

BK v2.9

API Reference Manual

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

Doc version	1.5
Status	Approved
Reference	IID-BKPR2-9-API-15-gowin

For internal use by customer only
Confidential



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, QuiddiKey RNG, Apollo, BK, BK-Demo, Zign, Zign RNG, Zign Tag, Citadel, iRNG and other designated brands included herein are trademarks of Intrinsic ID B.V. All other trademarks are the property of their respective owners.



Overview

This document describes the Application Programming Interface (API) and usage instructions for the Intrinsic ID BK product. It has been automatically generated using Doxygen.

Table of Contents	3
1 Introduction	4
1.1 Document Scope	4
1.2 Product Brief	4
1.3 Function Groups	4
2 Module Documentation	5
2.1 BK API	5
2.1.1 Detailed Description	12
2.1.2 Data Structure Documentation	13
2.1.3 Macro Definition Documentation	13
2.1.4 Typedef Documentation	31
2.1.5 Enumeration Type Documentation	33
2.1.6 Function Documentation	36
2.2 Return Codes	70
2.2.1 Detailed Description	71
2.2.2 Enumeration Type Documentation	71
2.3 Compiler Attributes	74
2.3.1 Detailed Description	74
2.3.2 Macro Definition Documentation	75
3 Example Documentation	76
3.1 iid_bk_examples_standalone.c	76
3.2 iid_bk_examples_mbedtls.c	79
3.3 iid_bk_examples_wolfssl.c	84

©2023 Intrinsic ID B.V. – all rights reserved.
The information contained herein is proprietary to Intrinsic ID B.V. and is made available under an obligation of confidentiality.
Receipt of this document does not imply any license under any intellectual property rights of Intrinsic ID B.V.



1. Introduction

1.1. Document Scope

This documentation explains the use of the BK embedded software (SW) library and is mainly intended for SW developers deploying BK in their application project. Targeted readers are expected to understand the basics of embedded SW development.

The scope of this documentation is confined to programming interfaces. It does not provide an in-depth description of the architecture.

1.2. Product Brief

BK 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.

In addition to device key generation and management, full-featured BK also enables various standard cryptographic operations, all protected by the device-unique security boundary rooted in the SRAM PUF. Available functionality includes, among other things:

- secure wrapping and unwrapping of application secrets, enabling secure storage
- symmetric key data encryption and decryption with AES
- symmetric key data authentication and verification with HMAC-SHA256 and CMAC-AES
- elliptic curve message signing and verification with ECDSA
- elliptic curve key agreement with ECDH
- elliptic curve secure messaging with Intrinsic ID's custom cryptogram format

This document provides the *User Manual* for the application programming interface (API) of the BK software library.

1.3. Function Groups

The functions declared in BK's public API are divided in the following subgroups:

- **BK Core Functions:** this group contains the functions for inspecting and controlling the BK module, but which do not perform any cryptographic operations. This function group includes `bk_get_product_info()`, which can always be called, and the functions for managing the operational state of BK, as described in the BK data sheet (IID-BK2-9-DS): `bk_init()`, `bk_enroll()`, `bk_start()` and `bk_stop()`.
- **BK Randomness and Device-Key Generation Functions:** this group contains the basic functions for generating random numbers and device-unique keys with BK. Functions in this group are only available when a cryptographic context is instantiated (after a successful call to `bk_enroll()` or `bk_start()`).
- **BK Symmetric Key Crypto Functions:** this group contains the functions for performing symmetric key crypto operations within an instantiated BK cryptographic context (after a successful call to `bk_enroll()` or `bk_start()`).
- **BK Elliptic Curve Crypto Functions:** this group contains the functions for performing elliptic curve (public key) crypto operations within an instantiated BK cryptographic context (after a successful call to `bk_enroll()` or `bk_start()`).



2. Module Documentation

2.1. BK API

BK Top-Level API.

Data Structures

- struct `bk_certificate_subject_t`
Certificate (subject) distinguished name. [More...](#)

Macros

- #define `BK_SECURITY_SIZE_BITS` 256
Root security strength of this BK configuration.
- #define `BK_SRAM_PUF_SIZE_BYTES` 1024
Size in bytes of the SRAM PUF allocation.
- #define `BK_AC_SIZE_BYTES` 968
Size in bytes of activation code buffer.
- #define `BK_CORE_UID_BYTES` 32
Size in bytes of the core unique identifier.
- #define `BK_SYM_MAX_KEY_BYTES` 32
Maximum size in bytes for this configuration, of a symmetric key used by BK.
- #define `BK_SYM_MAX_KEY_WORDS` 8
Maximum size in words for this configuration, of a symmetric key used by BK.
- #define `BK_SYM_MAX_KEY_CODE_BYTES` (44 + `BK_SYM_MAX_KEY_BYTES`)
Size in bytes of a symmetric key code capable of storing the maximum sized symmetric key for this configuration.
- #define `BK_USER_KEY_CODE_NONKEY_BYTES` (44)
Size in bytes of the header of a key code used to wrap external keys.
- #define `BK_ECC_CURVE_SECP521R1_N_BYTES` 66
Size in bytes, of an SECP521R1 elliptic curve scalar, smaller than the order of the curve.
- #define `BK_ECC_CURVE_SECP521R1_N_WORDS` 17
Size in words, of an SECP521R1 elliptic curve scalar, smaller than the order of the curve.
- #define `BK_ECC_CURVE_SECP521R1_P_BYTES` 66
Size in bytes, of an SECP521R1 elliptic curve field element that represents a coordinate of a curve point.
- #define `BK_ECC_CURVE_SECP521R1_P_WORDS` 17
Size in words, of an SECP521R1 elliptic curve field element that represents a coordinate of a curve point.
- #define `BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_BYTES` 66
Size in bytes, of an SECP521R1 elliptic curve private key.
- #define `BK_ECC_CURVE_SECP521R1_PUBLIC_KEY_BYTES` 133
Size in bytes, of an SECP521R1 elliptic curve public key.
- #define `BK_ECC_CURVE_SECP521R1_SIGNATURE_BYTES` 132
Size in bytes, of an SECP521R1 elliptic curve signature.
- #define `BK_ECC_CURVE_SECP521R1_SHARED_SECRET_BYTES` 66
Size in bytes, of an SECP521R1 elliptic curve shared secret.
- #define `BK_ECC_CURVE_SECP521R1_STATIC_CRYPTOGAM_HEADER_BYTES` 212
Size in bytes, of an SECP521R1 elliptic curve static cryptogram header.
- #define `BK_ECC_CURVE_SECP521R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES` 344
Size in bytes, of an SECP521R1 elliptic curve ephemeral cryptogram header.
- #define `BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES` 24
Size in bytes, of an SECP521R1 elliptic curve private key info ASN.1 header.
- #define `BK_ECC_CURVE_SECP521R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES` 25
Size in bytes, of an SECP521R1 elliptic curve public key info ASN.1 header.
- #define `BK_ECC_CURVE_SECP521R1_SIG_VALUE_DER_HEADER_BYTES` 9
Size in bytes, of an SECP521R1 elliptic curve sig value ASN.1 header.
- #define `BK_ECC_CURVE_SECP521R1_SIGNATURE_DER_HEADER_BYTES` 13
Size in bytes, of an SECP521R1 elliptic curve signature ASN.1 header.
- #define `BK_ECC_CURVE_SECP384R1_N_BYTES` 48



- Size in bytes, of an SECP384R1 elliptic curve scalar, smaller than the order of the curve.*
- #define **BK_ECC_CURVE_SECP384R1_N_WORDS** 12
- Size in words, of an SECP384R1 elliptic curve scalar, smaller than the order of the curve.*
- #define **BK_ECC_CURVE_SECP384R1_P_BYTES** 48
- Size in bytes, of an SECP384R1 elliptic curve field element that represents a coordinate of a curve point.*
- #define **BK_ECC_CURVE_SECP384R1_P_WORDS** 12
- Size in words, of an SECP384R1 elliptic curve field element that represents a coordinate of a curve point.*
- #define **BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_BYTES** 48
- Size in bytes, of an SECP384R1 elliptic curve private key.*
- #define **BK_ECC_CURVE_SECP384R1_PUBLIC_KEY_BYTES** 97
- Size in bytes, of an SECP384R1 elliptic curve public key.*
- #define **BK_ECC_CURVE_SECP384R1_SIGNATURE_BYTES** 96
- Size in bytes, of an SECP384R1 elliptic curve signature.*
- #define **BK_ECC_CURVE_SECP384R1_SHARED_SECRET_BYTES** 48
- Size in bytes, of an SECP384R1 elliptic curve shared secret.*
- #define **BK_ECC_CURVE_SECP384R1_STATIC_CRYPTOGAM_HEADER_BYTES** 176
- Size in bytes, of an SECP384R1 elliptic curve static cryptogram header.*
- #define **BK_ECC_CURVE_SECP384R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES** 272
- Size in bytes, of an SECP384R1 elliptic curve ephemeral cryptogram header.*
- #define **BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES** 22
- Size in bytes, of an SECP384R1 elliptic curve private key info ASN.1 header.*
- #define **BK_ECC_CURVE_SECP384R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES** 23
- Size in bytes, of an SECP384R1 elliptic curve public key info ASN.1 header.*
- #define **BK_ECC_CURVE_SECP384R1_SIG_VALUE_DER_HEADER_BYTES** 8
- Size in bytes, of an SECP384R1 elliptic curve sig value ASN.1 header.*
- #define **BK_ECC_CURVE_SECP384R1_SIGNATURE_DER_HEADER_BYTES** 11
- Size in bytes, of an SECP384R1 elliptic curve signature ASN.1 header.*
- #define **BK_ECC_CURVE_SECP256R1_N_BYTES** 32
- Size in bytes, of an SECP256R1 elliptic curve scalar, smaller than the order of the curve.*
- #define **BK_ECC_CURVE_SECP256R1_N_WORDS** 8
- Size in words, of an SECP256R1 elliptic curve scalar, smaller than the order of the curve.*
- #define **BK_ECC_CURVE_SECP256R1_P_BYTES** 32
- Size in bytes, of an SECP256R1 elliptic curve field element that represents a coordinate of a curve point.*
- #define **BK_ECC_CURVE_SECP256R1_P_WORDS** 8
- Size in words, of an SECP256R1 elliptic curve field element that represents a coordinate of a curve point.*
- #define **BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_BYTES** 32
- Size in bytes, of an SECP256R1 elliptic curve private key.*
- #define **BK_ECC_CURVE_SECP256R1_PUBLIC_KEY_BYTES** 65
- Size in bytes, of an SECP256R1 elliptic curve public key.*
- #define **BK_ECC_CURVE_SECP256R1_SIGNATURE_BYTES** 64
- Size in bytes, of an SECP256R1 elliptic curve signature.*
- #define **BK_ECC_CURVE_SECP256R1_SHARED_SECRET_BYTES** 32
- Size in bytes, of an SECP256R1 elliptic curve shared secret.*
- #define **BK_ECC_CURVE_SECP256R1_STATIC_CRYPTOGAM_HEADER_BYTES** 144
- Size in bytes, of an SECP256R1 elliptic curve static cryptogram header.*
- #define **BK_ECC_CURVE_SECP256R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES** 208
- Size in bytes, of an SECP256R1 elliptic curve ephemeral cryptogram header.*
- #define **BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES** 24
- Size in bytes, of an SECP256R1 elliptic curve private key info ASN.1 header.*
- #define **BK_ECC_CURVE_SECP256R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES** 26
- Size in bytes, of an SECP256R1 elliptic curve public key info ASN.1 header.*
- #define **BK_ECC_CURVE_SECP256R1_SIG_VALUE_DER_HEADER_BYTES** 8
- Size in bytes, of an SECP256R1 elliptic curve sig value ASN.1 header.*
- #define **BK_ECC_CURVE_SECP256R1_SIGNATURE_DER_HEADER_BYTES** 11
- Size in bytes, of an SECP256R1 elliptic curve signature ASN.1 header.*
- #define **BK_ECC_CURVE_SECP224R1_N_BYTES** 28
- Size in bytes, of an SECP224R1 elliptic curve scalar, smaller than the order of the curve.*
- #define **BK_ECC_CURVE_SECP224R1_N_WORDS** 7
- Size in words, of an SECP224R1 elliptic curve scalar, smaller than the order of the curve.*
- #define **BK_ECC_CURVE_SECP224R1_P_BYTES** 28
- Size in bytes, of an SECP224R1 elliptic curve field element that represents a coordinate of a curve point.*
- #define **BK_ECC_CURVE_SECP224R1_P_WORDS** 7
- Size in words, of an SECP224R1 elliptic curve field element that represents a coordinate of a curve point.*
- #define **BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_BYTES** 28
- Size in bytes, of an SECP224R1 elliptic curve private key.*
- #define **BK_ECC_CURVE_SECP224R1_PUBLIC_KEY_BYTES** 57



