without colored background. The sections with this gray colored background, describes the changes made for DemoKey.

## 1.2. Definitions, Acronyms and Abbreviations

| | |
|---|---|
| AC | Activation Code |
| API | Application Programming Interface |
| bk_ | BroadKey (as prefix in function/variable names) |
| ECC | Elliptic Curve Cryptography |
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| ECDH | Elliptic Curve Diffie-Hellman |
| EEPROM | Electrically Erasable Programmable Read-Only Memory |
| IID_ | Intrinsic ID (as prefix in return codes) |
| IP | Intellectual Property |
| k | x1,000 |
| M | x1,000,000 |
| MCU | MicroController Unit |
| N.A. | Not Available |

| | |
|---|---|
| NIST | National Institute of Standards and Technology (US agency) |
| NVM | Non-Volatile Memory |
| PUF | Physical Unclonable Function |
| SD | Start-up Data (of uninitialized SRAM) |
| SRAM | Static Random Access Memory |
| Vdd | Supply Voltage |

## 1.3. Product Brief

### 1.3.1. DemoKey

DemoKey is a software demonstration library, demonstrating the capabilities and interfaces of BroadKey (section 1.3.2). The goal of DemoKey is specifically for purposes of demonstration, pre-integration, functionality testing, etc.

**DemoKey is explicitly <u>not</u> intended to be used for the following purposes:**

- **Deployment in actual products, security.**
- **Use for security purposes, even outside a deployed product. Please take care that, even in demonstration or testing scenarios, DemoKey is not (inadvertently) used to process actual secrets used in real-life applications.**
- **Profiling of operations in terms of performance, memory use, code size, etc.**
- **Reliability assessments.**
- **Security assessments.**

**Notice: If you want to do one of the above, please contact Intrinsic ID, BroadKey should be used.**

In terms of configurations and functional interfaces, DemoKey is completely compatible with BroadKey, so an application developed against DemoKey's API can effortlessly be switched to BroadKey. On the other hand, in terms of security strength, DemoKey is severely crippled compared to BroadKey, to the extent that it cannot be used as a secure root of trust.

### 1.3.2. BroadKey

The product brief of BroadKey is provided in this section. Please note that this description only matches DemoKey in terms of functional interfaces, certainly not in terms of security strength. Furthermore, there are some limitations in functional execution which are detailed in Section 2.

BroadKey is a software IP solution representing Intrinsic ID's flagship product line for secret key generation and storage. It offers the full benefits of an SRAM Physical Unclonable Function or SRAM PUF in an optimized and configurable module.

Depending on its configuration, BroadKey comes with the following **functional features**:

- Generation, storage and reconstruction of device-unique keys (128, 192 and 256 bit symmetric keys and NIST P-192/224/256 elliptic curve key pairs)
- Generation of random numbers seeded by device noise
- Secure key storage based on device-unique wrapping keys
- Generation and reconstruction of device-specific or random elliptic curve key pairs
- Importing of external elliptic curve key pairs as device-protected key code formats
- Exporting of elliptic curve public keys, e.g. for certification
- Elliptic-curve signature generation and verification based on protected key codes
- Elliptic-curve key agreement function based on device-protected key codes
- Elliptic-curve-based secure cryptogram generation and processing based on device-protected key codes, for secure and authenticated messaging

Its main **benefits** are:

- Protects your data with the electronic fingerprint of the chip:
  - no need to store secrets in NVM;
  - the calling application does not have to handle unprotected private keys
  - keys and secrets are not present in the system when powered off
- Cryptographically secure RNG based on chip power-up noise
- SW-only solution[1] with HW security based on standard SRAM available on target
- Cost efficient; small footprint for the specified key strength
- Easy to use and easily scalable to billions of devices
- Overall security strength up to 256 bits for the root secrets of BroadKey.[2]
- Applies improved countermeasures against side-channel attacks

BroadKey's SRAM PUF works reliably under a wide range of **operating conditions**:

- Temperatures ranging from -55°C to +150°C
- Supply voltage variation of ±20%
- Qualified on semiconductor nodes ranging from 350nm to 7nm, including low power, high speed and high density processes
- Guaranteed lifetime >25 years, or limited by microcontroller's lifetime

## 1.4. Product Use

Figure 1 shows, very basically, the block diagram of a microcontroller unit (MCU). BroadKey is an (embedded) software module which is loaded from memory (e.g. NVM) and executed by the microcontroller. Typically, the BroadKey library is integrated into the customer's SW project.

[1] BroadKey SW is being developed to meet FIPS 140-2 Appendix B "Recommended Software Development Practices". (FIPS PUB 140-2 "Security Requirement for Cryptographic Modules", https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf) Please contact Intrinsic ID sales support for a compliance overview and list of deviations.

[2] The security strength of individual functions can be lower, depending on their cryptographic properties; e.g., the security strength of the elliptic curve cryptographic functions is limited to 128 bits, when the NIST P-256 curve is used.

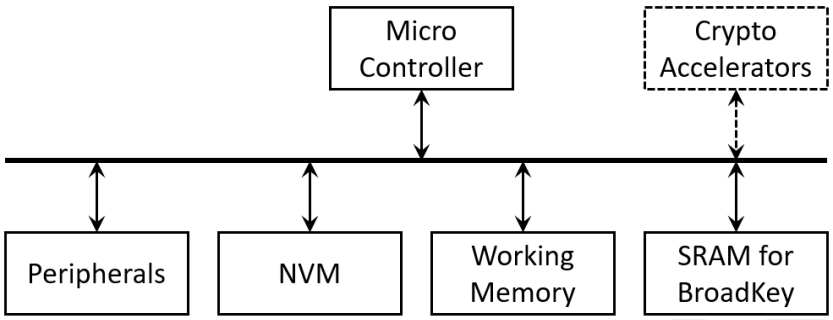


**Figure 1: Basic MCU block diagram.**

BroadKey allows for secure key extraction from unique physical properties of the underlying hardware (in particular the embedded SRAM block), instead of storing keys in tamper resistant NVM (e.g. secure EEPROM) or even hard-wiring it into the encryption core. This approach is based on the concept of Physical Unclonable Functions (PUFs). BroadKey generates an *Activation Code* (AC) which, in combination with SRAM start-up behavior, is used to reconstruct device-unique secrets for use by the system, without these secrets ever having to leave the device:

- The key storage functionality (both generation and protection) can be called via the commands of the BroadKey SW API.
- The key generation commands provide access to the device key generation functionality (both symmetric and elliptic curve). When a device key is not needed anymore by the software it can be removed from memory. When it is needed later it can be reconstructed again.
- The key protection commands provide access to the functionality for protecting the use of keys:
    - A user key protected by BroadKey can only be retrieved at a later time on the same device and will be meaningless on other devices.
    - BroadKey's elliptic-curve functionality is only called with device-protected keys. This avoids the calling application to have to handle sensitive key material. The key values themselves are only present internally to BroadKey.

The BroadKey security IP is a stateful software module, which entails that its available functionality is dependent on the state it is in. In addition, compatibility between BroadKey function calls at different points in time also depends on the *Activation Code* which is used to put BroadKey in its operational state.

BroadKey relies on and/or integrates with the standard building blocks of an MCU system (Figure 1) in the following way:

- Microcontroller:
    - Loads the BroadKey machine code and executes it.
- NVM:
    - Typically stores the BroadKey machine code (e.g. in Flash, or ROM).

  - Typically used to store the generated *Activation Code* (e.g. in Flash) in between generation and reconstruction.
  - Could be used to persistently store keys in the form of *key codes* which are protected with device-unique secrets.
  - For the storage of activation codes and key codes, no security assumptions about the NVM need to be made, since they are only intelligible to BroadKey running on the same device.
  - Optionally used for the storage of monotonic counters for particular cryptographic operations.
- SRAM/Working Memory:
  - BroadKey requires dedicated access to a block of uninitialized embedded SRAM which is used as SRAM PUF.
  - SRAM (or other embedded memory) is also used as regular working memory (stack and heap) in the conventional way.
- Peripherals:
  - BroadKey has no reliance on any peripherals.
- Crypto Accelerators:
  - BroadKey embeds all cryptographic components it needs internally in SW. In that sense, it has no reliance on the availability of dedicated crypto accelerators on the MCU.
  - However, if crypto accelerators are available on the MCU, BroadKey could leverage them to improve performance, given that they can be accessed by BroadKey's custom hardware abstraction layer.[3] BroadKey can benefit from access to accelerators for the following cryptographic operations:
    - SHA256 and/or HMAC-SHA256
    - AES





[3] Please contact Intrinsic ID sales support for questions related to integrating BroadKey with on-board crypto accelerators.

# 2. BroadKey Configurations and Function Sets

## 2.1. Configurations

**Table 1: Standard Configurations of BroadKey**

| Function Sets | | Configurations [Security Strength 128 or 256] | | |
|---|---|---|---|---|
| | | **BroadKey-Safe** | **BroadKey-Plus** | **BroadKey-Pro** |
| **Base** (see Section2.2) | *Product information (see Section 2.2.1)* | Y | Y | Y |
| | *State management (see Section 2.2.2)* | Y | Y | Y |
| **Unique device key and random value generation** (see Section 2.3) | *Unique device key generation (see Section 2.3.1)* | Y | Y | Y |
| | *Random value generation (see Section 2.3.2)* | Y | Y | Y |
| **Wrap and unwrap application keys** (see Section 2.4) | *Key wrapping (see Section 2.4.1)* | | **Y** | Y |
| **Public key management and crypto** (see Section 2.5) | *ECC key management (see Section 2.5.1)* | | | **Y** |
| | *ECC signing (see Section 2.5.2)* | | | **Y** |
| | *ECC key agreement and encryption (see Section 2.5.3)* | | | **Y** |

The BroadKey software module can be delivered in one of a number of standard configurations, as listed in Table 1. A standard BroadKey configuration is differentiated by the *function sets* which it supports, where a function set is a collection of software functions which naturally

belong together. Table 1 lists the different function sets which can be implemented by BroadKey:

- The *Base* function set, containing functions related to product information and BroadKey state management. The functions of this set are always implemented by BroadKey, regardless of its configuration.
- The *Unique device key and random value generation* function set, containing functions related to generating device-unique keys (symmetric as well as ECC private keys) and fully random bits.
- The *Wrap and unwrap application keys* function set, containing functions related to protecting and retrieving external application keys.
- The *Public key management and crypto* function set, containing functions related to managing device-unique as well as external private/public key pairs, and functions for performing public-key crypto operations based on these managed keys.

Table 1 shows how these function sets are implemented by three possible BroadKey standard configurations in an incremental manner:

1. BK-Safe, implementing the *Base* and the *Unique device key and random value generation* function sets. This configuration is indicated by the BK_CONFIGURATION_SAFE_ENABLED macro being defined (see Section 3.4.1).
2. BK-Plus, implementing the *Base*, the *Unique device key and random value generation*, and the *Wrap and unwrap application keys* function sets. This configuration is indicated by the BK_CONFIGURATION_PLUS_ENABLED macro being defined (see Section 3.4.1).
3. BK-Pro, implementing the *Base*, the *Unique device key and random value generation*, the *Wrap and unwrap application keys*, and the *Public key management and crypto* function sets. This configuration is indicated by the BK_CONFIGURATION_PRO_ENABLED macro being defined (see Section 3.4.1).

Only one of the BK_CONFIGURATION_... macros will be defined for a particular configuration.

Each of these three standard configurations is available in two variants with different *security strengths*, being 128-bit and 256-bit. This value is specified by the BK_SECURITY_SIZE_BITS constant (see Section 3.4.1) and indicates the highest security level which any of BroadKey's cryptographic functions can offer. Moreover, this value will also determine the size of some other product parameters for the given configuration, including the required SRAM PUF size and the size of the activation code.

As a reference and overview, Figure 2 visually shows the different function sets defined for BroadKey, and which functions they list. The next sections (Sections 2.2, 2.3, 2.4, and 2.5) respectively describe the functions contained in each function set. The API of these functions is defined in respectively Sections 3.5.1, 3.5.2, 3.5.3, and 3.5.4.

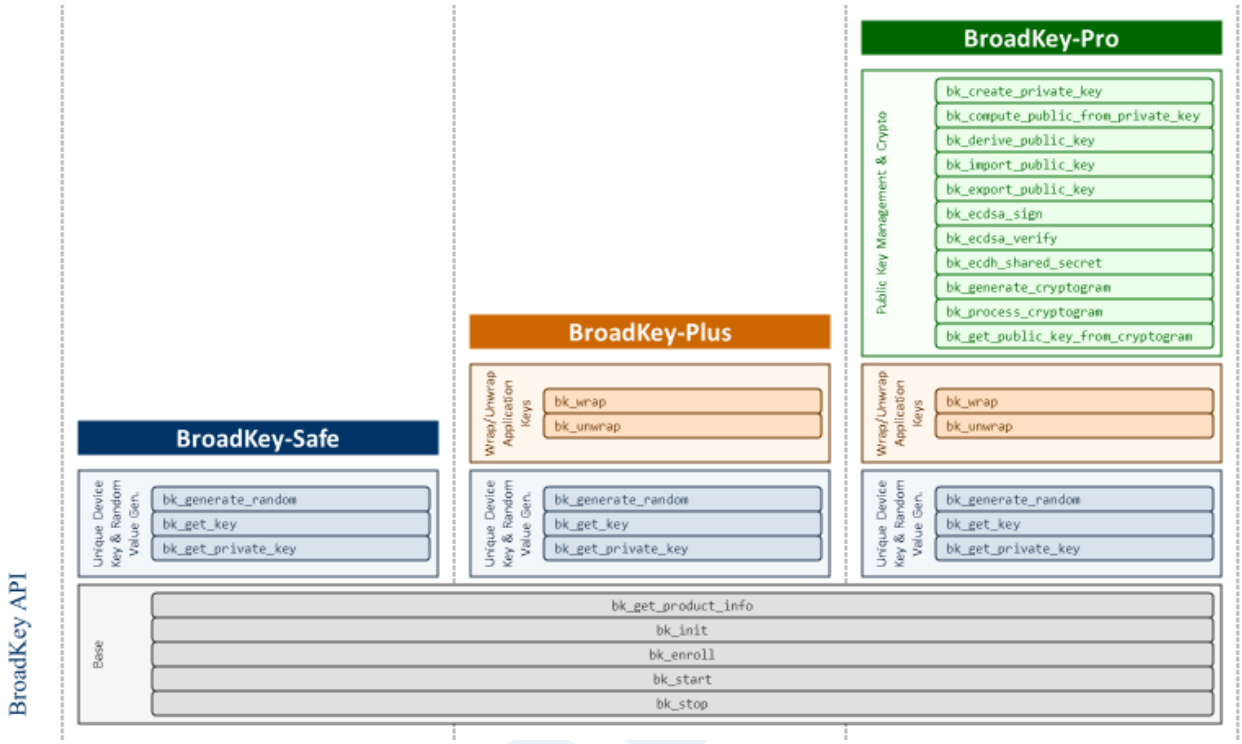


**Figure 2: Functions listed in each function set, and function sets supported by each configuration of BroadKey**

The available product configurations, and the function sets they comprise, are precisely the same for DemoKey as for BroadKey. One important side note is that, **while DemoKey is also available in 128- and 256-bit variants, this metric does not reflect DemoKey's effective security strength**. The effective security strength of DemoKey is virtually zero towards a knowledgeable attacker.

## 2.2. Base Function Set

The *Base* function set contains a number of functions which are always available in BroadKey, regardless of which configuration it is in. These *Base* functions do not provide any cryptographic functionality but are solely intended for inspecting and controlling the BroadKey module.

### 2.2.1. Product Information Function

BroadKey has a function which can be used to determine the exact version of the software library which is being used (**bk_get_product_info**, see Section 3.5.1.1 for the API). The version information is important to verify what the available functionality of the product is, and which product documentation applies to it.

For DemoKey the returned *product_id* parameter is different, to make explicitly clear that the product at hand is DemoKey, and not BroadKey. For DemoKey, the *product_id* is 0x44 (ASCII "D") while for BroadKey it is 0x42 (ASCII "B").

### 2.2.2. State Management Functions

The BroadKey security IP is a stateful software module, which entails that its available functionality is dependent on the state it is in. In addition, compatibility between BroadKey function calls at different points in time also depends on the *Activation Code* which is used to put BroadKey in its operational state. This subsection provides more information on the functional states of BroadKey, and the function calls for managing them. The full state-transition flow, and how to control it, is explained in Section 3.2.

#### 2.2.2.1. Device Power-up/Reset and Initializing BroadKey

The security of BroadKey is built upon an SRAM PUF, which consists of the start-up data (SD) of an SRAM range. After a cold reset (device power-up) or a warm reset (software/hardware reset), BroadKey always needs to condition and/or verify the SRAM SD for consistency and security. For this reason, after a cold or warm reset, the BroadKey module will be in an *Uninitialized* state. Before any other operation, one should first call **bk_init** (see Section 3.5.1.2 for the API) to bring BroadKey to the *Initialized* state.

#### 2.2.2.2. Enrolling BroadKey with Activation Code Output

After successful initialization, BroadKey can be *Enrolled* by calling **bk_enroll** (see Section 3.5.1.3 for the API). Enrollment instantiates a new *cryptographic context*[4] from the secret SRAM SD, and produces an AC which it outputs to the calling software. ***The cryptographic functionality offered by the BroadKey API (as specified by the function sets detailed in Sections 2.3, 2.4 and 2.5) only becomes available after a cryptographic context is instantiated (either through enrolling or starting BroadKey).***

It is important to note that the combination of SRAM SD *and* AC defines the instantiated cryptographic context. If, at a later point, the same context is required in order to perform compatible operations, the same AC needs to be provided to the BroadKey module on the same physical device (in a **bk_start** call, see Section 2.2.2.3). This entails that the AC needs to be stored in between usages of BroadKey, and it is the responsibility of the calling software to take care of this. The AC does not contain any confidential information and can therefore be stored publicly in a non-volatile memory, on- or off-chip, without additional protection. Evidently, the AC is also device-specific and using it on another device will result in an error. This results from the fact that the SRAM SD of every device is unique and does not match with an AC of another device. It is one of the core security features of BroadKey which protects the system against cloning and counterfeiting.

Over the lifetime of a device, **bk_enroll** needs to be called at least once to be able to use BroadKey's functions. After that, the same cryptographic context can always be re-instantiated

[4] A *cryptographic context* is BroadKey's internal representation of the device-unique cryptographic data that is derived from the SRAM start-up data (i.e. the PUF), and which is used as the root secret of BroadKey's cryptographic functionality.

using **bk_start**. Technically, **bk_enroll** can be called multiple times, but note that this will result in different ACs and contexts which are separated and hence incompatible with each other. Multiple calls to **bk_enroll** over a device's lifetime are only meaningful if the above is desired behavior.

#### 2.2.2.3. Starting BroadKey with Activation Code Input

After successful initialization, BroadKey can be *Started* by calling **bk_start** (see Section 3.5.1.4 for the API). Starting re-instantiates a cryptographic context which was previously generated with **bk_enroll**, from the secret SRAM SD and the AC output by by **bk_enroll**. After the cryptographic context is re-instantiated, BroadKey's cryptographic functionality (as specified by the function sets detailed in Sections 2.3, 2.4 and 2.5) becomes available.

Over the lifetime of a device, **bk_start** needs to be called every time BroadKey's cryptographic functionality is needed, after each device power-up or reset and **bk_init**, or after each call to **bk_stop** (see Section 2.2.2.4). It is the responsibility of the calling software to retrieve the correct AC from storage and provide it as an input to **bk_start**.

For BroadKey, the reliability of **bk_start** on a correct device is extremely high, with a failure rate under normal circumstances $<< 10^{-9}$, and even under extreme circumstances still $< 10^{-9}$. For DemoKey, the reliability is somewhat reduced to a failure rate $< 10^{-4}$ under normal circumstances. This level of reliability is totally sufficient for demonstration purposes.

Likewise, for BroadKey the false acceptance rate of **bk_start** is extremely low, i.e. the probability of **bk_start** succeeding on a wrong device is negligibly small. For DemoKey, this probability is somewhat elevated, but still $< 10^{-4}$ under normal circumstances. Again, for demonstration purposes this more than suffices.

#### 2.2.2.4. Stopping BroadKey

BroadKey also provides a **bk_stop** function (see Section 3.5.1.5 for the API) which has the opposite effect of **bk_start** and **bk_enroll**; i.e. it un-instantiates the cryptographic context and removes (zeroizes) all internal secrets related to it from its internal memory. Hence, after **bk_stop** BroadKey's cryptographic functionality is effectively *Stopped*. To make it available again, **bk_start** can be called again. After a call to **bk_stop**, **bk_enroll** is no longer available (see Section 3.2 for the full state transition diagram).

## 2.3. Unique Device Key and Random Value Generation Function Set

The *Unique device key and random value generation* function set contains a number of functions for generating symmetric device keys and random numbers. Device keys are random and device-unique, but can be regenerated at a later time since they are derived from the reproducible secret extracted from the SRAM PUF, whereas random values are completely unpredictable and irreproducible over time. This function set constitutes the most basic cryptographic functionality which can be expected from a PUF-based security module.

### 2.3.1. Unique Device Key Generation

#### 2.3.1.1. Generating Device-Unique Symmetric Keys

BroadKey can output a range of symmetric device keys by calling the **bk_get_key** function (see Section 3.5.2.2 for the API). Based on the provided *key_type* parameter, **bk_get_key** will generate keys with different lengths for symmetric cryptographic primitives (supported lengths are 128, 192 and 256 bits, where the latter two are only available in configurations with security strength 256). For each key length, up to 256 different and independent device keys can be produced with the same length, by changing the provided *index* parameter.

A device key is unpredictable and unique per device *and* per instantiated cryptographic context, and moreover cryptographically separated from other keys and secrets used by BroadKey. This entails that re-enrolling the same device (by calling **bk_enroll** more than once), or starting BroadKey with different activation codes, will result in different device keys which are unrelated and incompatible. On the other hand, when the same cryptographic context is instantiated on the same device, device keys are perfectly reconstructible, i.e. in that situation calling **bk_get_key** with the same parameters will always return the same key value.

Contrarily to BroadKey, for DemoKey device-unique symmetric keys are neither strongly cryptographically separated, nor fully unpredictably random. Nonetheless, even for DemoKey, device-unique symmetric keys are at first sight seemingly random and with high probability unique.

#### 2.3.1.2. Generating Device-Unique and Random Elliptic Curve Private Keys

The **bk_get_private_key** function (see Section 3.5.2.3 for the API) generates elliptic curve private keys. These private keys can come from two different sources:

1. Device-unique keys, generated from the device's secret fingerprint, which are always reconstructible on the same device in the same cryptographic context. This is similar to symmetric device keys as generated by **bk_get_key** (see Section 3.5.2.2), but with the right mathematical properties to be an elliptic curve private key.
2. Random keys, generated randomly from BroadKey's internal random number generator. This is similar to a call to **bk_generate_random** (see Section 3.5.2.1), but with the right mathematical properties to be an elliptic curve private key.

The calling application can optionally include external information in the private key derivation by providing it through a *usage_context* input. For both device-unique and random private keys, a calling application can use this usage context input to contribute additional context information or entropy to the key generation process. Moreover, for device private keys this usage context allows the calling application to generate multiple different private keys over the same elliptic curve.

It is important to note that all derived private keys are cryptographically separated, e.g. two derived device keys over the same curve with different usage context will be completely different, also two derived device keys with the same usage context over consecutive curves will be completely unrelated.

If **bk_get_private_key** is called twice within the same cryptographic context to generate two private key codes for *device-unique* private keys, and if all input parameters (curve, usage context) are equal for both calls, the same device-unique private key will be returned. With that in mind, a calling application does not need to store device-unique private keys, since they can always be recreated from the same inputs. For *random* private keys this will *not* be the case.

Similarly, as for device-unique symmetric keys, device unique and random elliptic curve private keys are neither strongly cryptographically separated, nor fully unpredictably random in DemoKey as compare to BroadKey.

In addition, in relation to its low overall security strength, DemoKey will only generate elliptic curve private keys from a severely limited subset of all possible private keys for a given curve. This is reflected by the following fixed structure of all private keys generated by DemoKey: regardless of the curve choice,

- the least significant byte of a private key will always be 0xFF,
- the second to fifth least significant bytes of a private key will be seemingly random, and
- all other (most-significant) bytes of a private key will always be 0x00.

This structure ensures that the generated private key will always be valid for the supported curves.

### 2.3.2. Random Value Generation Function

BroadKey can generate arrays of random bytes by calling the **bk_generate_random** function (see Section 3.5.2.1 for the API). Random bytes are generated internally using a cryptographically secure pseudorandom number generator which is seeded with entropy originating from random power-up noise on the device. Arbitrary length random arrays can be generated.

This functionality can also be used to generate random symmetric keys (as opposed to device symmetric keys). Note that, contrary to device keys, random keys generated in this way can never be reconstructed by BroadKey, and need to be stored by the calling application if they are needed at a later time.

Contrarily to BroadKey, DemoKey does not generate random bytes with a *cryptographically secure* pseudorandom number generator. The bytes are seemingly random, but not necessarily fully unpredictably random.

## 2.4. Wrap and Unwrap Application Keys Function Set

The *Wrap and unwrap application keys* function set contains a number of functions for protecting (wrapping) and retrieving (unwrapping) external application keys based on a device-unique PUF-derived secret. Keys in their *wrapped* form are referred to as *key codes*. This function set is a natural extension to BroadKey's cryptographic functionality, which allows an application to work with externally generated keys, yet benefit fully from the protection offered by the device-unique PUF secret.

### 2.4.1. Key Wrapping Functions

BroadKey's functions for protecting and retrieving provided application keys are respectively **bk_wrap** and **bk_unwrap** (see Sections 3.5.3.1 and 3.5.3.2 for the API). A call to **bk_wrap** securely wraps a provided application key value into a key code. The key code fully protects the wrapped key value (both in confidentiality and integrity), and can hence be treated without any further protection, e.g. can be stored in a non-volatile memory, on- or off-chip, without additional protection. A call to **bk_unwrap** on the same device instantiated with the same cryptographic context will successfully retrieve the originally wrapped key value from the key code. It is the responsibility of the calling software to store and retrieve key codes in between **bk_wrap** and **bk_unwrap**.

Since **bk_wrap** and **bk_unwrap** internally operate with device-unique secrets, key codes are only intelligible to BroadKey running on the same device *and* in the same cryptographic context, and completely unintelligible and meaningless otherwise. This entails that re-enrolling the same device (by calling **bk_enroll** more than once), or starting BroadKey with different activation codes, will result in the inability to retrieve keys from key codes which were produced with a different cryptographic context.

In addition, **bk_unwrap** can *only* unwrap key codes which were successfully wrapped by a call to **bk_wrap**. BroadKey can have other cryptographic functions which output key codes (e.g. **bk_create_private_key**, **bk_compute_public_from_private_key** and **bk_import_public_key**), but these *cannot* be unwrapped by **bk_unwrap**. Instead, these special-purpose key codes should be used as inputs to the dedicated public-key cryptographic functions.

The functionality of the key wrapping functions is the same for DemoKey as it is for BroadKey. However, note that the effective security provided by a DemoKey key code is limited by the effective overall security strength of DemoKey, which is very low.

**Don't rely on DemoKey's wrapping functionality to protect secrets used in real-life applications!**

## 2.5. Public Key Management and Crypto Function Set

The *Public key management and crypto* function set contains a number of functions for combining the strength of PUF-derived device-unique secrets with public-key cryptography, in particular elliptic-curve cryptography (ECC). This function set allows an application to:

- Derive device-unique ECC private keys
- Protect and manage ECC private and/or public keys based on device-unique secrets
- Call basic ECC crypto functions (ECDSA, ECDH, en/decrypt) with protected private and/or public key codes, such that no sensitive information needs to be passed as a parameter in a function call.

This function set constitutes a more advanced extension of BroadKey's cryptographic functionality.

### 2.5.1. ECC Key Management Functions

BroadKey has a set of functions for dealing with public/private key pairs based on elliptic curve cryptography. The basic functionality of these key management functions is the transformation of private and/or public keys from different sources (device keys, random keys or external keys) into dedicated key codes which can be used for the elliptic curve cryptography functions described in Sections 2.5.2 and 2.5.3. These key codes are secure and device-bound representations of the actual key values and associated data. As a result, applications calling the elliptic curve cryptography functions do not need to handle or pass sensitive data.

#### 2.5.1.1. Managing ECC Private Keys as Private Key Codes

The **bk_create_private_key** function (see Section 3.5.4.1 for the API) generates *private key codes* containing elliptic curve private keys. These private keys can come from three different sources:

1. Device keys which are always reconstructible on the same device in the same cryptographic context, similarly to symmetric device keys as generated by bk**_get__key** (see Section 3.5.2.2), but with the right mathematical properties to be an elliptic curve private key.
2. Random keys, generated randomly from BroadKey's internal random number generator similar to a call to **bk_generate_random** (see Section 3.5.2.1), but with the right mathematical properties to be an elliptic curve private key.
3. External private keys which are provided as an input in the API call. Private key codes for different elliptic curves are supported (supported curves are NIST/SEC P-192, P-224 and P-256).

In addition to the private key values, private key codes generated by **bk_create_private_key** also contain additional input information related to the private key:

- The elliptic curve over which the contained private key is defined.
- Purpose flags which indicate the allowed usage of the contained private key.

These two fields will determine the allowed use of the private key code. A private key code can only be used in an elliptic curve crypto function if its purpose flag allows it (e.g. a private key which is only purposed for decryption/key agreement cannot be used for signing, and vice versa), and only for operations over the same elliptic curve it was defined on.

Device private keys and random private keys are derived internally by **bk_create_private_key**, respectively from the device fingerprint or from the device noise entropy, and from the provided curve and purpose inputs. It is important to note that all derived private keys are cryptographically separated, e.g. two derived device keys over the same curve with different purpose flags will be completely different, also two derived device keys with the same purpose flags over consecutive curves will be completely unrelated.

In addition, the calling application can optionally include external information in the private key derivation by providing it through a *usage_context* input. For both device and random private keys, a calling application can use this usage context input to contribute additional context information or entropy to the key generation process. Moreover, for device private keys

this usage context allows the calling application to generate multiple different private keys for which all other parameters (curve, purpose) are equal. For externally provided private keys, the usage context input is not used, since the private key is not derived internally.

If **bk_create_private_key** is called twice within the same cryptographic context to generate two private key codes for *device* private keys, and if all input parameters (curve, purpose flags, usage context) are equal for both calls, both returned private key codes will contain the same device private key. With that in mind, a calling application does not need to store device private key codes, since they can always be recreated from the same inputs. For *random* private keys this will *not* be the case.

Similarly, as for the plain generation of elliptic curve private keys (see Section 2.3.1.2) the management of private keys as private key codes is restricted:

- The generation of private key codes from device-unique and random sources, has the same structure for its generated private keys as **bk_get_private_key** (see Section 2.3.1.2); i.e., they are always of the form {0x00, 0x00, …, 0x00, 0xX1, 0xX2, 0xX3, 0xX4, 0xFF}, where 0xX1-4 represent four seemingly random bytes which are derived from respectively a device-unique secret or a random source.
- The generation of private key codes from externally provided private keys puts a similar format restriction on the provided private key; i.e. only external private keys of the form {0x00, 0x00, …, 0x00, 0xX1, 0xX2, 0xX3, 0xX4, 0xX5} are accepted, where 0xX1-5 represent five arbitrary bytes. Presenting an external private key which does not meet this format will result in the function failing to succeed.

Similarly, as for **bk_get_key** and **bk_get_private_key**, keys generated by this function are not strongly cryptographically separated, nor fully unpredictably random.

### 2.5.1.2. Managing ECC Public Keys as Public Key Codes

The **bk_compute_public_from_private_key** function (see Section 3.5.4.2 for the API) takes as input a private key code and generates the corresponding public key code; i.e. a key code which contains the elliptic curve public key counterpart to the private key contained in the private key code. The additional information (curve, purpose) contained in the private key code will be copied to the public key code.

The **bk_derive_public_key** function (see Section 3.5.4.3 for the API) directly derives the public key value corresponding to a provided private key. This provides the same functionality as **bk_compute_public_from_private_key**, but avoids the need to present private and public keys as key codes. This is convenient for applications which do not need to/want to work with BroadKey's key code formats. On the other hand, this function requires private and public keys to be presented on the API in an unprotected format. When using this function, it is hence up to the calling application to ensure the protection of these values.

The **bk_import_public_key** function (see Section 3.5.4.4 for the API) imports an external elliptic-curve public key value to a public key code format which can be used with the elliptic

curve cryptography functions described in Sections 2.5.2 and 2.5.3. This function is needed, e.g. when a public key value from an external certificate is needed as an input for BroadKey's elliptic curve cryptography functions described in Sections 2.5.2 and 2.5.3.

The **bk_export_public_key** function (see Section 3.5.4.5 for the API) exports an elliptic curve public key value from its public key code container, for use by operations external to BroadKey. This function is needed, e.g. when a device public key needs to be output for certification by an external party.

For the **bk_derive_public_key** function, the same restrictions on the format of provided private keys apply as for **bk_create_private_key** with an externally provided private key (see Section 2.5.1.1).

### 2.5.2. ECC Signing Functions

BroadKey has a set of functions for performing basic ECDSA signature generation and verification (respectively **bk_ecdsa_sign** and **bk_ecdsa_verify**, see Sections 3.5.4.6 and 3.5.4.7 for the API). The **bk_ecdsa_sign** function takes as input a message string, or a message hash, and computes an ECDSA signature on it based on a provided private key (code). The **bk_ecdsa_verify** function takes as input a message string, or a message hash, a signature value, and a public key code, and verifies whether the provided signature matches the message under the given public key.

These functions take private and/or public *key codes* as inputs, instead of key values directly. This entails that an application calling these functions never needs to pass possibly sensitive key data, only key codes which are protected both in confidentiality (important for private keys) and integrity (important for public keys). An important consideration is that the key codes will also contain purpose flags which indicate whether they are allowed to be used for ECDSA-based operations, or not.

For **bk_ecdsa_sign** and **bk_ecdsa_verify**, the maximum length of a message which can be signed or verified by DemoKey is limited to 64 bytes. DemoKey will not accept longer messages.

### 2.5.3. ECC Key Agreement and Encryption Functions

BroadKey has a set of functions for performing basic ECDH-based key agreement between two parties, and in extension for setting up a secure messaging protocol between a sender and a receiver, based on cryptograms that are protected with an ECDH-derived shared secret.

These functions take private and public *key codes* as inputs, instead of key values directly. This entails that an application calling these functions never needs to pass possibly sensitive key data, only key codes which are protected both in confidentiality (important for private keys) and integrity (important for public keys). An important consideration is that the key codes will also contain purpose flags which indicate whether they are allowed to be used for ECDH-based operations, or not.

### 2.5.3.1. ECDH Key Agreement

BroadKey has a function (**bk_ecdh_shared_secret**, see Section 3.5.4.8 for the API) for generating a shared secret value from a private key code and a public key code based on the ECDH algorithm. The returned shared secret can be processed into one or more shared secret keys by the calling application, using the proper key derivation functions.

### 2.5.3.2. ECDH-based Cryptogram Generation and Processing

BroadKey has a set of functions for enabling a secure one-pass messaging protocol based on hybrid elliptic curve cryptography. The conceptual idea is that a sending system, embedding BroadKey (or a compatible implementation), can transform a plaintext message into a secure cryptogram using its own elliptic curve private key and a receiver's elliptic curve public key. The corresponding receiving system can use BroadKey (or a compatible implementation) to unpack the message from the cryptogram using its own elliptic curve private key (corresponding to the public key used by the sender), and the sender's public key (corresponding to the private key used by the sender). When used correctly, the basic properties achieved by this one-pass protocol are:

- The message contained in a cryptogram is **confidential**, i.e. the message is unintelligible except to the sender and the intended receiver.
- The cryptogram is **integrity-protected**, i.e. the receiver can verify that cryptogram is exactly like the sender created it.
- The receiver can **authenticate the sender** of the message in the cryptogram, i.e. the receiver can obtain proof that the message originated from the sender.
- In addition, given proper use of the functions, it can be verified that cryptograms are **non-replayable**, i.e. the receiver can detect whether a cryptogram has been received before, or whether it is replayed or received out-of-order.

The fact that these cryptogram functions constitute a one-pass secure protocol with these security properties, makes them well suited for use as import/export functions of external secrets (e.g. in a key provisioning scheme) or as a payload protection mechanism (e.g. in a secure update flow).