- *Size in bytes, of an SECP224R1 elliptic curve public key.*
• #define **BK_ECC_CURVE_SECP224R1_SIGNATURE_BYTES** 56
- *Size in bytes, of an SECP224R1 elliptic curve signature.*
• #define **BK_ECC_CURVE_SECP224R1_SHARED_SECRET_BYTES** 28
- *Size in bytes, of an SECP224R1 elliptic curve shared secret.*
• #define **BK_ECC_CURVE_SECP224R1_STATIC_CRYPTOGAM_HEADER_BYTES** 136
- *Size in bytes, of an SECP224R1 elliptic curve static cryptogram header.*
• #define **BK_ECC_CURVE_SECP224R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES** 192
- *Size in bytes, of an SECP224R1 elliptic curve ephemeral cryptogram header.*
• #define **BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES** 21
- *Size in bytes, of an SECP224R1 elliptic curve private key info ASN.1 header.*
• #define **BK_ECC_CURVE_SECP224R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES** 23
- *Size in bytes, of an SECP224R1 elliptic curve public key info ASN.1 header.*
• #define **BK_ECC_CURVE_SECP224R1_SIG_VALUE_DER_HEADER_BYTES** 8
- *Size in bytes, of an SECP224R1 elliptic curve sig value ASN.1 header.*
• #define **BK_ECC_CURVE_SECP224R1_SIGNATURE_DER_HEADER_BYTES** 11
- *Size in bytes, of an SECP224R1 elliptic curve signature ASN.1 header.*
• #define **BK_ECC_CURVE_SECP192R1_N_BYTES** 24
- *Size in bytes, of an SECP192R1 elliptic curve scalar, smaller than the order of the curve.*
• #define **BK_ECC_CURVE_SECP192R1_N_WORDS** 6
- *Size in words, of an SECP192R1 elliptic curve scalar, smaller than the order of the curve.*
• #define **BK_ECC_CURVE_SECP192R1_P_BYTES** 24
- *Size in bytes, of an SECP192R1 elliptic curve field element that represents a coordinate of a curve point.*
• #define **BK_ECC_CURVE_SECP192R1_P_WORDS** 6
- *Size in words, of an SECP192R1 elliptic curve field element that represents a coordinate of a curve point.*
• #define **BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_BYTES** 24
- *Size in bytes, of an SECP192R1 elliptic curve private key.*
• #define **BK_ECC_CURVE_SECP192R1_PUBLIC_KEY_BYTES** 49
- *Size in bytes, of an SECP192R1 elliptic curve public key.*
• #define **BK_ECC_CURVE_SECP192R1_SIGNATURE_BYTES** 48
- *Size in bytes, of an SECP192R1 elliptic curve signature.*
• #define **BK_ECC_CURVE_SECP192R1_SHARED_SECRET_BYTES** 24
- *Size in bytes, of an SECP192R1 elliptic curve shared secret.*
• #define **BK_ECC_CURVE_SECP192R1_STATIC_CRYPTOGAM_HEADER_BYTES** 128
- *Size in bytes, of an SECP192R1 elliptic curve static cryptogram header.*
• #define **BK_ECC_CURVE_SECP192R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES** 176
- *Size in bytes, of an SECP192R1 elliptic curve ephemeral cryptogram header.*
• #define **BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES** 24
- *Size in bytes, of an SECP192R1 elliptic curve private key info ASN.1 header.*
• #define **BK_ECC_CURVE_SECP192R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES** 26
- *Size in bytes, of an SECP192R1 elliptic curve public key info ASN.1 header.*
• #define **BK_ECC_CURVE_SECP192R1_SIG_VALUE_DER_HEADER_BYTES** 8
- *Size in bytes, of an SECP192R1 elliptic curve sig value ASN.1 header.*
• #define **BK_ECC_CURVE_SECP192R1_SIGNATURE_DER_HEADER_BYTES** 11
- *Size in bytes, of an SECP192R1 elliptic curve signature ASN.1 header.*
• #define **BK_ECC_MAX_CURVE_SIZE_N_BYTES** 66
- *Maximum size in bytes for this configuration, of an elliptic curve scalar, smaller than the order of the curve.*
• #define **BK_ECC_MAX_CURVE_SIZE_N_WORDS** 17
- *Maximum size in words for this configuration, of an elliptic curve scalar, smaller than the order of the curve.*
• #define **BK_ECC_MAX_CURVE_SIZE_P_BYTES** 66
- *Maximum size in bytes for this configuration, of an elliptic curve field element that represents a coordinate of a curve point.*
• #define **BK_ECC_MAX_CURVE_SIZE_P_WORDS** 17
- *Maximum size in words for this configuration, of an elliptic curve field element that represents a coordinate of a curve point.*
• #define **BK_ECC_MAX_CURVE_POINT_BYTES** 132
- *Maximum size in bytes for this configuration, of an elliptic curve field element that represents a curve point.*
• #define **BK_ECC_MAX_CURVE_POINT_WORDS** 33
- *Maximum size in words for this configuration, of an elliptic curve field element that represents a curve point.*
• #define **BK_ECC_MAX_SIG_VALUE_DER_HEADER_BYTES** 9
- *Maximum size in bytes for this configuration, of an elliptic curve sig value ASN.1 header.*
• #define **BK_ECC_MAX_SIGNATURE_DER_HEADER_BYTES** 13
- *Maximum size in bytes for this configuration, of an elliptic curve signature ASN.1 header.*
• #define **BK_ECC_MAX_PUBLIC_KEY_INFO_DER_HEADER_BYTES** 26
- *Maximum size in bytes for this configuration, of an elliptic curve public key info ASN.1 header.*
• #define **BK_ECC_MAX_PRIVATE_KEY_INFO_DER_HEADER_BYTES** 24
- *Maximum size in bytes for this configuration, of an elliptic curve private key info ASN.1 header.*
• #define **BK_ECC_MAX_CURVE_SIZE_BYTES** **BK_ECC_MAX_CURVE_SIZE_N_BYTES**



- Maximum size in bytes for this configuration, of an elliptic curve scalar, smaller than the order of the curve.*
- **#define BK_ECC_MAX_CURVE_SIZE_WORDS BK_ECC_MAX_CURVE_SIZE_N_WORDS**
- Maximum size in bytes for this configuration, of an elliptic curve field element that represents a coordinate of a curve point.*
- **#define BK_ECC_PRIVATE_KEY_BYTES BK_ECC_MAX_CURVE_SIZE_N_BYTES**
- Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve private key.*
- **#define BK_ECC_PUBLIC_KEY_BYTES BK_ECC_MAX_CURVE_POINT_BYTES**
- Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve public key.*
- **#define BK_ECC_STD_PUBLIC_KEY_BYTES (1 + BK_ECC_MAX_CURVE_POINT_BYTES)**
- Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve x9.62 public key.*
- **#define BK_ECC_SIGNATURE_BYTES BK_ECC_MAX_CURVE_POINT_BYTES**
- Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDSA signature.*
- **#define BK_ECC_SHARED_SECRET_BYTES BK_ECC_MAX_CURVE_SIZE_P_BYTES**
- Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDH shared secret.*
- **#define BK_ECC_PRIVATE_KEY_WORDS BK_ECC_MAX_CURVE_SIZE_N_WORDS**
- Size in words for this configuration, of the store capable of holding the maximum sized elliptic curve private key.*
- **#define BK_ECC_PUBLIC_KEY_WORDS BK_ECC_MAX_CURVE_POINT_WORDS**
- Size in words for this configuration, of the store capable of holding the maximum sized elliptic curve public key.*
- **#define BK_ECC_SIGNATURE_WORDS BK_ECC_MAX_CURVE_POINT_WORDS**
- Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDSA signature.*
- **#define BK_ECC_SHARED_SECRET_WORDS BK_ECC_MAX_CURVE_SIZE_P_WORDS**
- Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDH shared secret.*
- **#define BK_ECC_DER_SIG_VALUE_BYTES (BK_ECC_MAX_SIG_VALUE_DER_HEADER_BYTES + BK_ECC_SIGNATURE_BYTES)**
- Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded sig value.*
- **#define BK_ECC_DER_SIGNATURE_BYTES (BK_ECC_MAX_SIGNATURE_DER_HEADER_BYTES + BK_ECC_SIGNATURE_BYTES)**
- Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded signature.*
- **#define BK_ECC_DER_PUBLIC_KEY_INFO_BYTES (BK_ECC_MAX_PUBLIC_KEY_INFO_DER_HEADER_BYTES + BK_ECC_STD_PUBLIC_KEY_BYTES)**
- Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded public key info.*
- **#define BK_ECC_DER_PRIVATE_KEY_INFO_BYTES (BK_ECC_MAX_PRIVATE_KEY_INFO_DER_HEADER_BYTES + BK_ECC_PRIVATE_KEY_BYTES + BK_ECC_STD_PUBLIC_KEY_BYTES)**
- Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded public key info.*
- **#define BK_ECC_KEY_CODE_HEADER_BYTES 48**
- Size in bytes of the header of an elliptic curve key code.*
- **#define BK_ECC_KEY_CODE_HEADER_WORDS 12**
- Size in words of the header of an elliptic curve key code.*
- **#define BK_ECC_PRIVATE_KEY_CODE_SIZE_BYTES (4 * (BK_ECC_KEY_CODE_HEADER_WORDS + BK_ECC_MAX_CURVE_SIZE_N_WORDS))**
- Size in bytes for this configuration, of an elliptic curve private key code capable of storing the maximum sized private key.*
- **#define BK_ECC_PUBLIC_KEY_PACK_BYTES (BK_ECC_MAX_CURVE_POINT_BYTES)**
- Size in bytes for this configuration, of the representation capable of holding the maximum sized elliptic curve public key.*
- **#define BK_ECC_PUBLIC_KEY_CODE_SIZE_BYTES (4 * (BK_ECC_KEY_CODE_HEADER_WORDS + BK_ECC_MAX_CURVE_POINT_WORDS))**
- Size in bytes for this configuration, of an elliptic curve public key code capable of storing the maximum sized public key.*
- **#define BK_HYBRID_CRYPTOGRAM_HEADER_BYTES 80**
- Size in bytes of the fixed part of the header of a cryptogram.*
- **#define BK_HYBRID_MAX_STATIC_CRYPTOGRAM_HEADER_SIZE_BYTES (BK_HYBRID_CRYPTOGRAM_HEADER_BYTES + 4 * (BK_ECC_MAX_CURVE_POINT_WORDS))**
- maximum size in bytes for this configuration, of the header of a static key pair cryptogram*
- **#define BK_HYBRID_MAX_EPHEMERAL_CRYPTOGRAM_HEADER_SIZE_BYTES (BK_HYBRID_CRYPTOGRAM_HEADER_BYTES + 8 * (BK_ECC_MAX_CURVE_POINT_WORDS))**
- maximum size in bytes for this configuration, of the header of an ephemeral key pair cryptogram*
- **#define BK_ECC_CRYPTOGRAM_HEADER_SIZE_BYTES BK_HYBRID_CRYPTOGRAM_HEADER_BYTES**
- Size in bytes of the fixed part of the header of a cryptogram (legacy definition)*



Typedefs

- typedef `bk_key_source_id_t` `bk_ecc_key_source_t`
Key sources.
- typedef `uint8_t` `bk_key_purpose_t`
Key purpose.
- typedef `uint8_t` `bk_ecc_private_key_t`[66]
Elliptic curve private key.
- typedef `bk_key_purpose_t` `bk_ecc_key_purpose_t`
Elliptic curve key purpose.
- typedef `uint8_t` `bk_ecc_std_public_key_t`[(1+132)]
Elliptic curve public key.
- typedef `uint8_t` `bk_ecc_signature_t`[132]
Elliptic curve ECDSA signature.
- typedef `uint8_t` `bk_ecc_shared_secret_t`[66]
Elliptic curve ECDH shared secret.
- typedef `uint8_t` `bk_ecc_private_key_code_t`[(4*(12+17))]
Elliptic curve private key code.
- typedef `uint8_t` `bk_ecc_public_key_code_t`[(4*(12+33))]
Elliptic curve public key code.
- typedef `uint8_t` `bk_der_sig_value_t`[(9+132)]
DER-encoded ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax of [RFC 3279](#)
- typedef `uint8_t` `bk_der_signature_t`[(13+132)]
Elliptic curve ASN.1 DER encoded signature.
- typedef `uint8_t` `bk_der_public_key_info_t`[(26+(1+132))]
DER-encoded public key following the subject_public_key_info ASN.1 syntax of [RFC 5480](#)
- typedef `uint8_t` `bk_der_private_key_info_t`[(24+66+(1+132))]
DER-encoded elliptic-curve private key following the ECPrivateKey ASN.1 syntax of [RFC 5915](#)

Enumerations

- enum `bk_sym_key_type_t` {
 `BK_SYM_KEY_TYPE_128` = 0x00 ,
 `BK_SYM_KEY_TYPE_192` = 0x01 ,
 `BK_SYM_KEY_TYPE_256` = 0x02 }
Symmetric key type.
- enum `bk_key_source_id_t` {
 `BK_KEY_SOURCE_PUF_DERIVED` = 0x00 ,
 `BK_KEY_SOURCE_RNG_DERIVED` = 0x01 ,
 `BK_KEY_SOURCE_USER_PROVIDED` = 0x02 }
Key source identifier.
- enum `bk_ecc_curve_t` {
 `BK_ECC_CURVE_NIST_P192` = 0x00 ,
 `BK_ECC_CURVE_NIST_P224` = 0x01 ,
 `BK_ECC_CURVE_NIST_P256` = 0x02 ,
 `BK_ECC_CURVE_NIST_P384` = 0x03 ,
 `BK_ECC_CURVE_NIST_P521` = 0x04 }
Named elliptic curve.
- enum `bk_ecc_cryptogram_type_t` {
 `BK_ECC_CRYPTOGAM_TYPE_ECDH_STATIC` = 0x00 ,
 `BK_ECC_CRYPTOGAM_TYPE_ECDH_EPHEMERAL` = 0x01 }
Elliptic curve cryptogram type.