The **bk_generate_cryptogram** function (see Section 3.5.4.9 for the API) takes as input a message plaintext, the sender's private key code and the receiver's public key code, and creates a cryptogram from it. Two additional inputs are:

- A message counter value, needed for keeping track of the order of produced cryptograms. The **bk_generate_cryptogram** function updates this counter and returns the new counter as an output. It is up to the calling application to provide reliable storage and update mechanisms for this counter value.
- The cryptogram type to be used. BroadKey provides two different cryptogram types with slightly different efficiency and security properties.
    - Cryptogram type = "BK_ECC_CRYPTOGRAM_TYPE_ECDH_STATIC" This is the baseline cryptogram type which achieves the basic security properties listed above. Since it is solely based on static key pairs on both

sending and receiving side, this cryptogram type cannot achieve non-repudiation, nor forward secrecy.

- Cryptogram type = "BK_ECC_CRYPTOGRAM_TYPE_ECDH_EPHEMERAL"
  This is an extension on the baseline type in which the sending side uses an ephemeral key pair. It achieves the basic security properties listed above, and in addition it achieves forward secrecy with respect to loss of the sender's long-term private key. It does not achieve forward secrecy with respect to the receiver's private key, this cannot be accomplished in a one-pass protocol. It also still does not achieve non-repudiation. If non-repudiation is required, the cryptogram functionality can be combined with the ECDSA signing functionality.

The **bk_process_cryptogram** function (see Section 3.5.4.10 for the API) performs the opposite operation of **bk_generate_cryptogram**. It takes as input an earlier produced cryptogram, the receiver's private key code and the sender's public key code, and returns the contained plaintext message. Additionally:

- The **bk_process_cryptogram** function takes as input the last received message counter value. If the message counter of the received cryptogram does not exceed this counter input, it means that the cryptogram has been received out of order (e.g. replayed) and it will not be accepted. If the cryptogram is accepted and successfully processed, the **bk_process_cryptogram** updates this counter and returns the new counter as an output. It is up to the calling application to provide reliable storage and update mechanisms for this counter value.
- The **bk_process_cryptogram** function outputs the type of a successfully processed cryptogram (see above). Based on this type, the calling application knows which security properties are obtained by the cryptogram.

The **bk_get_public_key_from_cryptogram** helper function (see Section 3.5.4.11 for the API) optionally aids a receiver of a BroadKey cryptogram in obtaining the public key used by the sender, for further verification.

The functionality of the cryptogram functions is the same for DemoKey as it is for BroadKey. However, note that the effective security provided by a DemoKey cryptogram is limited by the effective overall security strength of DemoKey, which is very low.

**Don't rely on DemoKey's cryptogram functionality to protect secrets used in real-life applications!**

## 2.6. Profiling Information

### 2.6.1. BroadKey Performance and Stack Usage

Table 2 and Table 3 show the performance and memory use of BroadKey's functions when profiled on respectively an ARM Cortex-M4 CPU and an ARM Cortex-M0 CPU. A relevant subset of all possible function options was profiled: curve P-256 for the ECC functions, and

32 bytes for all variable-length inputs and/or outputs. The configuration of BroadKey that was used for this profiling was BroadKey-Pro with both 128-bit and 256-bit security strengths.

**Table 2: Profiling of BroadKey functions on an ARM Cortex-M4 CPU[5]**

| **Configuration Security Strength:** | **128 bit** | | **256 bit** | |
|---|---|---|---|---|
| *Function Call* | *Clock Cycles* | *Stack Usage (bytes)* | *Clock Cycles* | *Stack Usage (bytes)* |
| ***Base* function set** | | | | |
| **bk_init** | ~ 75k | ~ 0.80k | ~ 125k | ~ 0.80k |
| **bk_enroll** | ~ 405k | ~ 1.5k | ~ 620k | ~ 1.9k |
| **bk_stop** | ~ 3k | 8 | ~ 6k | 8 |
| **bk_start** (12.5% SRAM PUF bit errors) | ~ 695k | ~ 2.2k | ~ 1.1M | ~ 2.8k |
| ***Unique device key and random value generation* function set** | | | | |
| **bk_generate_random** (32 bytes) | ~ 115k | ~ 0.90k | ~ 115k | ~ 0.95k |
| **bk_get_key** (128-bit key) | ~ 35k | ~ 0.99k | ~ 40k | ~ 1.0k |
| **bk_get_key** (256-bit key) | *(Not available for 128-bit security strength)* | | ~ 40k | ~ 1.0k |
| **bk_get_private_key** (curve P-256, PUF-derived key) | ~ 215k | ~ 1.3k | ~ 225k | ~ 1.3k |
| **bk_get_private_key** (curve P-256, randomly generated key) | ~ 270k | ~ 1.4k | ~ 275k | ~ 1.4k |
| ***Wrap and unwrap application keys* function set** | | | | |
| **bk_wrap** (32 bytes) | ~ 185k | ~ 1.8k | ~ 195k | ~ 1.9k |
| **bk_unwrap** (32 bytes) | ~ 185k | ~ 1.8k | ~ 195k | ~ 1.8k |
| ***Public key management and crypto* function set** | | | | |
| **bk_create_private_key** (curve P-256, PUF-derived key) | ~ 395k | ~ 2.0k | ~ 415k | ~ 2.1k |

[5] These performance results are measured on a platform with an ARM Cortex-M4F CPU (STM32L476RG) running at 80MHz. BroadKey was compiled with GCC 5.2 with the following compiler options: Defined symbols: -DNDEBUG, Debug level: Off and Optimization level: Os. These results were obtained with assembly code optimizations which are specific to the MCU architecture used for profiling. For profiling results on other architectures, please contact Intrinsic ID sales support.

| Configuration Security Strength: | 128 bit | | 256 bit | |
|---|---|---|---|---|
| *Function Call* | *Clock Cycles* | *Stack Usage (bytes)* | *Clock Cycles* | *Stack Usage (bytes)* |
| **bk_create_private_key** (curve P-256, randomly generated key) | ~ 395k | ~ 2.0k | ~ 415k | ~ 2.1k |
| **bk_create_private_key** (curve P-256, user-provided key) | ~ 185k | ~ 2.0k | ~ 195k | ~ 2.1k |
| **bk_compute_public_from_private_key** (curve P-256) | ~ 6.6M | ~ 1.9k | ~ 6.6M | ~ 2.0k |
| **bk_import_public_key** (curve P-256) | ~ 200k | ~ 2.1k | ~ 215k | ~ 2.1k |
| **bk_export_public_key** (curve P-256) | ~ 195k | ~ 1.9k | ~ 205k | ~ 1.9k |
| **bk_derive_public_key** (curve P-256) | ~ 6.3M | ~ 0.85k | ~ 6.3M | ~ 0.85k |
| **bk_ecdsa_sign** (curve P-256, 32 bytes) | ~ 7.3M | ~ 2.2k | ~ 7.4M | ~ 2.3k |
| **bk_ecdsa_verify** (curve P-256, 32 bytes) | ~ 7.8M | ~ 2.5k | ~ 7.9M | ~ 2.6k |
| **bk_ecdh_shared_secret** (curve P-256) | ~ 6.6M | ~ 1.9k | ~ 6.7M | ~ 2.0k |
| **bk_generate_cryptogram** (curve P-256, static type, 32 bytes) | ~ 13.1M | ~ 2.9k | ~ 13.1M | ~ 3.0k |
| **bk_process_cryptogram** (curve P-256, static type, 32 bytes) | ~ 6.8M | ~ 2.8k | ~ 6.9M | ~ 2.9k |
| **bk_generate_cryptogram** (curve P-256, ephemeral type, 32 bytes) | ~ 25.8M | ~ 2.9k | ~ 25.9M | ~ 3.0k |
| **bk_process_cryptogram** (curve P-256, ephemeral type, 32 bytes) | ~ 13.1M | ~ 2.8k | ~ 13.1M | ~ 2.9k |
| **bk_get_public_key_from_cryptogram** (curve P-256, 32 bytes) | ~ 0.55k | 32 | ~ 0.55k | 32 |

**Table 3: Profiling of BroadKey functions on an ARM Cortex-M0 CPU[6]**

| Configuration Security Strength: | 128 bit | | 256 bit | |
|---|---|---|---|---|
| *Function Call* | *Clock Cycles* | *Stack Usage (bytes)* | *Clock Cycles* | *Stack Usage (bytes)* |
| ***Base* function set** | | | | |

[6] These performance results are measured on a platform with an ARM Cortex-M0 CPU (NXP-LPCXpresso11U37H) running at 48MHz. BroadKey was compiled with GCC 5.2 with the following compiler options: Defined symbols: -DNDEBUG, Debug level: Off and Optimization level: Os. These results were

| Configuration Security Strength: | 128 bit | | 256 bit | |
|---|---|---|---|---|
| *Function Call* | *Clock Cycles* | *Stack Usage (bytes)* | *Clock Cycles* | *Stack Usage (bytes)* |
| **bk_init** | ~ 185k | ~ 0.85k | ~ 325k | ~ 0.85k |
| **bk_enroll** | ~ 975k | ~ 1.5k | ~ 1.4M | ~ 1.8k |
| **bk_stop** | ~ 7k | 16 | ~ 9k | 16 |
| **bk_start** (12.5% SRAM PUF bit errors) | ~ 1.7M | ~ 2.2k | ~ 2.4M | ~ 2.8k |
| ***Unique device key and random value generation* function set** | | | | |
| **bk_generate_random** (32 bytes) | ~ 275k | ~ 0.95k | ~ 275k | ~ 0.95k |
| **bk_get_key** (128-bit key) | ~ 80k | ~ 1.0k | ~ 85k | ~ 1.0k |
| **bk_get_key** (256-bit key) | *(Not available for 128-bit security strength)* | | ~ 85k | ~ 1.0k |
| **bk_get_private_key** (curve P-256, PUF-derived key) | ~ 405k | ~ 1.3k | ~ 415k | ~ 1.4k |
| **bk_get_private_key** (curve P-256, randomly generated key) | ~ 535k | ~ 1.4k | ~ 540k | ~ 1.4k |
| ***Wrap and unwrap application keys* function set** | | | | |
| **bk_wrap** (32 bytes) | ~ 460k | ~ 1.8k | ~ 480k | ~ 1.9k |
| **bk_unwrap** (32 bytes) | ~ 460k | ~ 1.8k | ~ 480k | ~ 1.9k |
| ***Public key management and crypto* function set** | | | | |
| **bk_create_private_key** (curve P-256, PUF-derived key) | ~ 865k | ~ 2.1k | ~ 895k | ~ 2.1k |
| **bk_create_private_key** (curve P-256, randomly generated key) | ~ 865k | ~ 2.1k | ~ 895k | ~ 2.1k |
| **bk_create_private_key** (curve P-256, user-provided key) | ~ 460k | ~ 2.1k | ~ 485k | ~ 2.1k |
| **bk_compute_public_from_private_key** (curve P-256) | ~ 22.4M | ~ 2.0k | ~ 22.4M | ~ 2.0k |
| **bk_import_public_key** (curve P-256) | ~ 495k | ~ 2.1k | ~ 525k | ~ 2.1k |

obtained with assembly code optimizations which are specific to the MCU architecture used for profiling. For profiling results on other architectures, please contact Intrinsic ID sales support.

| Configuration Security Strength: | 128 bit | | 256 bit | |
|---|---|---|---|---|
| *Function Call* | *Clock Cycles* | *Stack Usage (bytes)* | *Clock Cycles* | *Stack Usage (bytes)* |
| **bk_export_public_key** (curve P-256) | ~ 475k | ~ 1.9k | ~ 500k | ~ 1.9k |
| **bk_derive_public_key** (curve P-256) | ~ 21.5M | ~ 0.85k | ~ 21.4M | ~ 0.85k |
| **bk_ecdsa_sign** (curve P-256, 32 bytes) | ~ 23.6M | ~ 2.3k | ~ 23.7M | ~ 2.3k |
| **bk_ecdsa_verify** (curve P-256, 32 bytes) | ~ 26.9M | ~ 2.5k | ~ 25.9M | ~ 2.5k |
| **bk_ecdh_shared_secret** (curve P-256) | ~ 22.5M | ~ 2.0k | ~ 22.6M | ~ 2.0k |
| **bk_generate_cryptogram** (curve P-256, static type, 32 bytes) | ~ 44.3M | ~ 3.0k | ~ 44.3M | ~ 3.0k |
| **bk_process_cryptogram** (curve P-256, static type, 32 bytes) | ~ 22.9M | ~ 2.9k | ~ 23.0M | ~ 2.9k |
| **bk_generate_cryptogram** (curve P-256, ephemeral type, 32 bytes) | ~ 87.8M | ~ 3.0k | ~ 87.8M | ~ 3.0k |
| **bk_process_cryptogram** (curve P-256, ephemeral type, 32 bytes) | ~ 44.5M | ~ 2.9k | ~ 44.5M | ~ 2.9k |
| **bk_get_public_key_from_cryptogram** (curve P-256, 32 bytes) | ~ 1.1k | 40 | ~ 1.1k | 40 |

The performance and stack usage results provided for BroadKey are not applicable for DemoKey. In general, DemoKey is a bit faster and uses less memory than BroadKey, but no strong guarantees can be given. The exact profiling results of DemoKey are not important for demonstration purposes.

### 2.6.2. BroadKey Memory Requirements

Table 4 shows the static memory requirements for the different BroadKey configurations (as defined in Table 1) when targeted to an ARM Cortex-M4 CPU. This table lists consecutively the requirements for:

- The SRAM needed as PUF: this needs to be a block of uninitialized SRAM which is (preferably) dedicated to be solely accessed by the BroadKey library.
- The base working memory needed by the BroadKey library in its specified configuration.
- The code size of the BroadKey library in its specified configuration and security strength, both for the ARM Cortex M4 and M0 platforms as specified in Section 2.6.1.

**Table 4: Memory requirements of different BroadKey configurations**

| Security Strength | Configuration | PUF SRAM Size (bytes) | Static Memory Usage (bytes) | Cortex-M4 Code Size (bytes) | Cortex-M0 Code Size (bytes) |
|---|---|---|---|---|---|
| **128** | **BroadKey-Safe** | 512 | 552 | 8,082 | 8,140 |
| | **BroadKey-Plus** | 512 | 568 | 10,426 | 10,576 |
| | **BroadKey-Pro** | 512 | 568 | 21,806 | 20,956 |
| **256** | **BroadKey-Safe** | 1024 | 1,096 | 8,198 | 8,284 |
| | **BroadKey-Plus** | 1024 | 1,128 | 10,530 | 10,724 |
| | **BroadKey-Pro** | 1024 | 1,128 | 21,950 | 21,150 |

The memory requirements provided for BroadKey are not applicable for DemoKey. In general, DemoKey is a bit smaller than BroadKey, but no strong guarantees can be given. The exact profiling results of DemoKey are not important for demonstration purposes.

# 3. BroadKey Module

## 3.1. Library Files

The BroadKey library software IP is delivered as a collection of four C code header files (**iidbroadkey.h**, **iidreturn_codes.h**, **iid_configuration.h** and **iid_platform.h**) and an object file containing the compiled binary.

The programming interface is entirely defined in **iidbroadkey.h** and **iidreturn_codes.h**:

- **iidreturn_codes.h** defines the possible return codes of the callable functions, as described in Section 3.3.
- **iidbroadkey.h** (which internally includes **iid_configuration.h** and **iid_platform.h**) defines the function API as described in Section 3.5.

To enable easy switching between DemoKey and BroadKey, the C code header files delivered with DemoKey have exactly the same names as for BroadKey. However, the binary object file will be named differently to avoid confusion.

## 3.2. States and State Transitions

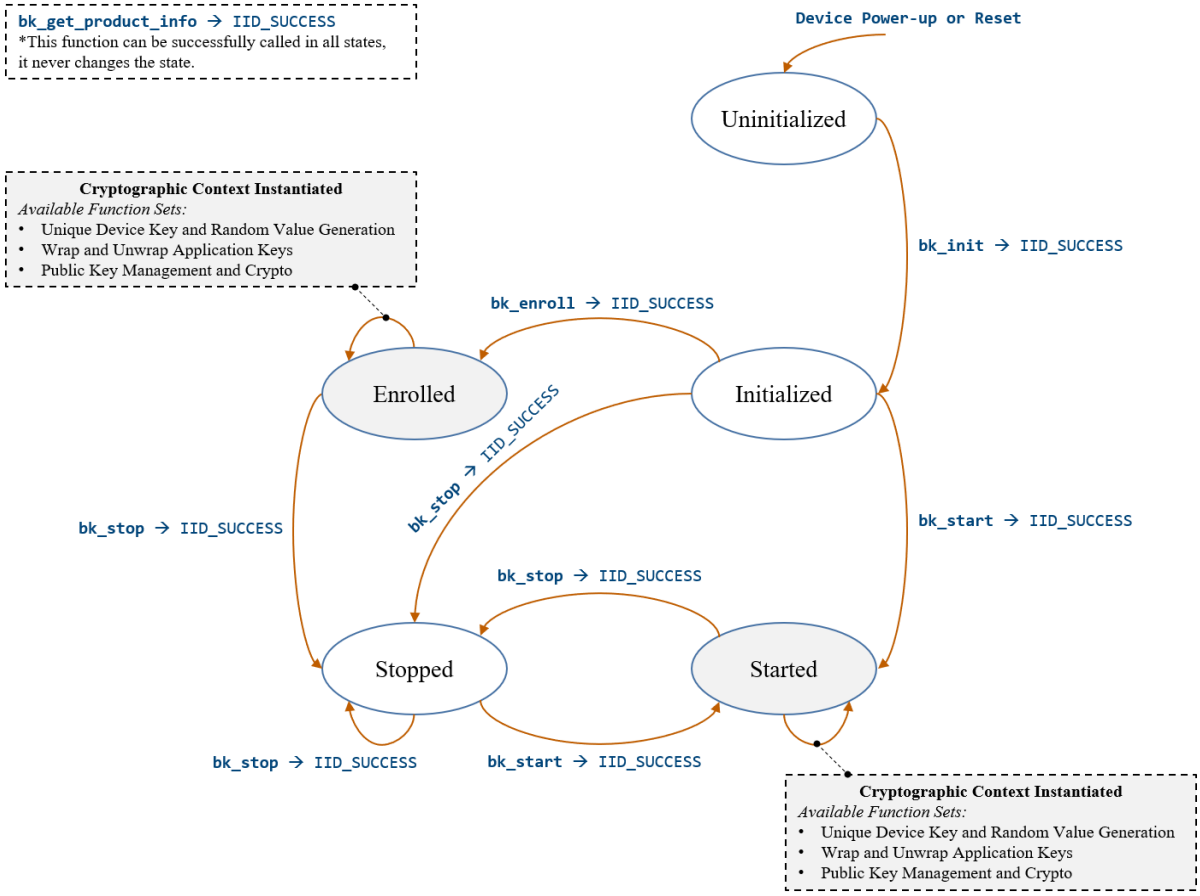


**Figure 3 BroadKey state diagram.**

Figure 3 BroadKey state diagram. depicts the state diagram of BroadKey. It shows the five different functional states which can be assumed by the BroadKey module, which are: Uninitialized, Initialized, Enrolled, Started and Stopped.