BK Core Functions

- `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`)



- Gets software product and version information.*
- `const char * bk_get_version_string` (void)
Gets specific software product and version string.
- `iid_return_t bk_init` (uint8_t *const sram_puf, const uint16_t sram_puf_size)
Initializes BK after power-up or reset.
- `iid_return_t bk_start` (const uint8_t *const activation_code)
Starts an existing cryptographic context for BK.
- `iid_return_t bk_stop` (void)
Stops the active cryptographic context of BK.

BK Randomness and Device-Key Generation Functions

- `iid_return_t bk_get_key` (const `bk_sym_key_type_t` key_type, const uint8_t index, uint8_t *const key)
(Re)generates a device-unique symmetric key
- `iid_return_t bk_generate_random` (const uint16_t number_of_bytes, uint8_t *const data_buffer)
Generates a sequence of random bytes.
- `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)
Generates an elliptic-curve private key.

BK Symmetric Key Crypto Functions

- `iid_return_t bk_wrap` (const uint8_t index, const uint8_t *const key, const uint16_t key_length, uint8_t *const key_code)
Securely wraps a presented key into a device-unique key code.
- `iid_return_t bk_unwrap` (const uint8_t *const key_code, uint8_t *const key, uint16_t *const key_length, uint8_t *const index)
Unwraps the key from a device-unique key code.

BK Elliptic Curve Crypto Functions

- `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 std_public_key)
Derives an elliptic curve public key from a private key.
- `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)
Protects an elliptic curve private key into a private key code, ready for use with BK's elliptic curve functions.
- `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)
Computes an elliptic curve public key code from a private key code, to be used with BK's elliptic curve functions.
- `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 std_public_key, `bk_ecc_public_key_code_t` *const public_key_code)
Imports an elliptic curve public key to the internal protected public key code format, ready for use with BK's elliptic curve functions.
- `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 std_public_key, `bk_ecc_curve_t` *const curve, `bk_ecc_key_purpose_t` *const purpose_flags)
Exports a binary elliptic curve public key from BK's internal protected public key code format.
- `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)
ECDSA-sign signs a message, using a BK protected private key code.



- `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)
Verifies an ECDSA-signed message, using a BK protected public key code.
- `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)
Computes an ECDH shared secret, from a pair of BK protected public and private key 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, `uint32_t` plaintext_length, `uint8_t` *const cryptogram, `uint32_t` *const cryptogram_length)
Generates a BK elliptic-curve cryptogram, providing message encryption and authentication.
- `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, `uint32_t` cryptogram_length, `uint8_t` *const plaintext, `uint32_t` *const plaintext_length)
Processes a received BK elliptic-curve cryptogram to retrieve the contained message.
- `iid_return_t bk_get_public_key_from_cryptogram` (bool use_point_compression, `bk_ecc_curve_t` curve, const `uint8_t` *const cryptogram, `uint32_t` cryptogram_length, `uint8_t` *const std_public_key)
Extracts the sender's public key embedded in a BK elliptic-curve cryptogram.

BK PKI Functions

- `iid_return_t bk_maxsizeof_csr` (const `bk_ecc_private_key_code_t` *const private_key_code, const bool use_point_compression, const `bk_certificate_subject_t` *const csr_subjects, `uint16_t` *const maxcsr_length)
Precomputes the (maximum) byte size of a certificate signing request (CSR).
- `iid_return_t bk_create_csr` (const `bk_ecc_private_key_code_t` *const private_key_code, const bool use_point_compression, const `bk_certificate_subject_t` *const csr_subjects, `uint8_t` *const csr, `uint16_t` *const csr_length)
Creates a certificate signing request (CSR) for an elliptic curve key pair.
- `iid_return_t bk_maxsizeof_selfsigned_certificate` (const `bk_ecc_private_key_code_t` *const private_key_code, const bool use_point_compression, const `uint8_t` *const serial, const `uint16_t` serial_length, const `bk_certificate_subject_t` *const ssc_subjects, `uint16_t` *const maxcertificate_length)
Precomputes the (maximum) byte size of a self-signed certificate (SSC).
- `iid_return_t bk_create_selfsigned_certificate` (const `bk_ecc_private_key_code_t` *const private_key_code, const bool use_point_compression, const `uint8_t` *const serial, const `uint16_t` serial_length, const char *const valid_start, const char *const valid_end, const `bk_certificate_subject_t` *const ssc_subjects, `uint8_t` *const certificate, `uint16_t` *const certificate_length)
Creates a self-signed certificate (SSC) for an elliptic curve key pair.
- `iid_return_t bk_write_ec_private_key` (const `bk_ecc_curve_t` curve, `uint8_t` *const private_key, `uint8_t` *const ECPriateKey, `uint16_t` *const ECPriateKey_length)
Creates a DER-encoded representation of an elliptic curve private key and its associated public key following the ECPriateKey ASN.1 syntax specified in [RFC 5915](#).
- `iid_return_t bk_write_subject_public_key_info` (const `bk_ecc_curve_t` curve, `uint8_t` *const std_public_key, `uint8_t` *const subject_public_key_info, `uint16_t` *const subject_public_key_info_length)
Creates a DER-encoded representation of an elliptic curve public key following the subject_public_key_info ASN.1 syntax specified in [RFC 5480](#).
- `iid_return_t bk_write_ecdsa_sig_value` (const `bk_ecc_curve_t` curve, const `uint8_t` *const signature, `uint8_t` *const ecdsa_sig_value, `uint16_t` *const ecdsa_sig_value_length)
Creates a DER-encoded representation of an ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax specified in [RFC 3279](#).
- `iid_return_t bk_read_ec_private_key` (const `uint8_t` *const ECPriateKey, `uint8_t` *const private_key, `uint16_t` *private_key_length, `bk_ecc_curve_t` *const curve)
Reads a DER-encoded representation of an elliptic curve private key and its associated public key following the ECPriateKey ASN.1 syntax specified in [RFC 5915](#).
- `iid_return_t bk_read_subject_public_key_info` (const `uint8_t` *const subject_public_key_info, `uint8_t` *const std_public_key, `uint16_t` *std_public_key_length, `bk_ecc_curve_t` *const curve)



Reads a DER-encoded representation of an elliptic curve public key following the subject_public_key_info ASN.1 syntax specified in [RFC 5480](#).

- `iid_return_t bk_read_ecdsa_sig_value` (const `bk_ecc_curve_t` curve, const `uint8_t *const ecdsa_sig_value`, `uint8_t *const signature`)

Reads a DER-encoded representation of an ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax specified in [RFC 3279](#).

BK DER/X.509 Encoding Definitions

- `#define BK_DER_LENGTH_OCTETS 8`
Maximum size in bytes of a length integer in a DER encoding.
- `#define BK_DER_MAX_LENGTH_FIELD (1 + BK_DER_LENGTH_OCTETS)`
Maximum size in bytes of the entire length field in a DER encoding.
- `#define BK_SSC_MAX_SERIAL_NUMBER_BYTES 20`
The maximum number of bytes in the (long integer) serial number of a self-signed certificate (SSC)
- `#define BK_SSC_TIME_DIGITS 14`
The maximum number of characters in the validity fields (notBefore, notAfter) generalised time value string of a self-signed certificate (SSC)
- `#define BK_SSN_UNDEFINED_VALIDITY "99991231235959Z"`
The GeneralizedTime value of "99991231235959Z" used in case of not well defined validity fields (notBefore, notAfter) of a self-signed certificate.
- `#define BK_DER_MAX_SUBJECT_STR 64`
The maximum number of characters of CSR or self-signed certificate (SSC) subject distinguished name fields. This size limit is not due to memory restrictions, but to guard against ill formatted strings.
- `#define BK_DER_MAX_OID_STR 32`
The maximum number of characters of CSR or self-signed certificate (SSC) OID strings. This size limit is not due to memory restrictions, but to guard against ill formatted strings.

BK ECC Key Source ID Definitions (legacy)

- `#define BK_ECC_KEY_SOURCE_PUF_DERIVED ((bk_ecc_key_source_t)0x00)`
Equivalent to `BK_KEY_SOURCE_PUF_DERIVED`.
- `#define BK_ECC_KEY_SOURCE_RANDOM ((bk_ecc_key_source_t)0x01)`
Equivalent to `BK_KEY_SOURCE_RNG_DERIVED`.
- `#define BK_ECC_KEY_SOURCE_USER_PROVIDED ((bk_ecc_key_source_t)0x02)`
Equivalent to `BK_KEY_SOURCE_USER_PROVIDED`.

BK Key Purpose Flag Definitions

- `#define BK_ECC_KEY_PURPOSE_ECDH ((bk_ecc_key_purpose_t)0x01)`
Elliptic curve key (code)s marked with this purpose flag can be used for elliptic curve key agreement (ECDH) and en/decryption functions (ECIES, cryptogram).
- `#define BK_ECC_KEY_PURPOSE_ECDSA ((bk_ecc_key_purpose_t)0x02)`
Elliptic curve key (code)s marked with this purpose flag can be used for elliptic curve signature (ECDSA) generation and verification functions (ECDSA-sign, ECDSA-verify, CSR, self-signed certificate).
- `#define BK_ECC_KEY_PURPOSE_ECDH_AND_ECDSA ((bk_ecc_key_purpose_t)(0x01 | 0x02))`
This is the combination of the ECDH and ECDSA flags. Elliptic curve key (code)s marked with this purpose flag can be used for elliptic curve key agreement (ECDH) en/decryption functions (ECIES, cryptogram) as well as signature (ECDSA) generation and verification functions (ECDSA-sign, ECDSA-verify, CSR, self-signed certificate)

2.1.1. Detailed Description

BK Top-Level API.



2.1.2. Data Structure Documentation

2.1.2.1. struct bk_certificate_subject_t Certificate (subject) distinguished name.

Structure containing the supported distinguished name fields (X.509) which can be defined for a certificate subject used in CSR and self-signed certificate (SSC) generation. The name fields are expected as C-style \0-terminated character strings. The maximim character length of each of these fields is limited to [BK_DER_MAX_SUBJECT_STR](#).

Data Fields

const char *	subject_c	String describing the subject's countryName field (OID 2.5.4.6)
const char *	subject_o	String describing the subject's organisationName field (OID 2.5.4.10)
const char *	subject_cn	String describing the subject's commonName field (OID 2.5.4.3)
const char *	subject_sn	String describing the subject's serialNumber field (OID 2.5.4.5)

2.1.3. Macro Definition Documentation

2.1.3.1. BK_SECURITY_SIZE_BITS #define BK_SECURITY_SIZE_BITS 256

Root security strength of this BK configuration.

2.1.3.2. BK_SRAM_PUF_SIZE_BYTES #define BK_SRAM_PUF_SIZE_BYTES 1024

Size in bytes of the SRAM PUF allocation.

Examples

[iid_bk_examples_standalone.c](#).

2.1.3.3. BK_AC_SIZE_BYTES #define BK_AC_SIZE_BYTES 968

Size in bytes of activation code buffer.

Examples

[iid_bk_examples_standalone.c](#).

©2023 Intrinsic ID B.V. – all rights reserved.
The information contained herein is proprietary to Intrinsic ID B.V. and is made available under an obligation of confidentiality.
Receipt of this document does not imply any license under any intellectual property rights of Intrinsic ID B.V.

**2.1.3.4. BK_CORE_UID_BYTES** `#define BK_CORE_UID_BYTES 32`

Size in bytes of the core unique identifier.

2.1.3.5. BK_SYM_MAX_KEY_BYTES `#define BK_SYM_MAX_KEY_BYTES 32`

Maximum size in bytes for this configuration, of a symmetric key used by BK.

2.1.3.6. BK_SYM_MAX_KEY_WORDS `#define BK_SYM_MAX_KEY_WORDS 8`

Maximum size in words for this configuration, of a symmetric key used by BK.

2.1.3.7. BK_SYM_MAX_KEY_CODE_BYTES `#define BK_SYM_MAX_KEY_CODE_BYTES (44 + BK_SYM_MAX_KEY_BYTES)`

Size in bytes of a symmetric key code capable of storing the maximum sized symmetric key for this configuration.

2.1.3.8. BK_USER_KEY_CODE_NONKEY_BYTES `#define BK_USER_KEY_CODE_NONKEY_BYTES (44)`

Size in bytes of the header of a key code used to wrap external keys.

2.1.3.9. BK_ECC_CURVE_SECP521R1_N_BYTES `#define BK_ECC_CURVE_SECP521R1_N_BYTES 66`

Size in bytes, of an SECP521R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.10. BK_ECC_CURVE_SECP521R1_N_WORDS `#define BK_ECC_CURVE_SECP521R1_N_WORDS 17`

Size in words, of an SECP521R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.11. BK_ECC_CURVE_SECP521R1_P_BYTES `#define BK_ECC_CURVE_SECP521R1_P_BYTES 66`

Size in bytes, of an SECP521R1 elliptic curve field element that represents a coordinate of a curve point.

**2.1.3.12. BK_ECC_CURVE_SECP521R1_P_WORDS** #define BK_ECC_CURVE_SECP521R1_P_WORDS 17

Size in words, of an SECP521R1 elliptic curve field element that represents a coordinate of a curve point.

2.1.3.13. BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_BYTES #define
BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_BYTES 66

Size in bytes, of an SECP521R1 elliptic curve private key.

2.1.3.14. BK_ECC_CURVE_SECP521R1_PUBLIC_KEY_BYTES #define
BK_ECC_CURVE_SECP521R1_PUBLIC_KEY_BYTES 133

Size in bytes, of an SECP521R1 elliptic curve public key.

2.1.3.15. BK_ECC_CURVE_SECP521R1_SIGNATURE_BYTES #define
BK_ECC_CURVE_SECP521R1_SIGNATURE_BYTES 132

Size in bytes, of an SECP521R1 elliptic curve signature.

2.1.3.16. BK_ECC_CURVE_SECP521R1_SHARED_SECRET_BYTES #define
BK_ECC_CURVE_SECP521R1_SHARED_SECRET_BYTES 66

Size in bytes, of an SECP521R1 elliptic curve shared secret.

2.1.3.17. BK_ECC_CURVE_SECP521R1_STATIC_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP521R1_STATIC_CRYPTOGAM_HEADER_BYTES 212

Size in bytes, of an SECP521R1 elliptic curve static cryptogram header.

2.1.3.18. BK_ECC_CURVE_SECP521R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP521R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES 344

Size in bytes, of an SECP521R1 elliptic curve ephemeral cryptogram header.



2.1.3.19. BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES 24

Size in bytes, of an SECP521R1 elliptic curve private key info ASN.1 header.

2.1.3.20. BK_ECC_CURVE_SECP521R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP521R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES 25

Size in bytes, of an SECP521R1 elliptic curve public key info ASN.1 header.

2.1.3.21. BK_ECC_CURVE_SECP521R1_SIG_VALUE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP521R1_SIG_VALUE_DER_HEADER_BYTES 9

Size in bytes, of an SECP521R1 elliptic curve sig value ASN.1 header.

2.1.3.22. BK_ECC_CURVE_SECP521R1_SIGNATURE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP521R1_SIGNATURE_DER_HEADER_BYTES 13

Size in bytes, of an SECP521R1 elliptic curve signature ASN.1 header.

2.1.3.23. BK_ECC_CURVE_SECP384R1_N_BYTES #define BK_ECC_CURVE_SECP384R1_N_BYTES 48

Size in bytes, of an SECP384R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.24. BK_ECC_CURVE_SECP384R1_N_WORDS #define BK_ECC_CURVE_SECP384R1_N_WORDS 12

Size in words, of an SECP384R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.25. BK_ECC_CURVE_SECP384R1_P_BYTES #define BK_ECC_CURVE_SECP384R1_P_BYTES 48

Size in bytes, of an SECP384R1 elliptic curve field element that represents a coordinate of a curve point.

**2.1.3.26. BK_ECC_CURVE_SECP384R1_P_WORDS** #define BK_ECC_CURVE_SECP384R1_P_WORDS 12

Size in words, of an SECP384R1 elliptic curve field element that represents a coordinate of a curve point.

2.1.3.27. BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_BYTES #define
BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_BYTES 48

Size in bytes, of an SECP384R1 elliptic curve private key.

2.1.3.28. BK_ECC_CURVE_SECP384R1_PUBLIC_KEY_BYTES #define
BK_ECC_CURVE_SECP384R1_PUBLIC_KEY_BYTES 97

Size in bytes, of an SECP384R1 elliptic curve public key.

2.1.3.29. BK_ECC_CURVE_SECP384R1_SIGNATURE_BYTES #define
BK_ECC_CURVE_SECP384R1_SIGNATURE_BYTES 96

Size in bytes, of an SECP384R1 elliptic curve signature.

2.1.3.30. BK_ECC_CURVE_SECP384R1_SHARED_SECRET_BYTES #define
BK_ECC_CURVE_SECP384R1_SHARED_SECRET_BYTES 48

Size in bytes, of an SECP384R1 elliptic curve shared secret.

2.1.3.31. BK_ECC_CURVE_SECP384R1_STATIC_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP384R1_STATIC_CRYPTOGAM_HEADER_BYTES 176

Size in bytes, of an SECP384R1 elliptic curve static cryptogram header.

2.1.3.32. BK_ECC_CURVE_SECP384R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP384R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES 272

Size in bytes, of an SECP384R1 elliptic curve ephemeral cryptogram header.



2.1.3.33. BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES 22

Size in bytes, of an SECP384R1 elliptic curve private key info ASN.1 header.

2.1.3.34. BK_ECC_CURVE_SECP384R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP384R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES 23

Size in bytes, of an SECP384R1 elliptic curve public key info ASN.1 header.

2.1.3.35. BK_ECC_CURVE_SECP384R1_SIG_VALUE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP384R1_SIG_VALUE_DER_HEADER_BYTES 8

Size in bytes, of an SECP384R1 elliptic curve sig value ASN.1 header.

2.1.3.36. BK_ECC_CURVE_SECP384R1_SIGNATURE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP384R1_SIGNATURE_DER_HEADER_BYTES 11

Size in bytes, of an SECP384R1 elliptic curve signature ASN.1 header.

2.1.3.37. BK_ECC_CURVE_SECP256R1_N_BYTES #define BK_ECC_CURVE_SECP256R1_N_BYTES 32

Size in bytes, of an SECP256R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.38. BK_ECC_CURVE_SECP256R1_N_WORDS #define BK_ECC_CURVE_SECP256R1_N_WORDS 8

Size in words, of an SECP256R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.39. BK_ECC_CURVE_SECP256R1_P_BYTES #define BK_ECC_CURVE_SECP256R1_P_BYTES 32

Size in bytes, of an SECP256R1 elliptic curve field element that represents a coordinate of a curve point.

**2.1.3.40. BK_ECC_CURVE_SECP256R1_P_WORDS** #define BK_ECC_CURVE_SECP256R1_P_WORDS 8

Size in words, of an SECP256R1 elliptic curve field element that represents a coordinate of a curve point.

2.1.3.41. BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_BYTES #define
BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_BYTES 32

Size in bytes, of an SECP256R1 elliptic curve private key.

Examples

[iid_bk_examples_standalone.c](#), and [iid_bk_examples_wolfssl.c](#).

2.1.3.42. BK_ECC_CURVE_SECP256R1_PUBLIC_KEY_BYTES #define
BK_ECC_CURVE_SECP256R1_PUBLIC_KEY_BYTES 65

Size in bytes, of an SECP256R1 elliptic curve public key.

2.1.3.43. BK_ECC_CURVE_SECP256R1_SIGNATURE_BYTES #define
BK_ECC_CURVE_SECP256R1_SIGNATURE_BYTES 64

Size in bytes, of an SECP256R1 elliptic curve signature.

2.1.3.44. BK_ECC_CURVE_SECP256R1_SHARED_SECRET_BYTES #define
BK_ECC_CURVE_SECP256R1_SHARED_SECRET_BYTES 32

Size in bytes, of an SECP256R1 elliptic curve shared secret.

2.1.3.45. BK_ECC_CURVE_SECP256R1_STATIC_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP256R1_STATIC_CRYPTOGAM_HEADER_BYTES 144

Size in bytes, of an SECP256R1 elliptic curve static cryptogram header.



2.1.3.46. BK_ECC_CURVE_SECP256R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP256R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES 208

Size in bytes, of an SECP256R1 elliptic curve ephemeral cryptogram header.

2.1.3.47. BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES 24

Size in bytes, of an SECP256R1 elliptic curve private key info ASN.1 header.

2.1.3.48. BK_ECC_CURVE_SECP256R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP256R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES 26

Size in bytes, of an SECP256R1 elliptic curve public key info ASN.1 header.

2.1.3.49. BK_ECC_CURVE_SECP256R1_SIG_VALUE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP256R1_SIG_VALUE_DER_HEADER_BYTES 8

Size in bytes, of an SECP256R1 elliptic curve sig value ASN.1 header.

2.1.3.50. BK_ECC_CURVE_SECP256R1_SIGNATURE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP256R1_SIGNATURE_DER_HEADER_BYTES 11

Size in bytes, of an SECP256R1 elliptic curve signature ASN.1 header.

2.1.3.51. BK_ECC_CURVE_SECP224R1_N_BYTES #define BK_ECC_CURVE_SECP224R1_N_BYTES 28

Size in bytes, of an SECP224R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.52. BK_ECC_CURVE_SECP224R1_N_WORDS #define BK_ECC_CURVE_SECP224R1_N_WORDS 7

Size in words, of an SECP224R1 elliptic curve scalar, smaller than the order of the curve.

**2.1.3.53. BK_ECC_CURVE_SECP224R1_P_BYTES** #define BK_ECC_CURVE_SECP224R1_P_BYTES 28

Size in bytes, of an SECP224R1 elliptic curve field element that represents a coordinate of a curve point.

2.1.3.54. BK_ECC_CURVE_SECP224R1_P_WORDS #define BK_ECC_CURVE_SECP224R1_P_WORDS 7

Size in words, of an SECP224R1 elliptic curve field element that represents a coordinate of a curve point.

2.1.3.55. BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_BYTES #define
BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_BYTES 28

Size in bytes, of an SECP224R1 elliptic curve private key.

2.1.3.56. BK_ECC_CURVE_SECP224R1_PUBLIC_KEY_BYTES #define
BK_ECC_CURVE_SECP224R1_PUBLIC_KEY_BYTES 57

Size in bytes, of an SECP224R1 elliptic curve public key.

2.1.3.57. BK_ECC_CURVE_SECP224R1_SIGNATURE_BYTES #define
BK_ECC_CURVE_SECP224R1_SIGNATURE_BYTES 56

Size in bytes, of an SECP224R1 elliptic curve signature.

2.1.3.58. BK_ECC_CURVE_SECP224R1_SHARED_SECRET_BYTES #define
BK_ECC_CURVE_SECP224R1_SHARED_SECRET_BYTES 28

Size in bytes, of an SECP224R1 elliptic curve shared secret.

2.1.3.59. BK_ECC_CURVE_SECP224R1_STATIC_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP224R1_STATIC_CRYPTOGAM_HEADER_BYTES 136

Size in bytes, of an SECP224R1 elliptic curve static cryptogram header.



2.1.3.60. BK_ECC_CURVE_SECP224R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP224R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES 192

Size in bytes, of an SECP224R1 elliptic curve ephemeral cryptogram header.

2.1.3.61. BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES 21

Size in bytes, of an SECP224R1 elliptic curve private key info ASN.1 header.

2.1.3.62. BK_ECC_CURVE_SECP224R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP224R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES 23

Size in bytes, of an SECP224R1 elliptic curve public key info ASN.1 header.

2.1.3.63. BK_ECC_CURVE_SECP224R1_SIG_VALUE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP224R1_SIG_VALUE_DER_HEADER_BYTES 8

Size in bytes, of an SECP224R1 elliptic curve sig value ASN.1 header.