Figure 3 BroadKey state diagram.only shows the allowed functions/function sets[7] for each state, i.e. the function calls which can be successfully called from each state.[8] Calling a function which is not allowed by the current state (i.e. any function/function set which is not shown in Figure 3 BroadKey state diagram.) will return the return code IID_NOT_ALLOWED and BroadKey will stay in the state it is in. Calling an allowed function which for some other reason does *not* return IID_SUCCESS will also cause BroadKey to remain in its current state.

Finally, Figure 3 BroadKey state diagram.also shows all possible state transitions following a successfully invoked function call (unsuccessful calls never change the state). Summarized, there are four BroadKey API function calls which are mainly intended for changing the state of BroadKey, which are **bk_init**, **bk_enroll**, **bk_start** and **bk_stop**, and no other functions of BroadKey can change its state under any circumstance. The other API function calls are intended for performing the cryptographic functionality which is available to BroadKey's configuration, as described in Sections 2.3, 2.4 and 2.5. It is clear from Figure 3 BroadKey state diagram. that the cryptographic functionality is only available if a cryptographic context is instantiated, i.e. when BroadKey is in the Enrolled or in the Started state.

A device power-up or reset will result in BroadKey going to the Uninitialized state, regardless of which state it was in before the power-up or reset.

Since BroadKey is a stateful software module, its functional interface is not thread-safe nor reentrant.

## 3.3. BroadKey API Function Return Codes

All BroadKey API function calls will have as a return value a specified return code indicating whether the function ended successfully, or if not, what went wrong. Data results are always passed through pointers in the argument list of the function call. The following return codes are defined for BroadKey in **iidreturn_codes.h**. Depending on the function sets implemented by the current configuration of BroadKey (see Section 2.1), subsets of these return codes are implemented in the delivered BroadKey software module.



[7] The functions implemented by the different function sets, *Unique device key and random value generation*, *Wrap and unwrap application keys*, and *Public key management and crypto*, are respectively described in Sections 2.3, 2.4 and 2.5.

[8] For a function call to return IID_SUCCESS, more conditions may have to be met than just being in an allowed state, e.g. providing the correct parameters in the API call.

| Return Code | Description |
|---|---|
| **Generic return codes** | |
| IID_SUCCESS | Indicating the successful execution of the called function. |
| IID_NOT_ALLOWED | Indicating that the given function call is not allowed in the current state. See the state transition description in Section 3.2 for resolving this. |
| IID_INVALID_PARAMETERS | Indicating that at least one of the parameters passed as argument with this function call has an invalid form and/or content. This also occurs when one of the required output buffers contains a NULL pointer, or when one of the provided length parameters is not long enough. See the API function definitions in Section 3.5 for resolving this. |
| **Return codes specific to the *Base* function set** | |
| IID_ERROR_STARTUP_DATA | Indicating that the appointed SRAM array does not contain qualitative start-up data which can be used as an SRAM PUF by BroadKey. See the integration guidelines in Section 4 for resolving this.<br><br>Note that this return value indicates a blocking error, since one is not able to get out of the Uninitialized state (recalling **bk_init** will just return the same error code). A device repower is anyway required to get out of this, in combination with a resolution of the cause of this problem. |
| IID_INVALID_AC | Indicating that the activation code provided to **bk_start** is not a valid activation code for this device, i.e. it was not generated by a successful **bk_enroll** call on the same device. See the information on activation codes and cryptographic contexts in Section 2.2.2 for resolving this. |
| **Return codes specific to the *Wrap and unwrap application keys* function set** | |

| Return Code | Description |
|---|---|
| IID_INVALID_KEY_CODE | Indicating that the key code provided to **bk_unwrap** is not a valid key code for this device, i.e. it was not generated by a successful **bk_wrap** call on the same device in the same cryptographic context. |
| **Return codes specific to the *Public key management and crypto* function set** | |
| IID_ECC_NOT_ALLOWED | Indicating that the provided private and/or public key code inputs do not have the right purpose flags for being used by the called function. See Section 3.4.3.3 for the description of key purposes, and see Sections 3.5.4.1 and 3.5.4.3 for information on how to set key purposes. |
| IID_INVALID_PRIVATE_KEY | Indicating that an externally provided private key input value (to **bk_create_private_key** or **bk_derive_public_key**) is an invalid private key for the specified elliptic curve. |
| IID_INVALID_PUBLIC_KEY | Indicating that the provided public key input value (to **bk_import_public_key**) is an invalid public key for the specified elliptic curve. |
| IID_INVALID_PRIVATE_KEY_CODE | Indicating that the provided private key code input is not a valid private key code for the called function on this device. |
| IID_INVALID_PUBLIC_KEY_CODE | Indicating that the provided public key code input is not a valid public key code for the called function on this device. |
| IID_CURVE_MISMATCH | Indicating that the simultaneously provided private and public key code to a function (to **bk_generate_cryptogram**, **bk_process_cryptogram** or **bk_ecdh_shared_secret**) do not match the same elliptic curve. |
| IID_INVALID_SIGNATURE | Indicating that the signature input provided to **bk_ecdsa_verify** is not a valid signature for the provided message under the provided public key. |

| Return Code | Description |
|---|---|
| IID_INVALID_COUNTER | Indicating that the counter input provided to **bk_generate_cryptogram** results in a counter overflow, or that the counter input provided to **bk_process_cryptogram** is not smaller than or equal to the counter value contained in the received cryptogram. The latter indicates that there is an attempt to process a cryptogram that is the same or older than a previously processed one. |
| IID_INVALID_CRYPTOGRAM | Indicating that the provided cryptogram input to **bk_process_cryptogram** or **bk_get_public_key_from_cryptogram** is not a valid cryptogram, e.g. it has the wrong type, format, or its integrity was compromised. |
| IID_INVALID_SENDER | Indicating that the provided public key code to **bk_process_cryptogram** does not match the public key of the cryptogram's sender; i.e. the cryptogram did not originate from the expected source. |

DemoKey will also return IID_INVALID_PRIVATE_KEY if the provided private key does not match the additional format restrictions put forward in Section 2.5.1.1.

## 3.4. BroadKey API Defines and Type Definitions

### 3.4.1. Defines

The following constant values are defined in a hard-coded manner for BroadKey. They determine the setup and configuration of the delivered BroadKey module, and aid the developer in specifying lengths for various functional parameters. Depending on the configuration of BroadKey (see Section 2.1), subsets of these defines are implemented in the delivered BroadKey software module.

| BroadKey Define | Value | Description |
|---|---|---|
| **Generic defines** | | |

| BroadKey Define | Value | Description |
|---|---|---|
| BK_SECURITY_SIZE_BITS<br><br>*(This value depends on the security strength of the BroadKey configuration.)* | 128 or 256 | Defines the root security strength of the delivered BroadKey module, in number of bits. The security strength indicates the difficulty of a brute-force attack on BroadKey's internal secrets. The scale of this security strength is equivalent to the key length of a symmetric block cipher (e.g. AES).[9] |
| BK_CONFIGURATION_SAFE_ENABLED | / | This macro is defined if, and only if, the configuration of the delivered BroadKey module is BK-Safe (see also Section 2.1) |
| BK_CONFIGURATION_PLUS_ENABLED | / | This macro is defined if, and only if, the configuration of the delivered BroadKey module is BK-Plus (see also Section 2.1) |
| BK_CONFIGURATION_PRO_ENABLED | / | This macro is defined if, and only if, the configuration of the delivered BroadKey module is BK-Pro (see also Section 2.1) |
| **Defines specific to the *Base* function set** | | |
| BK_SRAM_PUF_SIZE_BYTES<br><br>*(This value depends on the security strength of the BroadKey configuration.)* | 512 or 1024 | Defines the size, in bytes, of the SRAM range containing the start-up data used in **bk_init**. |
| BK_AC_SIZE_BYTES<br><br>*(This value depends on the security strength of the BroadKey configuration.)* | 480 or 788 | Defines the size, in bytes, of the activation code as produced by **bk_enroll** and as input to **bk_start**. |



[9] This security strength reflects the strength of BroadKey's internal secrets, and hence determines the highest level of cryptographic security any of BroadKey's cryptographic functions can offer. The effective security strength of individual function calls can be lower, depending on the called function and the provided parameters. In particular for elliptic-curve cryptography, security strength cannot be higher than ½ of the private key length over the used curve (e.g. the NIST P-256 curve has 256-bit private keys and hence provides at most 128-bit of security strength).

| BroadKey Define | Value | Description |
|---|---|---|
| **Defines specific to the *Wrap and unwrap application keys* function set** | | |
| BK_KEY_CODE_ HEADER_SIZE_BYTES | 44 | Defines the size, in bytes, of the header of a key code as produced by **bk_wrap** and as input to **bk_unwrap**. The key code header contains the additional data of a key code, incremental to the actual wrapped key. The total size of a key code will always be the size of the key, in bytes, incremented with the header size, in bytes. |
| **Defines specific to the *Public key management and crypto* function set** | | |
| BK_ECC_CRYPTOGRAM_ HEADER_SIZE_BYTES | 80 | Defines the size, in bytes, of the header of a cryptogram as produced by **bk_generate_cryptogram** and as input to **bk_process_cryptogram**. The cryptogram header contains the additional data of a cryptogram, incremental to the contained plaintext and public key(s). |
| BK_ECC_MAX_CURVE_ SIZE_BYTES | 32 | Defines the maximal size, in bytes, of the representation of a field element over which an elliptic curve is defined. The maximum is taken over all elliptic curves supported by BroadKey. |
| BK_ECC_KEY_CODE_ HEADER_SIZE_BYTES | 48 | Defines the size, in bytes, of the header of an elliptic curve private or public key code as produced by **bk_create_private_key**, **bk_compute_public_from_private_key** and **bk_import_public_key**, and as input to various BroadKey elliptic curve cryptography functions. The elliptic curve key code header contains the additional data of an elliptic curve key code, incremental to the actual wrapped elliptic curve private or public key. |

| BroadKey Define | Value | Description |
|---|---|---|
| BK_ECC_PRIVATE_ KEY_CODE_SIZE_BYTES | ((BK_ECC_KEY_CODE_HEADER_SIZE_BYTES) + (BK_ECC_MAX_CURVE_SIZE)) | Defines the size, in bytes, of a private key code as produced by **bk_create_private_key**. This size is independent of the elliptic curve over which the contained private key is defined. |
| BK_ECC_PUBLIC_ KEY_CODE_SIZE_BYTES | ((BK_ECC_KEY_CODE_HEADER_SIZE_BYTES) + (2 * (BK_ECC_MAX_CURVE_SIZE))) | Defines the size, in bytes, of a public key code as produced by **bk_compute_public_from_private_key** and **bk_import_public_key**. This size is independent of the elliptic curve over which the contained public key is defined. |

BK_SECURITY_SIZE_BITS is defined for DemoKey, but it does not reflect the effective security strength of DemoKey (see also Section 2.1).

### 3.4.2. Type Definitions of the *Unique Device Key and Random Value Generation* Function Set

#### 3.4.2.1. bk_sym_key_type_t

The bk_sym_key_type_t type definition defines the key types which can be generated by the **bk_get_key** function (see Section 3.5.2.2). This key type also implies the length of the referred key, as indicated in the table below.

| bk_sym_key_type_t | Key Length (bytes) | Description |
|---|---|---|
| BK_SYM_KEY_TYPE_128 | 16 | A fully random 128-bit key, e.g. to be used for symmetric AES-128-based encryption.<br><br>The security strength of a key of this type is 128-bit. |
| BK_SYM_KEY_TYPE_192<br><br>*(This type is only available in BroadKey configurations with security strength 256.)* | 24 | A fully random 192-bit key, e.g. to be used for symmetric AES-192-based encryption.<br><br>The security strength of a key of this type is 192-bit. |

| | | |
|---|---|---|
| BK_SYM_KEY_TYPE_256<br><br>*(This type is only available in BroadKey configurations with security strength 256.)* | 32 | A fully random 256-bit key, e.g. to be used for symmetric AES-256-based encryption.<br><br>The security strength of a key of this type is 256-bit. |

The security strength of the respective symmetric key types generated by DemoKey does not match with equivalent keys generated by BroadKey. The security strength of any key generated by DemoKey is upper bounded by the overall effective security strength of DemoKey, which is very low (see also Section 2.1).

### 3.4.3. Type Definitions of the *Public Key Management and Crypto* Function Set

#### 3.4.3.1. bk_ecc_curve_t

The bk_ecc_curve_t type definition defines the named elliptic curves which can be recognized by the **bk_create_private_key** function (see Section 3.5.4.1) and the **bk_import_public_key** function (see Section 3.5.4.4). This curve type also implies the length of private keys, public keys, shared secrets and signatures computed over these curves, as indicated in the table below.

| bk_ecc_curve_t | Description |
|---|---|
| BK_ECC_CURVE_NIST_P192 | The elliptic-curve cryptosystem specified by the NIST P-192/secp192r1 domain parameters.[10]<br><br>Functional operations over this elliptic curve expect parameters of the following sizes, in bytes:<br>• private keys: 24<br>• public keys: 49<br>• ECDSA signatures: 48<br>• ECDH shared secrets: 24<br><br>The maximal security strength for operations over this curve is 96-bit. |

[10] The domain parameters for the NIST prime-field curves are specified in NIST FIPS Pub 186-4 “Digital Signature Standard (DSS)” (http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf), appendix D. These are the same prime-field curves as recommended in “SEC 2: Recommended Elliptic Curve Domain Parameters”, v2.0, 2010 (http://www.secg.org/sec2-v2.pdf).

| bk_ecc_curve_t | Description |
|---|---|
| BK_ECC_CURVE_NIST_P224 | The elliptic-curve cryptosystem specified by the NIST P-224/secp224r1 domain parameters.[10]<br><br>Functional operations over this elliptic curve expect parameters of the following sizes, in bytes:<br>• private keys: 28<br>• public keys: 57<br>• ECDSA signatures: 56<br>• ECDH shared secrets: 28<br><br>The maximal security strength for operations over this curve is 112-bit. |
| BK_ECC_CURVE_NIST_P256 | The elliptic-curve cryptosystem specified by the NIST P-256/secp256r1 domain parameters.[10]<br><br>Functional operations over this elliptic curve expect parameters of the following sizes, in bytes:<br>• private keys: 32<br>• public keys: 65<br>• ECDSA signatures: 64<br>• ECDH shared secrets: 32<br><br>The maximal security strength for operations over this curve is 128-bit. |

The security strength of the respective elliptic curve keys and operations over the different curves for DemoKey does not match with equivalent elliptic curve keys and operations for BroadKey. The security strength of any elliptic curve key or operation for DemoKey is upper bounded by the overall effective security strength of DemoKey, which is very low (see also Section 2.1).

#### 3.4.3.2. bk_ecc_key_source_t

The bk_ecc_key_source_t type definition defines the allowed sources from which an elliptic curve private key (code) can be generated using the **bk_get_private_key** (see Section 3.5.2.3) and **bk_create_private_key** functions (see Section 3.5.4.1), as indicated in the table below. This source type also partly determines other input parameters to this function.

| bk_ecc_key_source_t | Description |
|---|---|
| BK_ECC_KEY_SOURCE_PUF_DERIVED | The private key is derived in a direct line from the SRAM PUF's device-unique start-up data. Private keys derived in this way can always be exactly rederived by calling the generating function with the same arguments in the same cryptographic context, and on the same device. |
| BK_ECC_KEY_SOURCE_RANDOM | The private key is randomly generated using BroadKey's internal cryptographically secure random number generator which is seeded by entropy coming from the device noise. Private keys generated in this way cannot be rederived by the generating function, so if they are needed at a later time, the corresponding key code needs to be stored. |
| BK_ECC_KEY_SOURCE_USER_PROVIDED | The private key is an external value which is provided by the calling application to the **bk_create_private_key** function; **bk_get_private_key** does not accept this key source |

See the DemoKey-specific limitations on elliptic curve keys from the respective sources as detailed in Sections 2.3.1.2 and 2.5.1.1).

#### 3.4.3.3. bk_ecc_key_purpose_t

The bk_ecc_key_purpose_t type definition defines the allowed purposes which can be assigned to an elliptic curve private or public key, it is stored alongside the key in the corresponding key code. The purpose flag of a public or private key code determines whether other BroadKey elliptic curve crypto functions accept or reject the key contained in this key code, as indicated in the table below.

| bk_ecc_key_purpose_t | Description | Accepted by functions |
|---|---|---|
| BK_ECC_KEY_PURPOSE_ECDH | Key (code)s marked with this purpose flag can be used for elliptic curve key agreement and en/decryption functions. | **bk_ecdh_shared_secret**<br>**bk_generate_cryptogram**<br>**bk_process_cryptogram** |
| BK_ECC_KEY_PURPOSE_ECDSA | Key (code)s marked with this purpose flag can be used for elliptic curve signature generation and verification functions. | **bk_ecdsa_sign**<br>**bk_ecdsa_verify** |

| | | |
|---|---|---|
| BK_ECC_KEY_ PURPOSE_ECDH_ AND_ECDSA | This is the combination of the ECDH and ECDSA flags. Key (code)s marked with this purpose flag can be used for everything allowed by the ECDH and the ECDSA flags. | Union of the accepted functions for the BK_ECC_KEY_PURPOSE _ECDH and BK_ECC_KEY_PURPOSE _ECDSA purpose flags |

#### 3.4.3.4. bk_ecc_cryptogram_type_t

The bk_ecc_cryptogram_type_t type definition defines the cryptogram type to be used in the **bk_generate_cryptogram** function (see Section 3.5.4.9), and correspondingly the cryptogram type which was received by the **bk_process_cryptogram** function (see Section 3.5.4.10). The cryptogram type determines the exact security properties of the cryptogram, but also has an impact on performance, as indicated in the table below.

| bk_ecc_cryptogram_type_t | Description | Cryptogram-Plaintext size relation (in bytes) |
|---|---|---|
| BK_ECC_ CRYPTOGRAM_TYPE_ ECDH_STATIC | Cryptogram type using static elliptic curve key pairs on both sending and receiving side.<br><br>All the basic security properties (confidentiality, integrity, source authentication, non-replayable), but no forward secrecy and no non-repudiation<br><br>This cryptogram type achieves the highest performance, both in sending and receiving. | sizeof(cryptogram) = sizeof(plaintext) + BK_ECC_CRYPTOGRAM_ HEADER_SIZE_BYTES + sizeof(public_key) – 1<br><br>(See Section 3.4.3.1 for size of public keys for different supported curves.) |

| | | |
|---|---|---|
| BK_ECC_ CRYPTOGRAM_TYPE_ ECDH_EPHEMERAL | Cryptogram type using an ephemeral key pair on the sending side, and a static key pair on the receiving side.<br><br>All the basic security properties as above, and in addition forward secrecy w.r.t. loss of the sender's private key. Still no non-repudiation.<br><br>The performance of this cryptogram type is lower than BK_ECC_CRYPTOGRAM _TYPE_ECDH_STATIC for both sending and receiving. | sizeof(cryptogram) = sizeof(plaintext) + BK_ECC_CRYPTOGRAM_ HEADER_SIZE_BYTES + 2*sizeof(public_key) – 2<br><br>(See Section 3.4.3.1 for size of public keys for different supported curves.) |

#### 3.4.3.5. bk_ecc_private_key_code_t and bk_ecc_public_key_code_t

The bk_ecc_private_key_code_t type definition defines the representation of private key codes, used by the elliptic curve cryptographic functions of BroadKey. It is defined as a byte array of size BK_ECC_PRIVATE_KEY_CODE_SIZE_BYTES.