2.1.3.64. BK_ECC_CURVE_SECP224R1_SIGNATURE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP224R1_SIGNATURE_DER_HEADER_BYTES 11

Size in bytes, of an SECP224R1 elliptic curve signature ASN.1 header.

2.1.3.65. BK_ECC_CURVE_SECP192R1_N_BYTES #define BK_ECC_CURVE_SECP192R1_N_BYTES 24

Size in bytes, of an SECP192R1 elliptic curve scalar, smaller than the order of the curve.

2.1.3.66. BK_ECC_CURVE_SECP192R1_N_WORDS #define BK_ECC_CURVE_SECP192R1_N_WORDS 6

Size in words, of an SECP192R1 elliptic curve scalar, smaller than the order of the curve.

**2.1.3.67. BK_ECC_CURVE_SECP192R1_P_BYTES** `#define BK_ECC_CURVE_SECP192R1_P_BYTES 24`

Size in bytes, of an SECP192R1 elliptic curve field element that represents a coordinate of a curve point.

2.1.3.68. BK_ECC_CURVE_SECP192R1_P_WORDS `#define BK_ECC_CURVE_SECP192R1_P_WORDS 6`

Size in words, of an SECP192R1 elliptic curve field element that represents a coordinate of a curve point.

2.1.3.69. BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_BYTES `#define BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_BYTES 24`

Size in bytes, of an SECP192R1 elliptic curve private key.

2.1.3.70. BK_ECC_CURVE_SECP192R1_PUBLIC_KEY_BYTES `#define BK_ECC_CURVE_SECP192R1_PUBLIC_KEY_BYTES 49`

Size in bytes, of an SECP192R1 elliptic curve public key.

2.1.3.71. BK_ECC_CURVE_SECP192R1_SIGNATURE_BYTES `#define BK_ECC_CURVE_SECP192R1_SIGNATURE_BYTES 48`

Size in bytes, of an SECP192R1 elliptic curve signature.

2.1.3.72. BK_ECC_CURVE_SECP192R1_SHARED_SECRET_BYTES `#define BK_ECC_CURVE_SECP192R1_SHARED_SECRET_BYTES 24`

Size in bytes, of an SECP192R1 elliptic curve shared secret.

2.1.3.73. BK_ECC_CURVE_SECP192R1_STATIC_CRYPTOGAM_HEADER_BYTES `#define BK_ECC_CURVE_SECP192R1_STATIC_CRYPTOGAM_HEADER_BYTES 128`

Size in bytes, of an SECP192R1 elliptic curve static cryptogram header.



2.1.3.74. BK_ECC_CURVE_SECP192R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES #define
BK_ECC_CURVE_SECP192R1_EPHEMERAL_CRYPTOGAM_HEADER_BYTES 176

Size in bytes, of an SECP192R1 elliptic curve ephemeral cryptogram header.

2.1.3.75. BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_INFO_DER_HEADER_BYTES 24

Size in bytes, of an SECP192R1 elliptic curve private key info ASN.1 header.

2.1.3.76. BK_ECC_CURVE_SECP192R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP192R1_PUBLIC_KEY_INFO_DER_HEADER_BYTES 26

Size in bytes, of an SECP192R1 elliptic curve public key info ASN.1 header.

2.1.3.77. BK_ECC_CURVE_SECP192R1_SIG_VALUE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP192R1_SIG_VALUE_DER_HEADER_BYTES 8

Size in bytes, of an SECP192R1 elliptic curve sig value ASN.1 header.

2.1.3.78. BK_ECC_CURVE_SECP192R1_SIGNATURE_DER_HEADER_BYTES #define
BK_ECC_CURVE_SECP192R1_SIGNATURE_DER_HEADER_BYTES 11

Size in bytes, of an SECP192R1 elliptic curve signature ASN.1 header.

2.1.3.79. BK_ECC_MAX_CURVE_SIZE_N_BYTES #define BK_ECC_MAX_CURVE_SIZE_N_BYTES 66

Maximum size in bytes for this configuration, of an elliptic curve scalar, smaller than the order of the curve.

2.1.3.80. BK_ECC_MAX_CURVE_SIZE_N_WORDS #define BK_ECC_MAX_CURVE_SIZE_N_WORDS 17

Maximum size in words for this configuration, of an elliptic curve scalar, smaller than the order of the curve.

**2.1.3.81. BK_ECC_MAX_CURVE_SIZE_P_BYTES** `#define BK_ECC_MAX_CURVE_SIZE_P_BYTES 66`

Maximum size in bytes for this configuration, of an elliptic curve field element that represents a coordinate of a curve point.

2.1.3.82. BK_ECC_MAX_CURVE_SIZE_P_WORDS `#define BK_ECC_MAX_CURVE_SIZE_P_WORDS 17`

Maximum size in words for this configuration, of an elliptic curve field element that represents a coordinate of a curve point.

2.1.3.83. BK_ECC_MAX_CURVE_POINT_BYTES `#define BK_ECC_MAX_CURVE_POINT_BYTES 132`

Maximum size in bytes for this configuration, of an elliptic curve field element that represents a curve point.

2.1.3.84. BK_ECC_MAX_CURVE_POINT_WORDS `#define BK_ECC_MAX_CURVE_POINT_WORDS 33`

Maximum size in words for this configuration, of an elliptic curve field element that represents a curve point.

2.1.3.85. BK_ECC_MAX_SIG_VALUE_DER_HEADER_BYTES `#define BK_ECC_MAX_SIG_VALUE_DER_HEADER_BYTES 9`

Maximum size in bytes for this configuration, of an elliptic curve sig value ASN.1 header.

2.1.3.86. BK_ECC_MAX_SIGNATURE_DER_HEADER_BYTES `#define BK_ECC_MAX_SIGNATURE_DER_HEADER_BYTES 13`

Maximum size in bytes for this configuration, of an elliptic curve signature ASN.1 header.

2.1.3.87. BK_ECC_MAX_PUBLIC_KEY_INFO_DER_HEADER_BYTES `#define BK_ECC_MAX_PUBLIC_KEY_INFO_DER_HEADER_BYTES 26`

Maximum size in bytes for this configuration, of an elliptic curve public key info ASN.1 header.



2.1.3.88. BK_ECC_MAX_PRIVATE_KEY_INFO_DER_HEADER_BYTES #define
BK_ECC_MAX_PRIVATE_KEY_INFO_DER_HEADER_BYTES 24

Maximum size in bytes for this configuration, of an elliptic curve private key info ASN.1 header.

2.1.3.89. BK_ECC_MAX_CURVE_SIZE_BYTES #define
BK_ECC_MAX_CURVE_SIZE_BYTES BK_ECC_MAX_CURVE_SIZE_N_BYTES

Maximum size in bytes for this configuration, of an elliptic curve scalar, smaller than the order of the curve.

2.1.3.90. BK_ECC_MAX_CURVE_SIZE_WORDS #define
BK_ECC_MAX_CURVE_SIZE_WORDS BK_ECC_MAX_CURVE_SIZE_N_WORDS

Maximum size in bytes for this configuration, of an elliptic curve field element that represents a coordinate of a curve point.

2.1.3.91. BK_ECC_PRIVATE_KEY_BYTES #define BK_ECC_PRIVATE_KEY_BYTES BK_ECC_MAX_CURVE_SIZE_N_BYTES

Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve private key.

2.1.3.92. BK_ECC_PUBLIC_KEY_BYTES #define BK_ECC_PUBLIC_KEY_BYTES BK_ECC_MAX_CURVE_POINT_BYTES

Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve public key.

2.1.3.93. BK_ECC_STD_PUBLIC_KEY_BYTES #define BK_ECC_STD_PUBLIC_KEY_BYTES (1 +
BK_ECC_MAX_CURVE_POINT_BYTES)

Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve x9.62 public key.

2.1.3.94. BK_ECC_SIGNATURE_BYTES #define BK_ECC_SIGNATURE_BYTES BK_ECC_MAX_CURVE_POINT_BYTES

Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDSA signature.

**2.1.3.95. BK_ECC_SHARED_SECRET_BYTES** #define

BK_ECC_SHARED_SECRET_BYTES BK_ECC_MAX_CURVE_SIZE_P_BYTES

Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDH shared secret.

2.1.3.96. BK_ECC_PRIVATE_KEY_WORDS #define

BK_ECC_PRIVATE_KEY_WORDS BK_ECC_MAX_CURVE_SIZE_N_WORDS

Size in words for this configuration, of the store capable of holding the maximum sized elliptic curve private key.

2.1.3.97. BK_ECC_PUBLIC_KEY_WORDS #define BK_ECC_PUBLIC_KEY_WORDS BK_ECC_MAX_CURVE_POINT_WORDS

Size in words for this configuration, of the store capable of holding the maximum sized elliptic curve public key.

2.1.3.98. BK_ECC_SIGNATURE_WORDS #define BK_ECC_SIGNATURE_WORDS BK_ECC_MAX_CURVE_POINT_WORDS

Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDSA signature.

2.1.3.99. BK_ECC_SHARED_SECRET_WORDS #define

BK_ECC_SHARED_SECRET_WORDS BK_ECC_MAX_CURVE_SIZE_P_WORDS

Size in bytes for this configuration, of the store capable of holding the maximum sized elliptic curve ECDH shared secret.

2.1.3.100. BK_ECC_DER_SIG_VALUE_BYTES #define

BK_ECC_DER_SIG_VALUE_BYTES (BK_ECC_MAX_SIG_VALUE_DER_HEADER_BYTES + BK_ECC_SIGNATURE_BYTES)

Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded sig value.

2.1.3.101. BK_ECC_DER_SIGNATURE_BYTES #define

BK_ECC_DER_SIGNATURE_BYTES (BK_ECC_MAX_SIGNATURE_DER_HEADER_BYTES + BK_ECC_SIGNATURE_BYTES)

Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded signature.

**2.1.3.102. BK_ECC_DER_PUBLIC_KEY_INFO_BYTES** #define

```
BK_ECC_DER_PUBLIC_KEY_INFO_BYTES (BK_ECC_MAX_PUBLIC_KEY_INFO_DER_HEADER_BYTES +  
BK_ECC_STD_PUBLIC_KEY_BYTES)
```

Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded public key info.

2.1.3.103. BK_ECC_DER_PRIVATE_KEY_INFO_BYTES #define

```
BK_ECC_DER_PRIVATE_KEY_INFO_BYTES (BK_ECC_MAX_PRIVATE_KEY_INFO_DER_HEADER_BYTES +  
BK_ECC_PRIVATE_KEY_BYTES + BK_ECC_STD_PUBLIC_KEY_BYTES)
```

Size in bytes for this configuration, of the byte array capable of holding the maximum sized elliptic curve ECDSA ASN.1 DER encoded public key info.

2.1.3.104. BK_ECC_KEY_CODE_HEADER_BYTES #define BK_ECC_KEY_CODE_HEADER_BYTES 48

Size in bytes of the header of an elliptic curve key code.

2.1.3.105. BK_ECC_KEY_CODE_HEADER_WORDS #define BK_ECC_KEY_CODE_HEADER_WORDS 12

Size in words of the header of an elliptic curve key code.

```
2.1.3.106. BK_ECC_PRIVATE_KEY_CODE_SIZE_BYTES #define BK_ECC_PRIVATE_KEY_CODE_SIZE_BYTES (4 *  
(BK_ECC_KEY_CODE_HEADER_WORDS + BK_ECC_MAX_CURVE_SIZE_N_WORDS))
```

Size in bytes for this configuration, of an elliptic curve private key code capable of storing the maximum sized private key.

2.1.3.107. BK_ECC_PUBLIC_KEY_PACK_BYTES #define

```
BK_ECC_PUBLIC_KEY_PACK_BYTES (BK_ECC_MAX_CURVE_POINT_BYTES)
```

Size in bytes for this configuration, of the representation capable of holding the maximum sized elliptic curve public key.

```
2.1.3.108. BK_ECC_PUBLIC_KEY_CODE_SIZE_BYTES #define BK_ECC_PUBLIC_KEY_CODE_SIZE_BYTES (4 *  
(BK_ECC_KEY_CODE_HEADER_WORDS + BK_ECC_MAX_CURVE_POINT_WORDS))
```

Size in bytes for this configuration, of an elliptic curve public key code capable of storing the maximum sized public key.

**2.1.3.109. BK_HYBRID_CRYPTOGRAM_HEADER_BYTES** `#define BK_HYBRID_CRYPTOGRAM_HEADER_BYTES 80`

Size in bytes of the fixed part of the header of a cryptogram.