The bk_ecc_public_key_code_t type definition defines the representation of public key codes, used by the elliptic curve cryptographic functions of BroadKey. It is defined as a byte array of size BK_ECC_PUBLIC_KEY_CODE_SIZE_BYTES.

## 3.5. BroadKey API Function Definitions

This section describes the BroadKey module functions, organized in subsections according to their respective function sets. These functional interfaces are declared in **iidbroadkey.h**. Also see Section 2.6.1 for profiling information of these functions on a reference platform.

***Note:*** *in the API function description, the in and/or out indication for the arguments reflects the direction of the actual data. When an argument is a pointer, the pointer, and the allocated memory it points to, must be defined by the calling software.*

***Note:*** *the functions described in this API are in general neither thread-safe nor reentrant.*

The functional programming interface of DemoKey's functions is completely equivalent with BroadKey. However, please note that there are some important differences in functional execution, mainly:

- The security strength of DemoKey's operations is significantly lower than that of BroadKey.

- DemoKey has some additional restrictions on the format or length of certain functional parameters.
- The profiling results (speed, memory, etc.) of DemoKey's operations do not match those of BroadKey.

These differences are discussed in sufficient detail in Section 2. These differences are not repeated in the API descriptions below, since technically they do not affect the interface itself. The reader is strongly recommended to read Section 2 as well, in addition to the API descriptions below.

### 3.5.1. Functions of the *Base* Function Set

#### 3.5.1.1. bk_get_product_info

This function can be used to get the exact name, version and patch number of the software module.

A call to **bk_get_product_info** is always allowed in all states.

**bk_get_product_info** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_get_product_info(
    uint8_t * const product_id,
    uint8_t * const major_version,
    uint8_t * const minor_version,
    uint8_t * const patch,
    uint8_t * const build_number);
```

| Parameters | | |
|---|---|---|
| product_name | out | Pointer to a byte buffer which will hold a product identifier.<br><br>***Note:*** *the size of the buffer must be 1 byte.* |
| major_version | out | Pointer to a byte buffer which will hold the major software version.<br><br>***Note:*** *the size of the buffer must be 1 byte.* |
| minor_version | out | Pointer to a byte buffer which will hold the minor software version.<br><br>***Note:*** *the size of the buffer must be 1 byte.* |
| patch | out | Pointer to a byte buffer which will hold the software patch number.<br><br>***Note:*** *the size of the buffer must be 1 byte.* |

| Parameters | | |
|---|---|---|
| build_number | out | Pointer to a byte buffer which will hold the build number |
| | | ***Note:*** *the size of the buffer must be 1 byte.* |

### 3.5.1.2. bk_init

This function is used to initialize the BroadKey software module before use, after each device power-up or reset. It points the BroadKey module to the system's SRAM range which is reserved as SRAM PUF (also see Section 2.2.2.1).

A call to **bk_init** is only allowed when BroadKey is in the Uninitialized state; a call from any other state will return IID_NOT_ALLOWED. When **bk_init** returns IID_SUCCESS, BroadKey moves from the Uninitialized to the Initialized state.

**bk_init** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_ERROR_STARTUP_DATA
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_init(
          uint8_t  * const sram_puf,
    const uint16_t         sram_puf_size);
```

| **Parameters** | | |
|---|---|---|
| sram_puf | in/out | Pointer to physical SRAM PUF used by the module. The physical SRAM pointed to is both read (the start-up data it contains is used as a PUF), and written (amongst other things to condition it against silicon aging).<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| sram_puf_size | in | The size in bytes of available SRAM PUF that can be used by the software.<br><br>***Note:*** *the size of the SRAM must be (at least) BK_SRAM_PUF_SIZE_BYTES bytes.* |

### 3.5.1.3. bk_enroll

This function is used to instantiate a cryptographic context of BroadKey for the first time. It returns an activation code which, at a later time, can be used to re-instantiate the same context using **bk_start**. Once a cryptographic context is instantiated, BroadKey's cryptographic functionality becomes available (also see Section 2.2.2.2).

A call to **bk_enroll** is only allowed when BroadKey is in the Initialized state; a call from any other state will return IID_NOT_ALLOWED. When **bk_enroll** returns IID_SUCCESS, BroadKey moves from the Initialized to the Enrolled state.

**bk_enroll** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_enroll(
   uint8_t * const activation_code);
```

| Parameters | | |
|---|---|---|
| activation_code | out | Pointer to a buffer that will hold the generated activation code.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.*<br><br>***Note:*** *the size of the activation_code output buffer must be at least* BK_AC_SIZE_BYTES *bytes.* |

### 3.5.1.4. bk_start

This function is used to re-instantiate a cryptographic context of BroadKey based on a provided activation code which was earlier generated by **bk_enroll**. It is the responsibility of the calling software to reliably store an activation code after **bk_enroll**, and retrieve it before **bk_start.** Once a cryptographic context is instantiated, BroadKey's cryptographic functionality becomes available (also see Section 2.2.2.3).

A call to **bk_start** is allowed when BroadKey is in the Initialized state or in the Stopped state; a call from any other state will return IID_NOT_ALLOWED. When **bk_start** returns IID_SUCCESS, BroadKey moves to the Started state.

**bk_start** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_AC

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_start(
   const uint8_t * const activation_code);
```

| Parameters | | |
|---|---|---|
| activation_code | in | Pointer to a buffer that holds the retrieved activation code.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.*<br><br>***Note:*** *the size of the buffer must be at least* BK_AC_SIZE_BYTES *bytes.* |

### 3.5.1.5. bk_stop

This function will uninstantiate a cryptographic context of BroadKey which was earlier instantiated by **bk_enroll** or **bk_start**. Once a cryptographic context is uninstantiated, BroadKey's cryptographic functionality becomes unavailable (also see Section 2.2.2.4). Moreover, **bk_stop** also ensures that all internal secrets related to the cryptographic context are effectively deleted (zeroized), which can be used as an additional security measure against attacks.

A call to **bk_stop** is always allowed, except when BroadKey is in the Uninitialized state, in which case it will return IID_NOT_ALLOWED. When **bk_stop** returns IID_SUCCESS, BroadKey moves to the Stopped state. Note that:

- **bk_stop** will always return IID_SUCCESS (when not called from the Uninitialized state), hence the calling software can be assured that a call to **bk_stop** will always uninstantiate the cryptographic context.
- it is allowed to call **bk_stop** even from the Initialized and Stopped states, when there are no cryptographic contexts initialized. Calling **bk_stop** when already in the Stopped state has no effect. Calling **bk_stop** from the Initialized state has as effect that BroadKey moves to the Stopped state; as a result, **bk_enroll** becomes unavailable until the device is repowered or reset.

**bk_stop** can exit with one of the following return codes:

- IID_SUCCESS
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_stop(
   void);
```

| **Parameters** |
|---|
| (none) |

### 3.5.2. Functions of the *Unique Device Key and Random Value Generation* Function Set

#### 3.5.2.1. bk_generate_random

This function will generate a sequence of random bytes using a cryptographically secure random number generator which is seeded with unpredictable noise entropy from the device.

A call to **bk_generate_random** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_generate_random**, successful or not, will *not* change the state of BroadKey.

**bk_generate_random** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_generate_random(
   const uint16_t         number_of_bytes,
         uint8_t  * const data_buffer);
```

| Parameters | | |
|---|---|---|
| number_of_bytes | in | Positive integer in the range [1:65535] which specifies the number of random bytes that will be returned. The size, in bytes, of the allocated output buffer pointed to by data_buffer needs to be at least equal to this value. |
| data_buffer | out | Pointer to a byte array buffer which will hold the requested random bytes.<br><br>***Note:*** *the size of the data_buffer output buffer must be at least number_of_bytes bytes.* |

### 3.5.2.2. bk_get_key

This function will (re)generate a device-unique symmetric key for the cryptographic context which was earlier instantiated by **bk_enroll** or re-instantiated by **bk_start** (also see Sections 2.2.2.2 and 2.2.2.3). The length of the generated key depends on the specified key type. For each key type, **bk_get_key** can (re)generate up to 256 independent device key values, controlled by the key index input parameter. A call to **bk_get_key** with the same input parameter values (key_type and index) on the same device instantiated with the same cryptographic context, will always return the same key value.

A call to **bk_get_key** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_get_key**, successful or not, will *not* change the state of BroadKey.

**bk_get_key** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_get_key(
   const bk_sym_key_type_t         key_type,
   const uint8_t                   index,
         uint8_t             * const key);
```

| Parameters | | |
|---|---|---|
| key_type | in | The type of the device key that will be generated. This must be a value of the enumeration type bk_sym_key_type_t which is declared in iidbroadkey.h. The allowed key types, and their meaning, are explained in Section 3.4.2.1. |
| index | in | An integer value in the range [0:255] indicating the index of the device key that will be generated for the specified key type. For each index value, a key is generated which is completely independent from keys generated by other key index values. |
| key | out | Pointer to a byte array buffer that will hold the generated device key.<br><br>***Note:*** *the size of the key buffer must be at least long enough to hold the generated key type, as specified by the key_type input parameter (see Section 3.4.2.1).* |

### 3.5.2.3. bk_get_private_key

This function will generate a random or a device-unique elliptic curve private key for the cryptographic context which was earlier instantiated by **bk_enroll** or re-instantiated by **bk_start** (also see Sections 2.2.2.2 and 2.2.2.3). The length of the generated private key depends on the specified elliptic curve (also see Section 3.4.3.1). For each curve option, **bk_get_private_key** can (re)generate multiple device-unique key values by altering the usage context input parameter. The usage context input can be used for key diversification in the application, and/or to include application-provided key information or entropy into the key generation process.

**bk_get_private_key** can generate elliptic curve private keys from two possible sources:

- device-unique private keys derived from the device's secret fingerprint. In this case, providing the same input parameter values (curve and usage context) on the same device instantiated with the same cryptographic context, will always return the same private key value.
- randomly generated private keys derived from the device's power-up noise. In this case, always a fresh and unpredictably random private key value is returned.

A call to **bk_get_private_key** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_get_private_key**, successful or not, will *not* change the state of BroadKey.

**bk_get_private_key** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_get_private_key(
   const bk_ecc_curve_t              curve,
   const uint8_t             * const usage_context,
   const uint32_t                    usage_context_length,
   const bk_ecc_key_source_t         key_source,
         uint8_t             * const private_key);
```

| Parameters | | |
|---|---|---|
| curve | in | Specifies the named elliptic curve on which the considered private key is defined. It must be a valid curve type of the bk_ecc_curve_t enumeration which is declared in iidbroadkey.h.The allowed curves, and their meaning, are explained in Section 3.4.3.1. |

| Parameters | | |
|---|---|---|
| usage_context | in | Pointer to a byte array buffer which holds an (optional) usage context. When used, the entropy of this buffer is included in the private key derivation for private keys derived from the device fingerprint (key_source=BK_ECC_KEY_SOURCE_PUF_DERIVED) or from the device's random number generator (key_source=BK_ECC_KEY_SOURCE_RANDOM).<br><br>***Note:*** *the size of the buffer must be at least usage_context_length bytes.*<br><br>***Note:*** *providing a usage context is optional, if the specified usage_context_length is 0, no usage context is taken into account.* |
| usage_context_length | in | Value which specifies the length in bytes of the usage_context buffer. If this length is set to 0, no usage context is taken into account. |
| key_source | in | Specifies the source of the elliptic curve private key. It must be a valid source of the bk_ecc_key_source_t enumeration which is declared in iidbroadkey.h. The allowed private key sources, and their meaning, are explained in Section 3.4.3.2. The allowed key sources for **bk_get_private_key** are:<br>– BK_ECC_KEY_SOURCE_PUF_DERIVED: the private key is derived from the device fingerprint and (optionally) the provided usage context.<br>– BK_ECC_KEY_SOURCE_RANDOM: the private key is uniformly randomly generated from BroadKey's internal random number generator and (optionally) the provided usage context. |
| private_key | out | Pointer to a byte array buffer that will hold the generated elliptic curve private key. The private key value is provided in binary format, in network byte order representation.<br><br>***Note:*** *the size of the private key buffer must be at least long enough to hold a private key for the provided curve type, as specified by the curve input parameter (see Section 3.4.3.1).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

## 3.5.3. Functions of the *Wrap and Unwrap Application Keys* Function Set

### 3.5.3.1. bk_wrap

This function will securely wrap (authenticated encrypt) an externally provided application key into a key code. The length of the provided key must be a multiple of 4 bytes, with a minimum of 4 bytes and maximum of 1024 bytes. The length of the generated key code will be the length of the provided key incremented with the constant size of the key code header (BK_KEY_CODE_HEADER_SIZE_BYTES). In addition to a variable-length key, the application can provide an index value which gets wrapped alongside the key. The application can assign a custom meaning to this index which relates to the context of the key. When the key code is unwrapped again with **bk_unwrap**, the index will also be returned.

A call to **bk_wrap** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_wrap**, successful or not, will *not* change the state of BroadKey.

**bk_wrap** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_wrap(
   const uint8_t          index,
   const uint8_t  * const key,
   const uint16_t         key_length,
         uint8_t  * const key_code);
```

| Parameters | | |
|---|---|---|
| index | in | An integer value in the range [0:255] indicating the index of the key-to-be-wrapped. For **bk_wrap/unwrap**, the index is an application-defined value that gets wrapped (in **bk_wrap**) and unwrapped (in **bk_unwrap**) alongside the actual key value. The application using BroadKey can use it, e.g. to specify context-information associated to the key. |
| key | in | Pointer to a byte array buffer that holds the plain key-to-be-wrapped.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.*<br><br>***Note:*** *the size of the buffer must be equal to key_length bytes.* |
| key_length | in | The length, in bytes, of the key-to-be-wrapped. This must be an integer value in the range [4:4:1024], i.e. the smallest allowed value is 4, the largest is 1024, and only values that are a multiple of 4 are allowed. |

| Parameters | | |
|---|---|---|
| key_code | out | Pointer to a byte array buffer that will hold the generated key code.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.*<br><br>***Note:*** *the size of the key_code buffer must be at least (key_length + BK_KEY_CODE_HEADER_SIZE_BYTES) bytes.* |

### 3.5.3.2. bk_unwrap

This function will successfully unwrap (decrypt and authenticate) a provided key code, given that it is called on the same device and in the same cryptographic context that was used to produce the key code with **bk_wrap**. In addition to the originally wrapped key, **bk_unwrap** will also return the index value that got wrapped alongside the key. The application can parse this index value to determine the context of the key.

**bk_unwrap** can determine the exact length of the key code automatically by parsing the key code header, so the application does not need to provide the key code length. However, the calling software does need to assure that the allocated key code buffer is large enough to hold the complete key code string.

A call to **bk_unwrap** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_unwrap**, successful or not, will *not* change the state of BroadKey.

**bk_unwrap** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_KEY_CODE

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_unwrap(
   const uint8_t  * const key_code,
         uint8_t  * const key,
         uint16_t * const key_length,
         uint8_t  * const index);
```

| Parameters | | |
|---|---|---|
| key_code | in | Pointer to a byte array buffer that holds the retrieved key code.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.*<br><br>***Note:*** *the size of the buffer must be sufficient to contain the full key code as was produced by a call to* ***bk_wrap****.* |
| key | out | Pointer to a byte array buffer that will hold the unwrapped key. |
| key_length | out | Pointer to a byte buffer which will contain the size in bytes of key. Its value will be in the [4, 1024] range and a multiple of 4. |

| Parameters | | |
|---|---|---|
| index | out | An integer value in the range [0:255] indicating the index of the unwrapped key. For **bk_wrap/unwrap**, the index is an application-defined value that gets wrapped (in **bk_wrap**) and unwrapped (in **bk_unwrap**) alongside the actual key value. The application using BroadKey can use it, e.g. to specify context-information associated to the key. |

## 3.5.4. Functions of the *Public Key Management and Crypto* Function Set

### 3.5.4.1. bk_create_private_key

This function transforms an elliptic curve private key into a protected private key code which is only usable within the same cryptographic context, and on the same unique device, it was created on.

This function can take private keys from three possible sources:

- device-unique private keys derived from the device's secret fingerprint
- randomly generated private keys derived from the device power-up noise
- user-provided private keys

Alongside the private key values, this function also stores the curve and key purpose flags in the private key code format. This makes the future use of a generated private key code self-contained, i.e. a consuming function knows on which curve the contained private key is defined, and for which purposes it is allowed to be used.

***Note:*** *the generation mechanisms for creating device-unique and random private keys are similar as for the* ***bk_get_private_key*** *function (see Section 3.5.2.3), but private keys generated by* ***bk_get_private_key*** *and* ***bk_create_private_key*** *are strongly cryptographically separated. This entails that calling* ***bk_get_private_key*** *and* ***bk_create_private_key*** *with equal parameters will always result in completely different private key values.*

***Note:*** *the protection mechanisms for transforming private keys into private key codes are similar as for the* ***bk_wrap*** *function (see Section 3.5.3.1), but private key codes cannot be unwrapped by* ***bk_unwrap****. The underlying internal keys used by* ***bk_create_private_key*** *for protecting private key codes are also different as for key codes generated by* ***bk_wrap****. Once packed into a private key code, the actual private key values can no longer be publicly retrieved by BroadKey.*

A call to **bk_create_private_key** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_create_private_key**, successful or not, will *not* change the state of BroadKey.

**bk_create_private_key** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PRIVATE_KEY

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_create_private_key(
   const bk_ecc_curve_t                    curve,
   const bk_ecc_key_purpose_t              purpose_flags,
   const uint8_t                   * const usage_context,
   const uint32_t                          usage_context_length,
   const bk_ecc_key_source_t               key_source,
   const uint8_t                   * const private_key,
         bk_ecc_private_key_code_t * const private_key_code);
```

| Parameters | | |
|---|---|---|
| curve | in | Specifies the named elliptic curve on which the considered private key is defined. It must be a valid curve type of the bk_ecc_curve_t enumeration which is declared in iidbroadkey.h.The allowed curves, and their meaning, are explained in Section 3.4.3.1. |
| purpose_flags | in | Flag which specifies the usage purpose of the private key. It must be a valid flag of the bk_ecc_key_purpose_t enumeration which is declared in iidbroadkey.h. The allowed key purposes, and their meaning, are explained in Section 3.4.3.3. |
| usage_context | in | Pointer to a byte array buffer which holds an (optional) usage context. When used, the entropy of this buffer is included in the private key derivation for private keys derived from the device fingerprint (key_source=BK_ECC_KEY_SOURCE_PUF_DERIVED) or from the device's random number generator (key_source=BK_ECC_KEY_SOURCE_RANDOM).<br><br>***Note:*** *the size of the buffer must be at least usage_context_length bytes.*<br><br>***Note:*** *providing a usage context is optional, if the specified usage_context_length is 0, no usage context is taken into account.* |
| usage_context_length | in | Value which specifies the length in bytes of the usage_context buffer. If this length is set to 0, no usage context is taken into account. |

| Parameters | | |
|---|---|---|
| key_source | in | Specifies the source of the elliptic curve private key. It must be a valid source of the bk_ecc_key_source_t enumeration which is declared in iidbroadkey.h. The allowed private key sources, and their meaning, are explained in Section 3.4.3.2. In summary:<br>– BK_ECC_KEY_SOURCE_PUF_DERIVED: the private key is derived from the device fingerprint and (optionally) the provided usage context.<br>– BK_ECC_KEY_SOURCE_RANDOM: the private key is uniformly randomly generated from BroadKey's internal random number generator and (optionally) the provided usage context.<br>– BK_ECC_KEY_SOURCE_USER_PROVIDED: the private key is provided externally. When this key source is selected, usage_context is not used and private_key is used directly with only a check that it is a well-formed private key for the specified curve.<br>The resulting private key will be wrapped by a device-unique PUF key; hence the resulting private key code can only be used within the same cryptographic context on the same device.<br><br>***Note:*** *for BK_ECC_KEY_SOURCE_PUF_DERIVED and BK_ECC_KEY_SOURCE_RANDOM the usage_context entropy, if present, is added to the key derivation process to provide a secure fallback in case of entropy shortage, or to allow for key diversification.* |
| private_key | in | Pointer to a byte array buffer which holds the private key used when key_source is BK_ECC_KEY_SOURCE_USER_PROVIDED. The expected input is binary in network byte order representation. Its size in bytes is determined by the specified curve, as detailed in Section 3.4.3.1.<br><br>***Note:*** *for key sources other than BK_ECC_KEY_SOURCE_USER_PROVIDED, this input is not used.*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

| Parameters | | |
|---|---|---|
| private_key_code | out | Pointer to a private key code type buffer which will hold the created elliptic curve private key. The private key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

### 3.5.4.2. bk_compute_public_from_private_key

This function computes the elliptic curve public key corresponding to a private key code created with **bk_create_private_key**, and outputs the public key in a corresponding public key code format. The curve and purpose flags of the public key (code) will be the same as the one of the provided private key (code).

***Note:*** *the protection mechanisms for storing public keys as public key codes are similar as for the* ***bk_wrap*** *function (see Section 3.5.3.1), but public key codes cannot be unwrapped by* ***bk_unwrap****. The underlying internal keys used by* ***bk_compute_public_from_private_key*** *for protecting public key codes are also different as for key codes generated by* ***bk_wrap****. If needed, the function* ***bk_export_public_key*** *(see Section 3.5.4.5) can be used to retrieve the public key value contained in a public key code.*

A call to **bk_compute_public_from_private_key** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_compute_public_from_private_key**, successful or not, will *not* change the state of BroadKey.

**bk_compute_public_from_private_key** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PRIVATE_KEY_CODE

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_compute_public_from_private_key(
   const bk_ecc_private_key_code_t * const private_key_code,
         bk_ecc_public_key_code_t  * const public_key_code);
```

| Parameters | | |
|---|---|---|
| private_key_code | in | Pointer to a private key code type buffer which holds the elliptic curve private key, and was created by **bk_create_private_key**. The private key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| public_key_code | out | Pointer to a public key code type buffer which will hold the elliptic curve public key computed from the provided private key (code) input. The public key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

### 3.5.4.3. bk_derive_public_key

This function computes the elliptic curve public key corresponding to a private key, e.g. created with **bk_get_private_key** (see Section 3.5.2.3) and outputs the public key.

A call to **bk_derive_public_key** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_derive_public_key**, successful or not, will *not* change the state of BroadKey.

**bk_derive_public_key** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PRIVATE_KEY

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_derive_public_key(
   const bool                  use_point_compression,
   const bk_ecc_curve_t        curve,
   const uint8_t        * const private_key,
         uint8_t        * const public_key);
```

| Parameters | | |
|---|---|---|
| use_point_compression | in | ***Note:*** *this flag is present for future use compatibility, but is not used for this product version of BroadKey.* ***For this product version, this value has to be set to False (no point compression)****, any other value will result in the return code IID_INVALID_PARAMETERS.* |
| curve | in | Specifies the named elliptic curve on which the considered private key is defined. It must be a valid curve type of the bk_ecc_curve_t enumeration which is declared in iidbroadkey.h.The allowed curves, and their meaning, are explained in Section 3.4.3.1. |
| private_key | in | Pointer to a byte array buffer which holds the elliptic curve private key. The expected input is binary in network byte order representation. Its size in bytes is determined by the specified curve, as detailed in Section 3.4.3.1.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

| Parameters | | |
|---|---|---|
| public_key | out | Pointer to a byte array buffer which will hold the elliptic curve public key computed from the provided private key input. The output is in X9.62 binary format. Its size in bytes is determined by the specified curve, as detailed in Section 3.4.3.1.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

### 3.5.4.4. bk_import_public_key

This function imports an elliptic curve public key from a provided X9.62 binary format (uncompressed) to a corresponding public key code format.[11] The curve and purpose flags of the public key (code) are also provided as inputs and stored in the public key code.

***Note:*** *the protection mechanisms for storing public keys as public key codes are similar as for the* ***bk_wrap*** *function (see Section 3.5.3), but public key codes cannot be unwrapped by* ***bk_unwrap****. The underlying internal keys used by* ***bk_import_public_key*** *for protecting public key codes are also different as for key codes generated by* ***bk_wrap****. If needed, the function* ***bk_export_public_key*** *(see Section 3.5.4.5) can be used to retrieve the public key value contained in a public key code.*

A call to **bk_import_public_key** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_import_public_key**, successful or not, will *not* change the state of BroadKey.

**bk_import_public_key** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PUBLIC_KEY

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_import_public_key(
   const bk_ecc_curve_t                    curve,
   const bk_ecc_key_purpose_t              purpose_flags,
   const uint8_t                   * const public_key,
         bk_ecc_public_key_code_t * const public_key_code);
```

| Parameters | | |
|---|---|---|
| curve | in | Specifies the named elliptic curve on which the imported public key is defined. It must be a valid curve type of the bk_ecc_curve_t enumeration which is declared in iidbroadkey.h. The allowed curves, and their meaning, are explained in Section 3.4.3.1. |
| purpose_flags | in | Flag which specifies the usage purpose of the public key. It must be a valid flag of the bk_ecc_key_purpose_t enumeration which is declared in iidbroadkey.h. The allowed key purposes, and their meaning, are explained in Section 3.4.3.3. |
| public_key | in | Pointer to a byte array buffer which holds the public key to be imported. The expected input is in X9.62 uncompressed binary format. Its size in bytes is determined by the specified curve as detailed in Section 3.4.3.1. |

| Parameters | | |
|---|---|---|
| public_key_code | out | Pointer to a public key code type buffer which will hold the imported elliptic curve public key. The public key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

### 3.5.4.5. bk_export_public_key

This function exports a public key from BroadKey's public key code format to an X9.62 (uncompressed) binary elliptic curve public key format.[11] The curve on which the public key is defined, as well as the purpose flags stored alongside the key in the public key code, are returned as well.

A call to **bk_export_public_key** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_export_public_key**, successful or not, will *not* change the state of BroadKey.

**bk_export_public_key** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PUBLIC_KEY_CODE

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_export_public_key(
   const bool                              use_point_compression,
   const bk_ecc_public_key_code_t * const public_key_code,
         uint8_t                  * const public_key,
         bk_ecc_curve_t           * const curve,
         bk_ecc_key_purpose_t     * const purpose_flags);
```

| Parameters | | |
|---|---|---|
| use_point_compression | in | ***Note:*** *this flag is present for future use compatibility, but is not used for this product version of BroadKey.* ***For this product version, this value has to be set to False (no point compression)****, any other value will result in the return code IID_INVALID_PARAMETERS.* |
| public_key_code | in | Pointer to a public key code type buffer which holds the elliptic curve public key to be exported. The public key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

[11] "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 2005. (http://webstore.ansi.org/ansidocstore)

| Parameters | | |
|---|---|---|
| public_key | out | Pointer to a byte array buffer which will hold the exported public key. The output is in X9.62 binary format. Its exact size in bytes is determined by the specified curve as detailed in Section 3.4.3.1. |
| curve | out | Specifies the named elliptic curve on which the exported public key is defined. It will be a curve type of the bk_ecc_curve_t enumeration which is declared in iidbroadkey.h. The possible curves, and their meaning, are explained in Section 3.4.3.1. |
| purpose_flags | out | Flag which indicates the stored usage purpose of the public key. It will be a valid flag of the bk_ecc_key_purpose_t enumeration which is declared in iidbroadkey.h. The possible key purposes, and their meaning, are explained in Section 3.4.3.3. |

### 3.5.4.6. bk_ecdsa_sign

This function signs a message or a hash of message using ECDSA with an elliptic curve private key in the internal private key code format. Signing can be done with either a random seed or a deterministically derived seed as indicated by the calling application.

A call to **bk_ecdsa_sign** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_ecdsa_sign**, successful or not, will *not* change the state of BroadKey.

**bk_ecdsa_sign** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PRIVATE_KEY_CODE
- IID_ECC_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_ecdsa_sign(
   const bk_ecc_private_key_code_t * const private_key_code,
   const bool                              deterministic_signature,
   const uint8_t                   * const message,
   const uint32_t                          message_length,
   const bool                              message_is_hash,
         uint8_t                   * const signature,
         uint16_t                  * const signature_length);
```

| Parameters | | |
|---|---|---|
| private_key_code | in | Pointer to a private key code type buffer which holds the elliptic curve private key to be used for signing, and as was created by **bk_create_private_key**. The private key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *a private key code used for signing shall have been created with purpose flags allowing its use for signing (see Section 3.4.3.3).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

| Parameters | | |
|---|---|---|
| deterministic_signature | in | Value which specifies if deterministic or non-deterministic signing will be used. If this value equals False, message will be signed using the standard ECDSA non-deterministic algorithm. Otherwise, the message will be signed using a deterministic algorithm.[12] |
| message | in | Pointer to a byte array buffer which holds the message or the message hash that will be signed. |
| message_length | in | Value which specifies the size in bytes of the message buffer.<br><br>If message_is_hash True (i.e. the provided message is actually a message hash), the size must be equal to the size in bytes of the used private key, as determined by the used curve, and as specified in Section 3.4.3.1.<br><br>Otherwise (i.e. the provided message is a raw message byte array), the size in bytes must be equal to the raw message length. In this case, message_length could also be zero in which case an empty message will be signed. |
| message_is_hash | in | Value which specifies if the provided message buffer contains an already hashed message, or a raw message byte array.<br><br>If this value equals False, **bk_ecdsa_sign** will treat the message buffer as a raw message, and will hash it first using SHA-256 and the trailing bytes will be truncated to equal the size of the used elliptic curve private key (see Section 3.4.3.1) before signing the resulting hash.<br><br>Otherwise, **bk_ecdsa_sign** will treat the message buffer as an already hashed message, and it will be signed directly. |
| signature | out | Pointer to a byte array buffer which will hold the computed ECDSA signature. Its exact size in bytes depends on the used curve, as contained in the private key code. Section 3.4.3.1 specifies the byte lengths for signatures over all supported curves. |

[12] Deterministic ECDSA signing is not a widely approved standard, but it can have certain security and/or usability benefits in particular situations. The algorithm used for deterministic ECDSA signing by BroadKey is compliant with the description in RFC-6979, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", T. Pornin, (https://tools.ietf.org/html/rfc6979).

| Parameters | | |
|---|---|---|
| signature_length | in/out | As input, this is the length in bytes of the allocated output buffer pointed to by signature. This value is used to check for buffer overflow.<br><br>***Note:*** *the size of the signature buffer must be at least long enough to hold a signature value for the used curve, as specified in Section 3.4.3.1.*<br><br>As output, this is the actual size in bytes of the returned signature value, as specified in Section 3.4.3.1. |

### 3.5.4.7. bk_ecdsa_verify

This function verifies the ECDSA signature of a message or a hash of message with an elliptic curve public key in the internal public key code format.[13]

***Note: bk_ecdsa_verify** has no explicit output parameters, only inputs. Its result is contained in its return code. If IID_SUCCESS is returned, the signature on the message is successfully verified. If IID_INVALID_SIGNATURE is returned, the signature on the message is invalid. For other return codes **bk_ecdsa_verify** failed to complete (see Section 3.3 for more information).*

A call to **bk_ecdsa_verify** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_ecdsa_verify**, successful or not, will *not* change the state of BroadKey.

**bk_ecdsa_verify** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PUBLIC_KEY_CODE
- IID_ECC_NOT_ALLOWED
- IID_INVALID_SIGNATURE

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_ecdsa_verify(
   const bk_ecc_public_key_code_t * const public_key_code,
   const uint8_t                  * const message,
   const uint32_t                         message_length,
   const bool                             message_is_hash,
   const uint8_t                  * const signature,
   const uint16_t                         signature_length);
```

[13] ECDSA signature verification is indifferent to whether deterministic or non-deterministic ECDSA signing was used.

**Parameters**

| | | |
|---|---|---|
| public_key_code | in | Pointer to a public key code type buffer which holds the elliptic curve public key to be used for signature verification. The public key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *a public key code used for signature verification shall have been computed or imported with purpose flags allowing its use for signing (see Section 3.4.3.3).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| message | in | Pointer to a byte array buffer which holds the message or the message hash on which a signature will be verified. |
| message_length | in | Value which specifies the size in bytes of the message buffer.<br><br>If message_is_hash is True (i.e. the provided message is actually a message hash), the size must be equal to the size in bytes of the used private key, as determined by the used curve, and as specified in Section 3.4.3.1.<br><br>Otherwise (i.e. the provided message is a raw message byte array), the size in bytes must be equal to the raw message length. In this case, message_length could also be zero in which case the signature on an empty message will be verified. |
| message_is_hash | in | Value which specifies if the provided message buffer contains an already hashed message, or a raw message byte array.<br><br>If this value equals False, **bk_ecdsa_verify** will treat the message buffer as a raw message, and will hash it first using SHA-256 and the trailing bytes will be truncated to equal the size of the used elliptic curve private key (see Section 3.4.3.1) before verifying the signature on the resulting hash.<br><br>Otherwise, **bk_ecdsa_verify** will treat the message buffer as an already hashed message, and it will be verified directly. |
| signature | in | Pointer to a byte array buffer which holds the to-be-verified ECDSA signature. |
| signature_length | in | This is the length in bytes of the provided signature.<br><br>***Note:*** *the size of the signature must be exactly the right value as specified for the used curve (see Section 3.4.3.1).* |

### 3.5.4.8. bk_ecdh_shared_secret

This function computes a shared secret value using the ECDH algorithm on the provided private and public key (codes). The returned shared secret comprises the X-coordinate of the mutual curve point computed with the elliptic curve Diffie-Hellman method.

A call to **bk_ecdh_shared_secret** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_ecdh_shared_secret**, successful or not, will *not* change the state of BroadKey.

**bk_ecdh_shared_secret** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_PRIVATE_KEY_CODE
- IID_INVALID_PUBLIC_KEY_CODE
- IID_ECC_NOT_ALLOWED
- IID_CURVE_MISMATCH

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_ecdh_shared_secret(
   const bk_ecc_private_key_code_t * const private_key_code,
   const bk_ecc_public_key_code_t  * const public_key_code,
         uint8_t                   * const shared_secret);
```

| Parameters | | |
|---|---|---|
| private_key_code | in | Pointer to a private key code type buffer which holds the elliptic curve private key to be used for computing the shared secret, and as was created by **bk_create_private_key**. The private key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *a private key code used for shared secret computation shall have been created with purpose flags allowing its use for ECDH (see Section 3.4.3.3).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

| Parameters | | |
|---|---|---|
| public_key_code | in | Pointer to a public key code type buffer which holds the elliptic curve public key to be used for computing the shared secret. The public key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *a public key code used for shared secret computation shall have been computed or imported with purpose flags allowing its use for ECDH (see Section 3.4.3.3).*<br><br>***Note:*** *a public key code used for shared secret computation shall contain a public key defined over the same curve as the simultaneously provided private key code (see Section 3.4.3.1).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| shared_secret | out | Pointer to a byte array buffer which will hold the computed shared secret. The shared secret will be equal to the X-coordinate of the commonly derived point on the elliptic curve. The exact size in bytes of the shared secret depends on the used curve, as contained in the private key code (see Section 3.4.3.1). |

### 3.5.4.9. bk_generate_cryptogram

This function packs a provided plaintext in a protected cryptogram format using an elliptic curve hybrid encryption scheme. The cryptogram format offers protection for confidentiality, integrity, sender authentication and replay. See Sections 2.5.3.2 and 3.4.3.4 for more background on the security properties depending on the used cryptogram type.