2.1.3.110. BK_HYBRID_MAX_STATIC_CRYPTOGRAM_HEADER_SIZE_BYTES `#define BK_HYBRID_MAX_STATIC_CRYPTOGRAM_HEADER_SIZE_BYTES (BK_HYBRID_CRYPTOGRAM_HEADER_BYTES + 4 * (BK_ECC_MAX_CURVE_POINT_WORDS))`

maximum size in bytes for this configuration, of the header of a static key pair cryptogram

2.1.3.111. BK_HYBRID_MAX_EPHEMERAL_CRYPTOGRAM_HEADER_SIZE_BYTES `#define BK_HYBRID_MAX_EPHEMERAL_CRYPTOGRAM_HEADER_SIZE_BYTES (BK_HYBRID_CRYPTOGRAM_HEADER_BYTES + 8 * (BK_ECC_MAX_CURVE_POINT_WORDS))`

maximum size in bytes for this configuration, of the header of an ephemeral key pair cryptogram

2.1.3.112. BK_ECC_CRYPTOGRAM_HEADER_SIZE_BYTES `#define BK_ECC_CRYPTOGRAM_HEADER_SIZE_BYTES BK_HYBRID_CRYPTOGRAM_HEADER_BYTES`

Size in bytes of the fixed part of the header of a cryptogram (legacy definition)

2.1.3.113. BK_DER_LENGTH_OCTETS `#define BK_DER_LENGTH_OCTETS 8`

Maximum size in bytes of a length integer in a DER encoding.

2.1.3.114. BK_DER_MAX_LENGTH_FIELD `#define BK_DER_MAX_LENGTH_FIELD (1 + BK_DER_LENGTH_OCTETS)`

Maximum size in bytes of the entire length field in a DER encoding.

2.1.3.115. BK_SSC_MAX_SERIAL_NUMBER_BYTES `#define BK_SSC_MAX_SERIAL_NUMBER_BYTES 20`

The maximum number of bytes in the (long integer) serial number of a self-signed certificate (SSC)

**2.1.3.116. BK_SSC_TIME_DIGITS** `#define BK_SSC_TIME_DIGITS 14`

The maximum number of characters in the validity fields (notBefore, notAfter) generalised time value string of a self-signed certificate (SSC)

2.1.3.117. BK_SSN_UNDEFINED_VALIDITY `#define BK_SSN_UNDEFINED_VALIDITY "99991231235959Z"`

The GeneralizedTime value of "99991231235959Z" used in case of not well defined validity fields (notBefore, notAfter) of a self-signed certificate.

2.1.3.118. BK_DER_MAX_SUBJECT_STR `#define BK_DER_MAX_SUBJECT_STR 64`

The maximum number of characters of CSR or self-signed certificate (SSC) subject distinguished name fields. This size limit is not due to memory restrictions, but to guard against ill formatted strings.

2.1.3.119. BK_DER_MAX_OID_STR `#define BK_DER_MAX_OID_STR 32`

The maximum number of characters of CSR or self-signed certificate (SSC) OID strings. This size limit is not due to memory restrictions, but to guard against ill formatted strings.

2.1.3.120. BK_ECC_KEY_SOURCE_PUF_DERIVED `#define
BK_ECC_KEY_SOURCE_PUF_DERIVED ((bk_ecc_key_source_t)0x00)`

Equivalent to [BK_KEY_SOURCE_PUF_DERIVED](#).

2.1.3.121. BK_ECC_KEY_SOURCE_RANDOM `#define
BK_ECC_KEY_SOURCE_RANDOM ((bk_ecc_key_source_t)0x01)`

Equivalent to [BK_KEY_SOURCE_RNG_DERIVED](#).

2.1.3.122. BK_ECC_KEY_SOURCE_USER_PROVIDED `#define
BK_ECC_KEY_SOURCE_USER_PROVIDED ((bk_ecc_key_source_t)0x02)`

Equivalent to [BK_KEY_SOURCE_USER_PROVIDED](#).



2.1.3.123. BK_ECC_KEY_PURPOSE_ECDH `#define BK_ECC_KEY_PURPOSE_ECDH ((bk_ecc_key_purpose_t)0x01)`

Elliptic curve key (code)s marked with this purpose flag can be used for elliptic curve key agreement (ECDH) and en/decryption functions (ECIES, cryptogram).

2.1.3.124. BK_ECC_KEY_PURPOSE_ECDSA `#define BK_ECC_KEY_PURPOSE_ECDSA ((bk_ecc_key_purpose_t)0x02)`

Elliptic curve key (code)s marked with this purpose flag can be used for elliptic curve signature (ECDSA) generation and verification functions (ECDSA-sign, ECDSA-verify, CSR, self-signed certificate).

2.1.3.125. BK_ECC_KEY_PURPOSE_ECDH_AND_ECDSA `#define BK_ECC_KEY_PURPOSE_ECDH_AND_ECDSA ((bk_ecc_key_purpose_t)(0x01 | 0x02))`

This is the combination of the ECDH and ECDSA flags. Elliptic curve key (code)s marked with this purpose flag can be used for elliptic curve key agreement (ECDH) en/decryption functions (ECIES, cryptogram) as well as signature (ECDSA) generation and verification functions (ECDSA-sign, ECDSA-verify, CSR, self-signed certificate).

2.1.4. Typedef Documentation

2.1.4.1. `bk_ecc_key_source_t` `typedef bk_key_source_id_t bk_ecc_key_source_t`

Key sources.

Defines the allowed sources from which an elliptic curve private key can be generated.

2.1.4.2. `bk_key_purpose_t` `bk_key_purpose_t`

Key purpose.

Generic key purpose type which is further specialized as a `bk_ecc_key_purpose_t`.

2.1.4.3. `bk_ecc_private_key_t` `bk_ecc_private_key_t`

Elliptic curve private key.

Defines the private key type capable of storing the the maximum sized private key for this configuration.

Examples

[iid_bk_examples_wolfssl.c](#).



2.1.4.4. **bk_ecc_key_purpose_t** `bk_ecc_key_purpose_t`

Elliptic curve key purpose.

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 public- or private key code.

2.1.4.5. **bk_ecc_std_public_key_t** `bk_ecc_std_public_key_t`

Elliptic curve public key.

Defines the public key type capable of storing the the maximum sized public key in X9.62 binary format for this configuration.

2.1.4.6. **bk_ecc_signature_t** `bk_ecc_signature_t`

Elliptic curve ECDSA signature.

Defines the ECDSA signature type capable of storing the the maximum sized signature for this configuration.

2.1.4.7. **bk_ecc_shared_secret_t** `bk_ecc_shared_secret_t`

Elliptic curve ECDH shared secret.

Defines the ECDH shared secret type capable of storing the the maximum sized shared secret for this configuration.

2.1.4.8. **bk_ecc_private_key_code_t** `bk_ecc_private_key_code_t`

Elliptic curve private key code.

Defines the private key code type capable of storing the the maximum sized private key for this configuration.

2.1.4.9. **bk_ecc_public_key_code_t** `bk_ecc_public_key_code_t`

Elliptic curve public key code.

Defines the public key code type capable of storing the the maximum sized public key for this configuration.

2.1.4.10. **bk_der_sig_value_t** `bk_der_sig_value_t`

DER-encoded ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax of [RFC 3279](#)

Defines the byte array type capable of storing the the maximum sized DER-encoded ECDSA-Sig-Value for this configuration.



2.1.4.11. `bk_der_signature_t` `bk_der_signature_t`

Elliptic curve ASN.1 DER encoded signature.

Defines the byte array type capable of storing the the maximum sized ECDSA ASN.1 DER encoded signature for this configuration.

2.1.4.12. `bk_der_public_key_info_t` `bk_der_public_key_info_t`

DER-encoded public key following the `subject_public_key_info` ASN.1 syntax of [RFC 5480](#)

Defines the byte array type capable of storing the the maximum sized DER-encoded `subject_public_key_info` for this configuration.

2.1.4.13. `bk_der_private_key_info_t` `bk_der_private_key_info_t`

DER-encoded elliptic-curve private key following the `ECPrivateKey` ASN.1 syntax of [RFC 5915](#)

Defines the byte array type capable of storing the the maximum sized DER-encoded elliptic-curve private key for this configuration.

2.1.5. Enumeration Type Documentation

2.1.5.1. `bk_sym_key_type_t` `enum bk_sym_key_type_t`

Symmetric key type.

Enumerates the symmetric key types which can be generated by the `bk_get_key()` function. This key type also implies the size in bytes of the referred key.

Enumerator

<code>BK_SYM_KEY_TYPE_128</code>	128-bit Symmetric Key A 128-bit key for symmetric cryptography algorithms: <ul style="list-style-type: none">size in bytes of the key value: 16size in bytes of the <code>bk_wrap()</code>-generated key code: (16 + <code>BK_USER_KEY_CODE_NONKEY_BYTES</code>)maximal security strength for operations with this key: 128-bit
<code>BK_SYM_KEY_TYPE_192</code>	192-bit Symmetric Key A 192-bit key for symmetric cryptography algorithms: <ul style="list-style-type: none">size in bytes of the key value: 24size in bytes of the <code>bk_wrap()</code>-generated key code: (24 + <code>BK_USER_KEY_CODE_NONKEY_BYTES</code>)maximal security strength for operations with this key: 192-bit
<code>BK_SYM_KEY_TYPE_256</code>	256-bit Symmetric Key A 256-bit key for symmetric cryptography algorithms: <ul style="list-style-type: none">size in bytes of the key value: 32size in bytes of the <code>bk_wrap()</code>-generated key code: (32 + <code>BK_USER_KEY_CODE_NONKEY_BYTES</code>)maximal security strength for operations with this key: 256-bit



2.1.5.2. `bk_key_source_id_t` enum `bk_key_source_id_t`

Key source identifier.

Enumerates the allowed sources from which a generic, symmetric or elliptic curve private key can be generated for use with cryptographic functions.

Enumerator

<code>BK_KEY_SOURCE_PUF_DERIVED</code>	Key derived from PUF. The key is derived in a direct line from the SRAM PUF's device unique start-up data. Keys derived in this way can always be exactly rederived by calling cryptographic functions with the same arguments in the same cryptographic context, and on the same device.
<code>BK_KEY_SOURCE_RNG_DERIVED</code>	Randomly generated key. The key is randomly generated using BK's internal cryptographically secure random number generator which is seeded by entropy coming from the device noise. Keys generated in this way cannot be rederived by cryptographic functions.
<code>BK_KEY_SOURCE_USER_PROVIDED</code>	Key provided by user. The key is an external value which is provided by the calling application to the <code>bk_create_private_key()</code> function.

2.1.5.3. `bk_ecc_curve_t` enum `bk_ecc_curve_t`

Named elliptic curve.

Enumerates the named elliptic curves which can be recognized by BK's elliptic curve functions. The used curve also implies the size in bytes of several input and output parameters of these functions.

Enumerator

<code>BK_ECC_CURVE_NIST_P192</code>	Curve NIST-P192. The elliptic-curve cryptosystem specified by the NIST-P192/secp192r1 domain parameters: <ul style="list-style-type: none">• BK curve ID: 0x00• size in bytes of private keys: 24• size in bytes of public keys: 49• size in bytes of ECDSA signatures: 48• size in bytes of ECDH shared secrets: 24• size in bytes of ECIES header: 49• maximal security strength for operations over this curve: 96-bit
<code>BK_ECC_CURVE_NIST_P224</code>	Curve NIST-P224. The elliptic-curve cryptosystem specified by the NIST-P224/secp224r1 domain parameters: <ul style="list-style-type: none">• BK curve ID: 0x01• size in bytes of private keys: 28• size in bytes of public keys: 57• size in bytes of ECDSA signatures: 56• size in bytes of ECDH shared secrets: 28• size in bytes of ECIES header: 57• maximal security strength for operations over this curve: 112-bit



Enumerator

BK_ECC_CURVE_NIST_P256	Curve NIST-P256. The elliptic-curve cryptosystem specified by the NIST-P256/secp256r1 domain parameters: <ul style="list-style-type: none">• BK curve ID: 0x02• size in bytes of private keys: 32• size in bytes of public keys: 65• size in bytes of ECDSA signatures: 64• size in bytes of ECDH shared secrets: 32• size in bytes of ECIES header: 65• maximal security strength for operations over this curve: 128-bit
BK_ECC_CURVE_NIST_P384	Curve NIST-P384. The elliptic-curve cryptosystem specified by the NIST-P384/secp384r1 domain parameters: <ul style="list-style-type: none">• BK curve ID: 0x03• size in bytes of private keys: 48• size in bytes of public keys: 97• size in bytes of ECDSA signatures: 96• size in bytes of ECDH shared secrets: 48• size in bytes of ECIES header: 97• maximal security strength for operations over this curve: 192-bit
BK_ECC_CURVE_NIST_P521	Curve NIST-P521. The elliptic-curve cryptosystem specified by the NIST-P521/secp521r1 domain parameters: <ul style="list-style-type: none">• BK curve ID: 0x04• size in bytes of private keys: 66• size in bytes of public keys: 133• size in bytes of ECDSA signatures: 132• size in bytes of ECDH shared secrets: 66• size in bytes of ECIES header: 133• maximal security strength for operations over this curve: 260-bit

2.1.5.4. `bk_ecc_cryptogram_type_t` `enum bk_ecc_cryptogram_type_t`

Elliptic curve cryptogram type.