A cryptogram is the single message in a one-pass protocol from a sender to a receiver. The cryptogram generation is based simultaneously on the sender's private key and the receiver's public key. Both private and public key are provided as key codes which have to be defined over the same elliptic curve, and both keycodes must have their purpose flags set to allow the keys being used for encryption.

A call to **bk_generate_cryptogram** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_generate_cryptogram**, successful or not, will *not* change the state of BroadKey.

**bk_generate_cryptogram** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_COUNTER
- IID_INVALID_PRIVATE_KEY_CODE
- IID_INVALID_PUBLIC_KEY_CODE
- IID_ECC_NOT_ALLOWED
- IID_CURVE_MISMATCH

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_generate_cryptogram(
   const bk_ecc_public_key_code_t  * const receiver_public_key_code,
   const bk_ecc_private_key_code_t * const sender_private_key_code,
   const bk_ecc_cryptogram_type_t          cryptogram_type,
         uint8_t                   * const counter64,
   const uint8_t                   * const plaintext,
   const uint32_t                          plaintext_length,
         uint8_t                   * const cryptogram,
         uint32_t                  * const cryptogram_length);
```

| Parameters | | |
|---|---|---|
| receiver_public_key_code | in | Pointer to a public key code type buffer which holds the elliptic curve public key of the receiver to whom the generated cryptogram will be sent. The public key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *in order to have secure receiver authentication (i.e. assurance to the sender that only the intended receiver will be able to unpack the cryptogram), the public key contained in receiver_public_key_code shall have been verified in an independent manner (e.g. through certificate validation), or it shall come from an independent trusted source.*<br><br>***Note:*** *a public key code used for cryptogram generation shall have been computed or imported with purpose flags allowing its use for ECDH/encryption (see Section 3.4.3.3).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| sender_private_key_code | in | Pointer to a private key code type buffer which holds the elliptic curve private key of the sender whom will send the generated cryptogram. The private key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *a private key code used for cryptogram generation shall have been created with purpose flags allowing its use for ECDH/encryption (see Section 3.4.3.3).*<br><br>***Note:*** *a private key code used for cryptogram generation shall contain a private key defined over the same curve as the simultaneously provided public key code (see Section 3.4.3).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| cryptogram_type | in | Flag which specifies the cryptogram type to be generated. It must be a valid flag of the bk_ecc_cryptogram_type_t enumeration which is declared in iidbroadkey.h. The supported cryptogram types, and their meaning, are explained in Section 3.4.3.4. |

| Parameters | | |
|---|---|---|
| counter64 | in/out | Pointer to a byte array buffer which holds the current 64-bit monotonic counter used for cryptogram replay protection, and will hold the new counter value after successful function completion. Its size in bytes is 8.<br><br>***Note:*** *the counter value is represented in the byte array as a 64-bit unsigned integer in big-endian format (counter[0] contains the most-significant byte).*<br><br>***Note:*** *a separate counter byte array shall be used for each distinct sender-receiver key pair. The calling application needs to retrieve this array from persistent storage before each call to* ***bk_generate_cryptogram****, and store back its updated value in persistent storage after the function completes successfully (return code is IID_SUCCESS). Upon first use of a counter array for a sender-receiver pair, the counter array needs to be initialized to all-zero bytes, after the initial validation of the receiver's public key.* |
| plaintext | in | Pointer to a byte array buffer which holds the plaintext which will be encrypted in the cryptogram. |
| plaintext_length | in | Value which specifies the size in bytes of the plaintext buffer. Its value must be positive (>0) and a multiple of 4. |
| cryptogram | out | Pointer to a byte array buffer which will hold the generated cryptogram. Its exact size in bytes depends on the used curve, as contained in the private key code, on the specified cryptogram_type, and on the length of the provided plaintext, as specified in Section 3.4.3.4.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

| Parameters | | |
|---|---|---|
| cryptogram_length | in/out | As input, this is the length in bytes of the allocated output buffer pointed to by cryptogram. This value is used to check for buffer overflow.<br><br>***Note:*** *the size of the cryptogram buffer must be at least long enough to hold a cryptogram for the provided plaintext length, and the used curve and cryptogram type, as specified in Section 3.4.3.4.*<br><br>As output, this is the actual size in bytes of the returned cryptogram, as specified in Section 3.4.3.4.<br><br>***Note:*** *if the size provided by this parameter as input is smaller than the minimum required size, the function returns with IID_INVALID_PARAMETERS, but still sets the output value of this parameter to the actual required size.* |

### 3.5.4.10. bk_process_cryptogram

This function processes a received cryptogram in a protected cryptogram format, to retrieve the contained plaintext, using an elliptic curve hybrid decryption scheme. The cryptogram format offers protection for confidentiality, integrity, sender authentication and replay. See Sections 2.5.3.2 and 3.4.3.4 for more background on the security properties depending on the used cryptogram type.

A cryptogram is the single message in a one-pass protocol from a sender to a receiver. The cryptogram processing is based simultaneously on the receiver's private key and the sender's public key. Both private and public key are provided as key codes which shall have been defined over the same elliptic curve, and which shall both have their purpose flags set to allow their use for encryption.

A received cryptogram can only be correctly processed (decrypted and authenticated) if the provided receiver private key and sender public key correspond respectively to the receiver public key and sender private key used to create the cryptogram, e.g. using **bk_generate_cryptogram**.

A call to **bk_process_cryptogram** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_process_cryptogram**, successful or not, will *not* change the state of BroadKey.

**bk_process_cryptogram** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED
- IID_INVALID_COUNTER
- IID_INVALID_PRIVATE_KEY_CODE
- IID_INVALID_PUBLIC_KEY_CODE
- IID_ECC_NOT_ALLOWED
- IID_CURVE_MISMATCH
- IID_INVALID_SENDER
- IID_INVALID_CRYPTOGRAM

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_process_cryptogram(
   const bk_ecc_private_key_code_t * const receiver_private_key_code,
   const bk_ecc_public_key_code_t  * const sender_public_key_code,
         bk_ecc_cryptogram_type_t  * const cryptogram_type,
         uint8_t                   * const counter64,
   const uint8_t                   * const cryptogram,
   const uint32_t                          cryptogram_length,
         uint8_t                   * const plaintext,
         uint32_t                  * const plaintext_length);
```

| Parameters | |
|---|---|
| receiver_private_key_code in | Pointer to a private key code type buffer which holds the elliptic curve private key of the receiver by whom the cryptogram is processed. The private key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *a private key code used for cryptogram processing shall have been created with purpose flags allowing its use for ECDH/encryption (see Section 3.4.3.3).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |

| Parameters | | |
|---|---|---|
| sender_public_key_code | in | Pointer to a public key code type buffer which holds the elliptic curve public key of the sender from whom the to-be-processed cryptogram was received. The public key code type is detailed in Section 3.4.3.5.<br><br>***Note:*** *in order to have secure sender authentication (i.e. assurance to the receiver that the cryptogram comes from the expected sender), the public key contained in sender_public_key_code shall have been verified in an independent manner (e.g. through certificate validation), or it shall come from an independent trusted source.*<br><br>***Note:*** *a public key code used for cryptogram processing shall have been computed or imported with purpose flags allowing its use for ECDH/encryption (see Section 3.4.3.3).*<br><br>***Note:*** *a public key code used for cryptogram processing shall contain a public key defined over the same curve as the simultaneously provided private key code (see Section 3.4.3.1).*<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| cryptogram_type | out | Pointer to a cryptogram type buffer which will hold the cryptogram type which was used to generate the cryptogram. It will contain a valid flag of the bk_ecc_cryptogram_type_t enumeration which is declared in iidbroadkey.h. The supported cryptogram types, and their meaning, are explained in Section 3.4.3.4. |

| Parameters | | |
|---|---|---|
| counter64 | in/out | Pointer to a byte array buffer which holds a 64-bit monotonic counter used for cryptogram replay protection, and will hold the new counter value after successful function completion. Its size in bytes is 8.<br><br>***Note:*** *the counter value is represented in the byte array as a 64-bit unsigned integer in big-endian format (counter[0] contains the most-significant byte).*<br><br>***Note:*** *a separate counter byte array shall be used for each distinct receiver-sender key pair. The calling application needs to retrieve this array from persistent storage before each call to* ***bk_process_cryptogram****, and store back its updated value in persistent storage after the function completes successfully (return code is IID_SUCCESS). Upon first use of a counter array for a receiver-sender pair, the counter array needs to be initialized to all-zero bytes, after the initial validation of the sender's public key.*<br><br>***Note:*** ***bk_process_cryptogram*** *will only be able to successfully process cryptograms which have been generated with a corresponding counter value which is strictly larger than the integer value provided in counter64. This prevents replay of old cryptograms, but also obstructs the ability to receive multiple consecutive cryptograms out of order.* |
| cryptogram | in | Pointer to a byte array buffer which holds the full cryptogram to be processed.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| cryptogram_length | in | Value which specifies the size in bytes of the cryptogram buffer.<br><br>***Note:*** *this value must be the exact size of the cryptogram to-be-processed, as specified in Section 3.4.3.4.* |
| plaintext | out | Pointer to a byte array buffer which will hold the decrypted plaintext. The exact size in bytes of the plaintext depends on the cryptogram length, the used elliptic curve, and the used cryptogram type, as specified in Section 3.4.3.4. |

| Parameters | | |
|---|---|---|
| plaintext_length | in/out | As input, this is the length in bytes of the allocated output buffer pointed to by plaintext. This value is used to check for buffer overflow.<br><br>***Note:*** *the size of the plaintext buffer must be at least long enough to hold a plaintext for the provided cryptogram length, and the used curve and cryptogram type, as specified in Section 3.4.3.4.*<br><br>As output, this is the actual size in bytes of the returned plaintext, as specified in Section 3.4.3.4.<br><br>***Note:*** *if the size provided by this parameter as input is smaller than the minimum required size, the function returns with IID_INVALID_PARAMETERS, but still sets the output value of this parameter to the actual required size.* |

### 3.5.4.11. bk_get_public_key_from_cryptogram

This helper function extracts the sender's public key from a received cryptogram. This function is (optionally) used, prior to cryptogram processing with **bk_process_cryptogram**, to facilitate the validation of the sender's public key. In particular, this function is needed when then receiver upfront has no knowledge of which public key was used by the sender.

***Note:*** *this is an optional helper function. Preferably, a receiver already possesses a trusted copy of the public key used by the expected sender, in which case it is not necessary to use this function.*

***Note:*** *this helper function solely attempts to extract the sender's public key value from a provided cryptogram. The outcome of this function provides no guarantees whatsoever about the correctness/validity/authenticity of the provided cryptogram or the extracted public key.*

***Note:*** *if this function is used to extract a sender's public key, it is extremely important that the retrieved public key is independently validated before calling* ***bk_process_cryptogram*** *with it. This can be done, e.g. by looking up and verifying the certificate corresponding to the public key, or by verifying that the public key matches a trusted copy of that key, e.g. in a local database. Calling* ***bk_process_cryptogram*** *with an unvalidated sender public key voids the sender authentication property of the cryptogram functionality.*

A call to **bk_get_public_key_from_cryptogram** is only allowed when BroadKey is either in the Enrolled or in the Started state; a call from any other state will return IID_NOT_ALLOWED. A call to **bk_get_public_key_from_cryptogram**, successful or not, will *not* change the state of BroadKey.

**bk_get_public_key_from_cryptogram** can exit with one of the following return codes:

- IID_SUCCESS
- IID_INVALID_CRYPTOGRAM
- IID_INVALID_PARAMETERS
- IID_NOT_ALLOWED

See Section 3.3 for the meaning of the return codes.

```
iid_return_t bk_get_public_key_from_cryptogram(
   const bool                  use_point_compression,
   const bk_ecc_curve_t        curve,
   const uint8_t        * const cryptogram,
   const uint32_t              cryptogram_length,
         uint8_t        * const public_key);
```

| Parameters | | |
|---|---|---|
| use_point_compression | in | ***Note:*** *this flag is present for future use compatibility, but is not used for this product version of BroadKey.* ***For this product version, this value has to be set to False (no point compression)****, any other value will result in the return code IID_INVALID_PARAMETERS.* |
| curve | in | Specifies the named elliptic curve on which the provided cryptogram is defined. It will be a curve type of the bk_ecc_curve_t enumeration which is declared in iidbroadkey.h. The possible curves, and their meaning, are explained in Section 3.4.3.1. |
| cryptogram | in | Pointer to a byte array buffer which holds the full cryptogram from which the sender's public key will be extracted.<br><br>***Note:*** *the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.* |
| cryptogram_length | in | Value which specifies the size in bytes of the cryptogram buffer.<br><br>***Note:*** *this value must be the exact size of the cryptogram to-be-processed, as specified in Section 3.4.3.4.* |
| public_key | out | Pointer to a byte array buffer which will hold the extracted public key. The output is in X9.62 binary format. Its exact size in bytes is determined by the specified curve as detailed in Section 3.4.3.1. |

# 4. Integration Guidelines

## 4.1. Integration Considerations

The following needs to be taken into account when using the BroadKey module:

- The PUF SRAM location and size are fixed over the lifetime of the device.
- The PUF SRAM may not be manipulated outside of the control of BroadKey.
- BroadKey does not allocate any memory: it is the responsibility of the caller to allocate all required buffers with the correct sizes.
- The activation code generated during enrollment must be stored by the caller. The availability and integrity of the activation code is a requirement for the use of BroadKey. Loss or corruption of an activation code inevitably leads to the inability to ever re-instantiate the cryptographic context associated to it, potentially resulting in loss of data protected under that context.
- Since the activation code does not contain any sensitive information, further protection (confidentiality, authenticity) is not required.
- Re-enrollment of a device will generate a new activation code and instantiate a new cryptographic context which is incompatible with earlier contexts instantiated on the same device. As a result, device keys will be different from those generated in earlier contexts, and key codes generated in earlier contexts cannot be unwrapped or used in the new context.

## 4.2. Reliability Optimizations

While a chip is in use (meaning powered on) the physical parameters of the device will change slightly over the course of years. The speed of the changes depends heavily on the conditions under which the chip is used (temperature, power supply voltage level, etc.). This process is called (silicon) aging, and it potentially also affects the SRAM start-up behavior.

To improve long-term reliability, *anti-aging measures* are taken during **bk_init**, and during a successful **bk_start** operation. These measures put the SRAM used by BroadKey in an optimized state such that it can remain powered on for a long time without being degraded by the aging process. While in this state, it is still possible to use BroadKey's function in the regular way. It is recommended to not leave BroadKey for a long time in the Uninitialized state, but call **bk_init** at the earliest possible convenient time. The optimal anti-aging measures are only completely taken when a correct AC is provided during **bk_start**.

Contrarily to BroadKey, DemoKey does not implement anti-aging measures for the used SRAM. As a result, it is not recommended to use DemoKey on devices which are powered-on for significant amounts of time, as this might degrade its reliability. Given that DemoKey is intended to be used for demonstration purposes only, this should pose no problem.

**Please note that DemoKey is also not to be used in experiments validating the reliability of BroadKey. Please contact Intrinsic ID for more information regarding reliability testing.**

## 4.3. Power-up Recommendations

To get good SRAM start-up behavior, it is recommended that the voltage power-up curve of the SRAM power supply meets the following guidelines:

- Power supply voltage must be monotonously increasing (no power dips during voltage ramp-up).
- Power supply voltage rise time from 0V to 90% of Vdd must be less than 0.5 ms.
- Before power-on, the power supply must have been at 0V for a sufficiently long time in order to guarantee fresh startup values in the SRAM memory. Typically, a power-off time of 100 ms is enough. For extreme low temperatures (-40ºC and below) the required power-off time may need to be extended to around 500 ms.

# Appendix A. Example Code for Software Development

The example code (of BroadKey) provided below here is fully applicable on DemoKey as well, taking into account the functional limitations as detailed in Section 2.

To help software development, this section provides example code (C-style) that shows how to call the functions of the library. It is not intended to be optimal code for the target processor, nor is it intended to be a meaningful order of operations, e.g. for a particular use case. None of the examples contain error handling. This must be added by the programmer.

The code is based on the functional interfaces described in Section 3.5.

***Note**: Because reading and/or writing to NVM typically require specific control sequences, it is assumed they are handled respectively by read_from_nvm() and write_to_nvm(), which the customer needs to map on the respective NVM I/O functions on his system*

## A.1. Includes and Defines for Example Code

```
/* Includes for BroadKey */
#include "iidreturn_codes.h"
#include "iidbroadkey.h"

/* Define the start address of the SRAM that is used for BroadKey */
#define SRAM_PUF_ADDRESS 0xMMMMMMMM

/* Define the start address of the AC in NVM */
#define AC_NVM_ADDRESS 0xNNNNNNNN

/* Define a variable for functions' return value */
iid_return_t return_value;
```

## A.2. Example Code for BroadKey Initialization

```
/* ... includes and defines of A.1. section ... */

...

/***********************
 * Initialize BroadKey *
 ***********************/
return_value = bk_init((uint8_t * const)SRAM_PUF_ADDRESS,
BK_SRAM_PUF_SIZE_BYTES);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.3. Example Code for BroadKey Enroll and Stop

```
/* ... includes and defines of A.1. section ... */

...

/***********************
 * Initialize BroadKey *
 ***********************/
return_value = bk_init((uint8_t * const)SRAM_PUF_ADDRESS,
BK_SRAM_PUF_SIZE_BYTES);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN uint8_t activation_code[BK_AC_SIZE_BYTES] POST_HIS_ALIGN;

/*********************
 * Enroll the Device *
 *********************/
return_value = bk_enroll(activation_code);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/*****************
 * Stop BroadKey *
 *****************/
return_value = bk_stop();

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/* Store the AC in NVM */
write_to_nvm(AC_NVM_ADDRESS, activation_code, BK_AC_SIZE_BYTES);
```

## A.4. Example Code for BroadKey Start and Stop

```
/* ... includes and defines of A.1. section ... */

/* Enrollment must have been performed before the following code, as shown in
the previous example */

...

/***********************
 * Initialize BroadKey *
 ***********************/
return_value = bk_init((uint8_t * const)SRAM_PUF_ADDRESS,
BK_SRAM_PUF_SIZE_BYTES);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/******************
 * Start BroadKey *
 ******************/
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN uint8_t ac[BK_AC_SIZE_BYTES] POST_HIS_ALIGN;
read_from_nvm(AC_NVM_ADDRESS, ac, BK_AC_SIZE_BYTES);
return_value = bk_start(ac);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/*****************
 * Stop BroadKey *
 *****************/
return_value = bk_stop();

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.5. Example Code for BroadKey Get Key

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/************
 * Get Key *
 ************/
uint8_t key_type = BK_SYM_KEY_TYPE_256;
uint8_t index = 0;
uint8_t key[32];

return_value = bk_get_key(key_type, index, key);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.6. Example Code for BroadKey Get Private Key

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/*******************
 * Get Private Key *
 *******************/
bk_ecc_curve_t curve = BK_ECC_CURVE_NIST_P256;
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN uint8_t private_key[32] POST_HIS_ALIGN;

return_value = bk_get_private_key(curve,
                                  NULL, /* no usage_context is used */
                                  0,    /* no usage context is used */
                                  BK_ECC_KEY_SOURCE_PUF_DERIVED,
                                  private_key);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.7. Example Code for BroadKey Generate Random

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/*******************
 * Generate Random *
 *******************/
uint8_t data_buffer[32];

return_value = bk_generate_random(32, data_buffer);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.8. Example Code for BroadKey Wrap and Unwrap

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/************
 * Wrap Key *
 ************/
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN uint8_t key[32] POST_HIS_ALIGN = { 0x01, 0x02, ... };
PRE_HIS_ALIGN uint8_t key_code[BK_KEY_CODE_HEADER_LENGTH + sizeof(key)]
POST_HIS_ALIGN;
uint8_t index = 0;

return_value = bk_wrap(index,
                       key,
                       sizeof(key),
                       key_code);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/**************
 * Unwrap Key *
 **************/
PRE_HIS_ALIGN uint8_t key_unwrapped[32] POST_HIS_ALIGN;
uint16_t key_unwrapped_length;
uint8_t index_unwrapped;
return_value = bk_unwrap(key_code,
                         key_unwrapped,
                         &key_unwrapped_length,
                         &index_unwrapped);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.9. Example Code for BroadKey Private and Public Key Computation, and Reconstruction without Storage

BroadKey can create and reconstruct PUF-derived private keys without storing them. This section first describes how to create a private key, and compute and export its public key, and then how to rederive the same private key on a subsequent power cycle, without the need to save it.

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/**********************
 * Create Private Key *
 **********************/
bk_ecc_curve_t curve = BK_ECC_CURVE_NIST_P256;
bk_ecc_key_purpose_t purpose_flags = BK_ECC_KEY_PURPOSE_ECDH_AND_ECDSA;
bk_ecc_key_source_t key_source = BK_ECC_KEY_SOURCE_PUF_DERIVED;
uint8_t usage_context[] = {'M', 'y', ' ', 'U', 's', 'a', 'g', 'e', ' ',
                           'C', 'o', 'n', 't', 'e', 'x', 't'};

/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN bk_ecc_private_key_code_t private_key_code POST_HIS_ALIGN;

return_value = bk_create_private_key(curve,
                                     purpose_flags,
                                     usage_context,
                                     sizeof(usage_context),
                                     key_source,
                                     NULL, /* no external private key is used */
                                     &private_key_code);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/***************************
 * Compute Public Key Code *
 ***************************/
PRE_HIS_ALIGN bk_ecc_public_key_code_t public_key_code POST_HIS_ALIGN;

return_value = bk_compute_public_from_private_key(&private_key_code,
                                                  &public_key_code);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
```

```
}

/*********************
 * Export Public Key *
 *********************/
uint8_t public_key[65]; /* BK_ECC_CURVE_NIST_P256 public key computed above */
bk_ecc_curve_t curve_out;
bk_ecc_key_purpose_t purpose_flags_out;

return_value = bk_export_public_key(false, /* point compression is disabled */
                                    &public_key_code,
                                    public_key,
                                    &curve_out,
                                    &purpose_flags_out);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

To re-derive the same private key after a power cycle, without storing the key, perform the following sequence.

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Start have to be performed before the following code, as
shown in previous examples. Also, BroadKey must not be in the stopped state. */

...

/*********************
 * Create Private Key *
 *********************/

bk_ecc_curve_t curve = BK_ECC_CURVE_NIST_P256;
bk_ecc_key_purpose_t purpose_flags = BK_ECC_KEY_PURPOSE_ECDH_AND_ECDSA;
bk_ecc_key_source_t key_source = BK_ECC_KEY_SOURCE_PUF_DERIVED;
uint8_t usage_context[] = {'M', 'y', ' ', 'U', 's', 'a', 'g', 'e', ' ',
                           'C', 'o', 'n', 't', 'e', 'x', 't'};


/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN bk_ecc_private_key_code_t private_key_code POST_HIS_ALIGN;


return_value = bk_create_private_key(curve,
                                     purpose_flags,
                                     usage_context,
                                     sizeof(usage_context),
                                     key_source,
                                     NULL, /* no external private key is used */
```

```
                                    &private_key_code);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

The following items need to be the same in order for the private key to reconstruct to the same value:

- same chip and SRAM buffer address as during enrollment
- same Activation Code used for calling bk_start
- same usage_context
- same curve and purpose_flags
- key source needs to be set to BK_ECC_KEY_SOURCE_PUF_DERIVED

If public_key_code and/or public_key are also not stored locally they can be recomputed as needed using the respective “Compute Public Key Code” and the “Export Public Key” sequences from above.

## A.10. Example Code for BroadKey Public Key Import

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/**********************
 * Import Public Key *
 **********************/
bk_ecc_curve_t curve = BK_ECC_CURVE_NIST_P256;
bk_ecc_key_purpose_t purpose_flags = BK_ECC_KEY_PURPOSE_ECDH_AND_ECDSA;
uint8_t public_key[65] = {
    0x04, ...
};
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN bk_ecc_public_key_code_t public_key_code POST_HIS_ALIGN;

return_value = bk_import_public_key(curve,
                                    purpose_flags,
                                    public_key,
                                    &public_key_code);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.11. Example Code for BroadKey Derive Public Key

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/*********************
 * Derive Public Key *
 *********************/
bk_ecc_curve_t curve = BK_ECC_CURVE_NIST_P256;
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN uint8_t private_key[32] POST_HIS_ALIGN = {
    ...
};
uint8_t public_key[65];

return_value = bk_derive_public_key(false, /* point compression is disabled */
                                    curve,
                                    private_key,
                                    public_key);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.12. Example Code for BroadKey ECDSA Sign and Verify

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/****************
 * Sign Message *
 ****************/
uint8_t message[32] = { ... };
uint8_t signature[64]; /* size = 64, assuming that the private key of 0. is
                        * used, which belongs to BK_ECC_CURVE_NIST_P256.
                        */
uint16_t signature_length = sizeof(signature);

return_value = bk_ecdsa_sign(&private_key_code, /* generated in 0. */
                             false, /* non-deterministic signing algorithm */
                             message,
                             sizeof(message),
                             false, /* message is not hashed */
                             signature,
                             &signature_length);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}

/************************
 * Verify Signed Message *
 ************************/
return_value = bk_ecdsa_verify(&public_key_code, /* generated in 0. */
                               message,
                               sizeof(message),
                               false, /* message is not hashed */
                               signature,
                               sizeof(signature));

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.13. Example Code for BroadKey ECDH

```
/* ... includes and defines of A.1. section ... */

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/********
 * ECDH *
 ********/
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN bk_ecc_private_key_code_t private_key_code1 POST_HIS_ALIGN;
PRE_HIS_ALIGN bk_ecc_public_key_code_t public_key_code2 POST_HIS_ALIGN;
uint8_t shared_secret[32]; /* size = 32, assuming that the key pairs have been
                            * generated with BK_ECC_CURVE_NIST_P256.
                            */

/* private_key_code1 belongs to key pair 1, while public_key_code2 belongs
 * to key pair 2.
 * public_key_code2 cannot be derived from private_key_code1.
 * Generation of the 2 different key pairs is omitted. An example can be
 * found in A.9.
 */

return_value = bk_ecdh_shared_secret(&private_key_code1,
                                     &public_key_code2,
                                     shared_secret);

if (IID_SUCCESS != return_value) {
    /* ... handle error ... */
}
```

## A.14. Example Code for BroadKey Cryptogram Generation and Processing

Among other things, BroadKey's cryptogram functionality features an anti-rollback option: **bk_generate_cryptogram** and **bk_process_cryptogram** can protect messages against replay attacks (rejecting repeat cryptograms and old cryptograms). Such an anti-rollback feature typically uses monotonic counters or timestamps in order achieve that. In the case of BroadKey, monotonic counters are used. This section describes:

- how to use the cryptogram functions
- how to initialize the counters on both sides (generate and process counters set in NVM to 0)
- when not to save the counters to NVM (upon error)
- how to bypass anti-rollback (by always passing 0 to _process, rather than using NVM).

When the rollback protection functionality is not needed, it can be disabled by passing 0 for the counter64 value to **bk_process_cryptogram**. This will result in all cryptograms being processed successfully, including old or repeat messages. Similarly, when anti-rollback is not needed the sender side can simply use 0 as the value of counter64 for simplicity. There is no need to store counter64 to NVM when the anti-rollback functionality is not used.

When anti-rollback is used the *sender* sequence is as follows:

Before the very first cryptogram to a given receiver is generated, as indicated by the first use of the receiver's public key:

```
/* Define the size (in bytes) of a cryptogram counter */
#define COUNTER64_SIZE_BYTES (64 / 8)

/* Define the address of the sender's cryptogram counter in NVM */
#define COUNTER64_SENDER_ADDRESS 0xNNNNNNNN

uint8_t counter64_sender[COUNTER64_SIZE_BYTES];
memset(counter64_sender, 0, sizeof(counter64_sender));
write_to_nvm(COUNTER64_SENDER_ADDRESS, counter64_sender, COUNTER64_SIZE_BYTES);
```

And then, for sending messages as cryptograms:

```
/* ... includes and defines of A.1. section ... */

/* Define the address of the sender's cryptogram counter in NVM */
#define COUNTER64_SENDER_ADDRESS 0xNNNNNNNN

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...
```

```
/**********************
 * Generate Cryptogram *
 **********************/
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN bk_ecc_private_key_code_t sender_private_key_code POST_HIS_ALIGN;
PRE_HIS_ALIGN bk_ecc_public_key_code_t receiver_public_key_code POST_HIS_ALIGN;
PRE_HIS_ALIGN uint8_t plaintext[32] POST_HIS_ALIGN = { ... };
PRE_HIS_ALIGN uint8_t cryptogram[BK_CRYPTOGRAM_HEADER_SIZE_BYTES +
                                 64 +
                                 sizeof(plaintext)]
                                 POST_HIS_ALIGN; /* 64 is added to the length,
                                                  * assuming that the key pairs
                                                  * have been generated with
                                                  * BK_ECC_CURVE_NIST_P256.
                                                  */
uint32_t cryptogram_length = sizeof(cryptogram);
uint8_t counter64_sender[COUNTER64_SIZE_BYTES];

/* Generation of private and public key codes is omitted. An example
 * can be be found inA.9.
 */

read_from_nvm(COUNTER64_SENDER_ADDRESS, counter64_sender, COUNTER64_SIZE_BYTES);

return_value = bk_generate_cryptogram(&receiver_public_key_code,
                                      &sender_private_key_code,
                                      BK_ECC_CRYPTOGRAM_TYPE_ECDH_STATIC,
                                      counter64_sender,
                                      plaintext,
                                      sizeof(plaintext),
                                      cryptogram,
                                      &cryptogram_length);

if (IID_SUCCESS == return_value) {
    write_to_nvm(COUNTER64_SENDER_ADDRESS, counter64_sender,
COUNTER64_SIZE_BYTES);
    send_or_store(cryptogram, &cryptogram_length);}
else {
    /* ... handle error ... */
}
```

When anti-rollback is used the *receiver* sequence is as follows:

Before the very first cryptogram from a given sender is processed, as indicated by the first use of the sender's public key:

```
/* Define the address of the receiver's cryptogram counter in NVM */
#define COUNTER64_RECEIVER_ADDRESS 0xNNNNNNNN

uint8_t counter64_receiver[COUNTER64_SIZE_BYTES];
memset(counter64_receiver, 0, sizeof(counter64_receiver));
write_to_nvm(COUNTER64_RECEIVER_ADDRESS, counter64_receiver,
COUNTER64_SIZE_BYTES);
```

And then, for processing messages inside received cryptograms:

```
/* ... includes and defines of A.1. section ... */

/* Define the address of the receiver’s cryptogram counter in NVM */
#define COUNTER64_RECEIVER_ADDRESS 0xNNNNNNNN

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...

/**********************
 * Process Cryptogram *
 **********************/
PRE_HIS_ALIGN bk_ecc_private_key_code_t receiver_private_key_code
POST_HIS_ALIGN;
PRE_HIS_ALIGN bk_ecc_public_key_code_t sender_public_key_code POST_HIS_ALIGN;
PRE_HIS_ALIGN uint8_t plaintext_decrypted[32] POST_HIS_ALIGN;
bk_ecc_cryptogram_type_t cryptogram_type;

uint8_t counter64_receiver[COUNTER64_SIZE_BYTES];

uint32_t plaintext_decrypted_length = sizeof(plaintext_decrypted);

/* Generation of private and public key codes is omitted. An example can be
found
 * in A.9. */

read_from_nvm(COUNTER64_RECEIVER_ADDRESS, counter64_receiver,
              COUNTER64_SIZE_BYTES);

return_value = bk_process_cryptogram(&receiver_private_key_code,
                                     &sender_public_key_code,
                                     &cryptogram_type,
                                     counter64_receiver,
                                     cryptogram,
                                     sizeof(cryptogram),
                                     plaintext_decrypted,
                                     &plaintext_decrypted_length);

if (IID_SUCCESS == return_value) {
   write_to_nvm(COUNTER64_RECEIVER_ADDRESS, counter64_receiver,
                COUNTER64_SIZE_BYTES);
   use(plaintext_decrypted, &plaintext_decrypted_length);
}
else {
   /* ... handle error ... */
}
```

## A.15. Example Code for BroadKey Get Public Key From Cryptogram and Multiple Sender Authentication

This section describes how to correctly do sender authentication for received cryptograms. It details how to extract the sender's public key from a cryptogram and check it against a list of trusted sender public keys, before conditionally invoking **bk_process_cryptogram** only when the extracted key is found in the list. Verifying a public key against a trusted list is one possible way of authenticating the sender, another option would be validating the certificate chain belonging to that public key (if any) with a trusted PKI root key.

```
/* ... includes and defines of A.1. section ... */

/* Define the size (in bytes) of a cryptogram counter */
#define COUNTER64_SIZE_BYTES (64 / 8)

/* Define the address of the receiver's cryptogram counter in NVM, for a given
sender index */
#define COUNTER64_RECEIVER_ADDRESS(i) ( 0xNNNNNNNN + COUNTER64_SIZE_BYTES*(i) )

/* Define the number of public keys in the trusted sender list */
#define N_PUB_KEYS 5

/* Initialization and Enrollment/Start have to be performed before the following
code, as shown in previous examples. Also, BroadKey must not be in the stopped
state. */

...
/* PRE_HIS_ALIGN and POST_HIS_ALIGN macros can be used to align a variable
address to 32 bits */
PRE_HIS_ALIGN bk_ecc_private_key_code_t receiver_private_key_code
POST_HIS_ALIGN;
PRE_HIS_ALIGN bk_ecc_public_key_code_t sender_public_key_code POST_HIS_ALIGN;
PRE_HIS_ALIGN uint8_t plaintext_decrypted[32] POST_HIS_ALIGN;
bk_ecc_cryptogram_type_t cryptogram_type;

uint8_t counter64_receiver[COUNTER64_SIZE_BYTES];

uint32_t plaintext_decrypted_length = sizeof(plaintext_decrypted);

/* Generation of private and public key codes is omitted. An example can be
found in A.9. */

/*********************************
 * Get Public Key From Cryptogram *
 *********************************/
bk_ecc_curve_t = BK_ECC_CURVE_NIST_P256;
PRE_HIS_ALIGN uint8_t cryptogram[...] POST_HIS_ALIGN;
uint32_t cryptogram_length = sizeof(cryptogram);
typedef uint8_t pub_key_256_t[65];
pub_key_256_t public_key;
pub_key_256_t trusted_public_keys[N_PUB_KEYS] = { {1}, {2}, {3}, {4}, {5} };
```

```
int i, j;

/* cryptogram[...] is loaded with received cryptogram array */
return_value = bk_get_public_key_from_cryptogram(false,
                                                 curve,
                                                 cryptogram,
                                                 cryptogram_length,
                                                 public_key);

if (IID_SUCCESS != return_value) {
/* ... handle error ... */
}

/*****************************************
 * Search the list of trusted Public Keys *
 *****************************************/
i = -1;
for(j = 0; j<N_PUB_KEYS; j++) {
    if(0 == memcmp(public_key, trusted_public_keys[j], sizeof(public_key)) ) {
        i = j;
        break;
    }
}

/*****************************************
 * Process for trusted sender public key #i *
 *****************************************/
if (0 <= i) {
    /**********************
    * Import Public Key *
    **********************/
    return_value = bk_import_public_key(curve,
                                        purpose_flags,
                                        public_key,
                                        &sender_public_key_code);

    if (IID_SUCCESS != return_value) {
        /* ... handle error ... */
    }
    else {
        /**********************
         * Process Cryptogram *
         **********************/
read_from_nvm(COUNTER64_RECEIVER_ADDRESS(i), counter64_receiver,
              COUNTER64_SIZE_BYTES);

        return_value = bk_process_cryptogram(&receiver_private_key_code,
                                             &sender_public_key_code,
                                             &cryptogram_type,
                                             counter64_receiver,
                                             cryptogram,
                                             sizeof(cryptogram),
```

```
                                            plaintext_decrypted,
                                            &plaintext_decrypted_length);

        if (IID_SUCCESS == return_value) {
            /* save counter to sender index i */
            write_to_nvm(COUNTER64_RECEIVER_ADDRESS(i), counter64_receiver,
                         COUNTER64_SIZE_BYTES);

            /* use plaintext_decrypted in sender secure key/code slot #i */
            use(i, plaintext_decrypted, &plaintext_decrypted_length);
        }
        else {
            /* ... handle error ... */
        }
    }
}
```