Enumerates the defined cryptogram types used in `bk_generate_cryptogram()` and returned by `bk_process_cryptogram()`.



Enumerator

BK_ECC_CRYPTOGRAM_TYPE_ECDH_STATIC	Cryptogram type using static key pairs on both sides. Cryptogram type using static elliptic curve key pairs on both sending and receiving side: <ul style="list-style-type: none">• offers message confidentiality, message integrity, source authentication, replay protection• does not offer forward secrecy, non-repudiation• size in bytes of cryptogram = size in bytes of plaintext + BK_ECC_CRYPTOGRAM_HEADER_SIZE_BYTES + size in bytes of public key according to curve - 1
BK_ECC_CRYPTOGRAM_TYPE_ECDH_EPHEMERAL	Cryptogram type using an ephemeral sender key pair. Cryptogram type using an ephemeral key pair on the sending side, and a static key pair on the receiving side: <ul style="list-style-type: none">• offers message confidentiality, message integrity, source authentication, replay protection, and forward secrecy against compromise of the sender's private key• does not offer non-repudiation• size in bytes of cryptogram = size in bytes of plaintext + BK_ECC_CRYPTOGRAM_HEADER_SIZE_BYTES + (2* size in bytes of public key according to curve) - 2

2.1.6. Function Documentation

```
2.1.6.1. bk_get_product_info() 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 )
```

Gets software product and version information.

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

Precondition

A call to this function is allowed in all states of BK.

Postcondition

A call to this function never changes the operational state of BK.

©2023 Intrinsic ID B.V. – all rights reserved.
The information contained herein is proprietary to Intrinsic ID B.V. and is made available under an obligation of confidentiality.
Receipt of this document does not imply any license under any intellectual property rights of Intrinsic ID B.V.



Parameters

out	<i>product_id</i>	Pointer to an output buffer with a size of 1 byte which will hold the product identifier character
out	<i>major_version</i>	Pointer to an output buffer with a size of 1 byte which will hold the major software version
out	<i>minor_version</i>	Pointer to an output buffer with a size of 1 byte which will hold the minor software version
out	<i>patch</i>	Pointer to an output buffer with a size of 1 byte which will hold the software patch number
out	<i>build_number</i>	Pointer to an output buffer with a size of 1 byte which will hold the build number

Returns

IID_SUCCESS if successful, or **IID_INVALID_PARAMETERS** otherwise

2.1.6.2. bk_get_version_string() `const char * bk_get_version_string (`
`void)`

Gets specific software product and version string.

This function can be used to get the exact name, flavor, commit and version.

Precondition

A call to this function is allowed in all states of BK.

Postcondition

A call to this function never changes the operational state of BK.

Returns

A version string

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

Initializes BK after power-up or reset.

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

Precondition

A call to this function is only allowed when BK is in the *Uninitialized* state. Called from any other operational state, this function will return immediately with return code **IID_NOT_ALLOWED**, without taking any action.

Postcondition

A successful call to this function changes the operational state of BK to the *Initialized* state.



Parameters

<code>in, out</code>	<code>sram_puf</code>	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). Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
<code>in</code>	<code>sram_puf_size</code>	The size in bytes of available SRAM PUF that can be used by the software. Note: the size of the SRAM must be (at least) <code>BK_SRAM_PUF_SIZE_BYTES</code> bytes

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_ERROR_STARTUP_DATA` or `IID_NOT_ALLOWED` otherwise

Examples

`iid_bk_examples_mbedtls.c`, `iid_bk_examples_standalone.c`, and `iid_bk_examples_wolfssl.c`.

2.1.6.4. `bk_start()` `iid_return_t bk_start (`
`const uint8_t *const activation_code)`

Starts an existing cryptographic context for BK.

This function is used to re-instantiate a cryptographic context of BK 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, BK's cryptographic functionality becomes available

Precondition

A call to this function is only allowed when BK is in the *Initialized* or *Stopped* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A successful call to this function (return code equals `IID_SUCCESS`) changes the operational state of BK to the *Started* state.

Parameters

<code>in</code>	<code>activation_code</code>	Pointer to an input buffer of byte size <code>BK_AC_SIZE_BYTES</code> which holds the activation code. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
-----------------	------------------------------	---



Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_INVALID_AC` or `IID_NOT_ALLOWED` otherwise

Examples

`iid_bk_examples_mbedtls.c`, `iid_bk_examples_standalone.c`, and `iid_bk_examples_wolfssl.c`.

2.1.6.5. `bk_stop()` `iid_return_t bk_stop (`
`void)`

Stops the active cryptographic context of BK.

This function will unstantiate a cryptographic context of BK which was earlier instantiated by `bk_enroll()` or `bk_start()`. Once a cryptographic context is unstantiated, BK's cryptographic functionality becomes unavailable. 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.

Precondition

A call to this function is only allowed when BK is in the *Initialized* state, the *Enrolled* state, the *Started* state, or the *Stopped* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Note

- Calling this function from the *Stopped* state is allowed but will have no effect (BK remains in the *Stopped* state).
- Calling this function from the *Initialized* state is allowed, and will have as effect that `bk_enroll()` becomes unavailable until the next device repower or reset.
- Calling this function from the *Uninitialized* state is not allowed.

Postcondition

A successful call to this function (return code equals `IID_SUCCESS`) changes the operational state of BK to the *Stopped* state.

Returns

`IID_SUCCESS` if successful, or `IID_NOT_ALLOWED` otherwise

Examples

`iid_bk_examples_standalone.c`.



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

(Re)generates a device-unique symmetric 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()`. 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.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<code>key_type</code>	The type of the device key that will be generated. This must be a value of the enumeration type <code>bk_sym_key_type_t</code> .
in	<code>index</code>	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.
out	<code>key</code>	Pointer to an output buffer which will hold the generated device key. Note: the size of the key buffer must be large enough to hold a key type as specified by the <code>key_type</code> input parameter, as indicated here .

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS` or `IID_NOT_ALLOWED` otherwise

Examples

`iid_bk_examples_mbedtls.c`, `iid_bk_examples_standalone.c`, and `iid_bk_examples_wolfssl.c`.

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

Generates a sequence of random bytes.

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.



Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>number_of_bytes</i>	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 <i>data_buffer</i> needs to be at least equal to this value.
out	<i>data_buffer</i>	Pointer to an output buffer of byte size <i>number_of_bytes</i> which will hold the requested random bytes.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS` or `IID_NOT_ALLOWED` otherwise

Examples

`iid_bk_examples_mbedtls.c`, `iid_bk_examples_standalone.c`, and `iid_bk_examples_wolfssl.c`.

2.1.6.8. `bk_get_private_key()` `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)`

Generates an elliptic-curve 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()`. The length of the generated private key depends on the specified `curve`. 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.



Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>curve</i>	Specifies the named elliptic curve on which the considered private key is defined. It must be a valid curve type of the <code>bk_ecc_curve_t</code> enumeration.
in	<i>usage_context</i>	Pointer to an input buffer of byte size <code>usage_context_length</code> which holds the (optional) usage context. When used, the entropy of this buffer is included in the private key derivation for private keys derived <ul style="list-style-type: none">from the device fingerprint (<code>key_source = BK_ECC_KEY_SOURCE_PUF_DERIVED</code>)from the device's random number generator (<code>key_source = BK_ECC_KEY_SOURCE_RANDOM</code>) Note: providing a usage context is optional. If the specified <code>usage_context_length</code> is 0, usage context is taken into account and <code>usage_context</code> must be NULL.
in	<i>usage_context_length</i>	The size in bytes of the <code>usage_context</code> buffer. If this size is set to 0, no usage context is taken into account and <code>usage_context</code> must be NULL.
in	<i>key_source</i>	Specifies the source of the elliptic curve private key. It must be a valid source of the <code>bk_key_source_id_t</code> enumeration. The allowed private key sources, and their meaning, are explained here . The allowed key sources for <code>bk_get_private_key()</code> are: <ul style="list-style-type: none"><code>BK_ECC_KEY_SOURCE_PUF_DERIVED</code>: the private key is derived from the device fingerprint and (optionally) the provided usage context.
out	<i>private_key</i>	Pointer to an output buffer which will hold the generated elliptic curve private key. The private key value is provided in raw binary format, in network byte order representation. Note: the size of the private key buffer must be large enough to hold a private key as specified by the <code>curve</code> parameter. For the given configuration, the buffer type <code>bk_ecc_private_key_t</code> can hold all raw private keys.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS` or `IID_NOT_ALLOWED` otherwise

Examples

[iid_bk_examples_mbedtls.c](#), [iid_bk_examples_standalone.c](#), and [iid_bk_examples_wolfssl.c](#).



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

Securely wraps a presented key into a device-unique key code.

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_USER_KEY_CODE_NONKEY_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.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code **IID_NOT_ALLOWED**, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>index</i>	An integer value in the range [0:255] indicating the index of the key-to-be-wrapped. For bk_wrap() and bk_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 BK can use it, e.g. to specify context-information associated to the key.
in	<i>key</i>	Pointer to an input buffer with a size of <i>key_length</i> bytes, which holds the plain key-to-be-wrapped. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
in	<i>key_length</i>	The length, in bytes, of the key-to-be-wrapped. This must be an integer value in the range [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.
out	<i>key_code</i>	Pointer to an output buffer of size (<i>key_length</i> + BK_USER_KEY_CODE_NONKEY_BYTES) bytes which will hold the generated key code containing the wrapped key. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.

Returns

IID_SUCCESS if succesful, or **IID_INVALID_PARAMETERS** or **IID_NOT_ALLOWED** otherwise

```
2.1.6.10. bk_unwrap() iid_return_t bk_unwrap (  
    const uint8_t *const key_code,
```



```
uint8_t *const key,  
uint16_t *const key_length,  
uint8_t *const index )
```

Unwraps the key from a device-unique key code.

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.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>key_code</i>	Pointer to an input buffer which holds the key-code-to-be-unwrapped. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
out	<i>key</i>	Pointer to an output buffer with a size of (sizeof(<i>key_code</i>) - BK_USER_KEY_CODE_NONKEY_BYTES) bytes that will hold the unwrapped key.
out	<i>key_length</i>	Pointer to an output buffer with a size of 2 bytes (representing a <code>uint16_t</code>) which will hold the size in bytes, of the unwrapped key.
out	<i>index</i>	Pointer to an output buffer with a size of 1 byte which will hold the index value which was wrapped alongside the unwrapped key. For bk_wrap() and bk_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 BK can use it, e.g. to specify context-information associated to the key.

Returns

[IID_SUCCESS](#) if succesful, or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#) or [IID_INVALID_KEY_CODE](#) otherwise

2.1.6.11. [bk_derive_public_key\(\)](#) `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 std_public_key)`



Derives an elliptic curve public key from a private key.

This function derives the elliptic curve public key corresponding to a private key created with [bk_get_private_key\(\)](#), and outputs the public key.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>use_point_compression</i>	Note: this flag is present for future use compatibility, but is not used for this product version of BK. For this product version, this value has to be set to False (no point compression). Any other values will result in the return code IID_INVALID_PARAMETERS .
in	<i>curve</i>	Specifies the named elliptic curve on which the private key is defined. It must be a valid curve type of the bk_ecc_curve_t enumeration.
in	<i>private_key</i>	Pointer to an input buffer that holds the elliptic curve private key. The expected input is binary in network byte order representation. Its size in bytes is determined by the used curve . For the given configuration, the buffer type bk_ecc_private_key_t can hold all raw private keys. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
out	<i>std_public_key</i>	Pointer to an output buffer that will hold the elliptic curve public key computed from the private_key input. The output is in X9.62 binary format. Its size in bytes is determined by the used curve . For the given configuration, the buffer type bk_ecc_std_public_key_t can hold all X9.62 binary format public keys. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.

Returns

[IID_SUCCESS](#) or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#) or [IID_INVALID_PRIVATE_KEY](#)

```
2.1.6.12. bk_create_private_key() 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 )
```



Protects an elliptic curve private key into a private key code, ready for use with BK's elliptic curve functions.

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:

- private keys derived from the device's secret fingerprint
- randomly generated private keys
- 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 protection mechanisms for transforming private keys into private key codes are similar as for the [bk_wrap\(\)](#) function, 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 BK.

The generation mechanisms for creating device-unique and random private keys are similar as for [bk_get_private_key\(\)](#), 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.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>curve</i>	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.
in	<i>purpose_flags</i>	Flag which specifies the purpose flags of the private key.
in	<i>usage_context</i>	Pointer to an input buffer of size <code>usage_context_length</code> bytes which holds the (optional) usage context. When used, the entropy of this buffer is included in the private key derivation for private keys derived: <ul style="list-style-type: none">• from the device fingerprint (<code>key_source = BK_ECC_KEY_SOURCE_PUF_DERIVED</code>)• from the device's random number generator (<code>key_source = BK_ECC_KEY_SOURCE_RANDOM</code>) Note: providing a usage context is optional. If the specified <code>usage_context_length</code> is 0, no usage context is taken into account and <code>usage_context</code> must be NULL.
in	<i>usage_context_length</i>	The size in bytes of the <code>usage_context</code> buffer. If this length is set to 0, no usage context is taken into account and <code>usage_context</code> must be NULL.



Parameters

in	<i>key_source</i>	<p>Specifies the source of the elliptic curve private key. It must be a valid source of the bk_ecc_key_source_t enumeration. The allowed key sources for bk_create_private_key() are:</p> <ul style="list-style-type: none">• BK_ECC_KEY_SOURCE_PUF_DERIVED: the private key is derived from the device fingerprint and (optionally) the provided usage context.• BK_ECC_KEY_SOURCE_RANDOM: the private key is uniformly randomly generated from BK's internal random number generator and (optionally) the provided usage context• BK_ECC_KEY_SOURCE_USER_PROVIDED: the private key is provided externally. When this key source is selected, <code>usage_context</code> is not used and <code>private_key</code> is used directly with only a check that it is a well-formed private key for the specified curve. 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. <p>Note: for</p> <ul style="list-style-type: none">• BK_ECC_KEY_SOURCE_PUF_DERIVED• BK_ECC_KEY_SOURCE_RANDOM the <code>usage_context</code> entropy, if present, is added to the key derivation process to provide a secure fallback or key diversification.
in	<i>private_key</i>	<p>Pointer to an input buffer that holds the private key used when <code>key_source</code> is user-provided. The expected input is binary in network byte order representation. Its size in bytes is determined by the used curve. For the given configuration, the buffer type bk_ecc_private_key_t can hold all raw private keys.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p> <p>Note: for key sources other than BK_ECC_KEY_SOURCE_USER_PROVIDED, this input is not used.</p>
out	<i>private_key_code</i>	<p>Pointer to an output buffer for a bk_ecc_private_key_code_t which will hold the created elliptic curve private key code.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p>

Returns

[IID_SUCCESS](#) if successful, or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#) or [IID_INVALID_PRIVATE_KEY](#) otherwise

2.1.6.13. [bk_compute_public_from_private_key\(\)](#) [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*)

Computes an elliptic curve public key code from a private key code, to be used with BK's elliptic curve functions.

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, 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()` can be used to retrieve the public key value contained in a public key code.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<code>private_key_code</code>	Pointer to an input buffer for a <code>bk_ecc_private_key_code_t</code> which contains an elliptic curve private key code created by <code>bk_create_private_key()</code> . Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
out	<code>public_key_code</code>	Pointer to an output buffer for a <code>bk_ecc_public_key_code_t</code> which will hold the created elliptic curve public key code. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_INVALID_PRIVATE_KEY_CODE` otherwise

```
2.1.6.14. bk_import_public_key() 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 std_public_key,  
    bk_ecc_public_key_code_t *const public_key_code )
```

Imports an elliptic curve public key to the internal protected public key code format, ready for use with BK's elliptic curve functions.

This function imports an elliptic curve public key from a provided X9.62 binary format (compressed or uncompressed) to a corresponding public key code format. 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, 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()` can be used to retrieve the public key value contained in a public key code.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>curve</i>	Specifies the named elliptic curve on which the public key is defined. It must be a valid curve type of the <code>bk_ecc_curve_t</code> enumeration.
in	<i>purpose_flags</i>	Flag which specifies the <code>purpose_flags</code> of the public key.
in	<i>std_public_key</i>	Pointer to an input buffer which holds the elliptic curve public key to be imported. The expected input is in X9.62 binary format. Its size in bytes is determined by the used [curve] (<code>bk_ecc_curve_t</code>). For the given configuration, the buffer type <code>bk_ecc_std_public_key_t</code> can hold all X9.62 binary format public keys.
out	<i>public_key_code</i>	Pointer to an output buffer for a <code>bk_ecc_public_key_code_t</code> which will hold the created elliptic curve public key code. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.

Returns

`IID_SUCCESS` or `IID_INVALID_PARAMETERS` or `IID_NOT_ALLOWED` or `IID_INVALID_PUBLIC_KEY`

```
2.1.6.15. bk_export_public_key() 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 std_public_key,  
    bk_ecc_curve_t *const curve,  
    bk_ecc_key_purpose_t *const purpose_flags )
```

Exports a binary elliptic curve public key from BK's internal protected public key code format.

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



Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>use_point_compression</i>	This flag is present for future use compatibility, but is not used for this product version of BK. For this product version, this value has to be set to False (no point compression). Any other values will result in the return code <code>IID_INVALID_PARAMETERS</code> .
in	<i>public_key_code</i>	Pointer to an input buffer for a <code>bk_ecc_public_key_code_t</code> which holds the elliptic curve public key code to be exported. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
out	<i>std_public_key</i>	Pointer to an output buffer which will hold the exported elliptic curve public key. The output is in X9.62 binary format. Its size in bytes is determined by the used [curve] (<code>bk_ecc_curve_t</code>). For the given configuration, the buffer type <code>bk_ecc_std_public_key_t</code> can hold all X9.62 binary format public keys.
out	<i>curve</i>	Pointer to an output buffer for a <code>bk_ecc_curve_t</code> which will hold the curve on which the exported public key is defined.
out	<i>purpose_flags</i>	Pointer to an output buffer for a <code>bk_ecc_key_purpose_t</code> which will hold the <code>purpose flags</code> of the exported public key.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_INVALID_PUBLIC_KEY_CODE` otherwise

```
2.1.6.16. bk_ecdsa_sign() 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 )
```

ECDSA-sign signs a message, using a BK protected private key code.

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.



Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>private_key_code</i>	Pointer to an input buffer for a <code>bk_ecc_private_key_code_t</code> which holds the elliptic curve private key code to be used for signing. Note: a private key code used for signing shall have been created with <code>bk_create_private_key()</code> , with [purpose flags] (<code>bk_defgroup_keypurpose</code>) allowing its use for ECDSA operations. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
in	<i>deterministic_signature</i>	Flag which specifies if either deterministic or non-deterministic signing will be used. If this value equals <code>False</code> , message will be signed using the standard ECDSA non-deterministic algorithm. Otherwise, the message will be signed using a deterministic variant of ECDSA .
in	<i>message</i>	Pointer to an input buffer of size <code>message_length</code> bytes which holds the message or the message hash that will be signed. If <code>message_length</code> is zero, <code>message</code> must be <code>NULL</code> .
in	<i>message_length</i>	Value which specifies the size in bytes of the <code>message</code> buffer. If <code>message_is_hash</code> equals <code>True</code> (i.e. the provided message is actually a message hash), this size must be equal to the size in bytes of the used private key, as determined by the used curve . Otherwise (i.e. the provided message is a raw message buffer), the size in bytes must be equal to the raw message size. In this case, <code>message_length</code> could also be zero in which case an empty message will be signed. If <code>message_length</code> is zero, <code>message</code> must be <code>NULL</code> .
in	<i>message_is_hash</i>	Flag which specifies if the provided <code>message</code> buffer contains an already hashed message, or a raw message buffer. If this value equals <code>False</code> , <code>bk_ecdsa_sign()</code> will treat the <code>message</code> buffer as a raw message, and will hash it first using SHA-256 and the trailing bytes will be truncated to equal the length of the used elliptic curve private key before signing the resulting hash. Otherwise, <code>bk_ecdsa_sign()</code> will treat the message buffer as an already hashed message, and it will be signed directly. For this hashed message, SHA-256, SHA-384 and SHA-512 are accepted.
out	<i>signature</i>	Pointer to an output buffer which will hold the computed ECDSA signature. The signature is formatted as the direct concatenation of ECDSA's (r, s) values as two fixed-length unsigned integers, and the signature size in bytes depends on the used curve . For the given configuration, the buffer type <code>bk_ecc_signature_t</code> can hold all signatures. The used curve is set during the creation of the private key code with <code>bk_create_private_key()</code> .
in, out	<i>signature_length</i>	Pointer to an input/output buffer for a <code>uint16</code> which as an input holds the allocated size in bytes of the <code>signature</code> buffer, and as an output will hold the exact size of the returned signature value. If the size provided by the input is smaller than the required size as determined by the used curve , the function returns with <code>IID_INVALID_PARAMETERS</code> but still sets the output value at this pointer to the required buffer size.



Returns

[IID_SUCCESS](#) if successful, or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#), [IID_INVALID_PRIVATE_KEY_CODE](#) or [IID_ECC_NOT_ALLOWED](#) otherwise

2.1.6.17. `bk_ecdsa_verify()` `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)`

Verifies an ECDSA-signed message, using a BK protected public key code.

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.

Note

This function has no explicit output parameter(s), but the result of this function will be contained in its return code. Upon return of [IID_SUCCESS](#), the presented `signature` was valid for the `message`, while upon return of [IID_INVALID_SIGNATURE](#), the presented `signature` was not valid for the `message`.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>public_key_code</i>	Pointer to an input buffer for a bk_ecc_public_key_code_t which holds the elliptic curve public key code to be used for verification. Note: a public key code used for signature verification shall have been computed with bk_compute_public_from_private_key() or imported with bk_import_public_key() , with purpose flags allowing its use for ECDSA operations. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
in	<i>message</i>	Pointer to an input buffer of size <code>message_length</code> bytes which holds the message or the message hash for which the signature will be verified. If <code>message_length</code> is zero, <code>message</code> must be NULL.



Parameters

in	<i>message_length</i>	Value which specifies the size in bytes of the <code>message</code> buffer. If <code>message_is_hash</code> equals True (i.e. the provided message is actually a message hash), this length must be equal to the size in bytes of the used private key, as determined by the used <code>[curve]</code> (<code>bk_ecc_curve_t</code>). Otherwise (i.e. the provided message is a raw message buffer), the size in bytes must be equal to the raw message length. In this case, <code>message_length</code> could also be zero in which case an empty message will be verified. If <code>message_length</code> is zero, <code>message</code> must be NULL.
in	<i>message_is_hash</i>	Flag which specifies if the provided <code>message</code> buffer contains an already hashed message, or a raw message buffer. If this value equals False, <code>bk_ecdsa_verify()</code> will treat the <code>message</code> buffer as a raw message, and will hash it first using SHA-256 and the trailing bytes will be truncated to equal the length of the used elliptic curve private key before verifying the resulting hash. Otherwise, <code>bk_ecdsa_verify()</code> will treat the message buffer as an already hashed message, and it will be verified directly. For this hashed message, SHA-256, SHA-384 and SHA-512 are accepted.
in	<i>signature</i>	Pointer to an input buffer of size <code>signature_length</code> bytes, which holds the ECDSA signature to be verified. The expected format of the provided signature is the same format as generated by <code>bk_ecdsa_sign()</code> .
in	<i>signature_length</i>	The size in bytes of the signature to be verified, which depends on the <code>[curve]</code> (<code>bk_ecc_curve_t</code>) used for the signing operation.

Returns

`IID_SUCCESS` or `IID_INVALID_SIGNATURE` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_INVALID_PUBLIC_KEY_CODE` or `IID_ECC_NOT_ALLOWED` otherwise

2.1.6.18. `bk_ecdh_shared_secret()` `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`)

Computes an ECDH shared secret, from a pair of BK protected public and private key codes.

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.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.



Parameters

in	<i>private_key_code</i>	Pointer to an input buffer for a <code>bk_ecc_private_key_code_t</code> which holds the elliptic curve private key code to be used for the Diffie-Hellmann operation. Note: a private key code used for shared secret computation shall have been created with <code>bk_create_private_key()</code> , with <code>purpose flags</code> allowing its use for ECDH operations. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
in	<i>public_key_code</i>	Pointer to an input buffer for a <code>bk_ecc_public_key_code_t</code> which holds the elliptic curve public key code to be used for the Diffie-Hellmann operation. Note: a public key code used for shared secret computation shall have been computed with <code>bk_compute_public_from_private_key()</code> or imported with <code>bk_import_public_key()</code> , with <code>purpose flags</code> allowing its use for ECDH operations. 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. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
out	<i>shared_secret</i>	Pointer to an output buffer which will hold the computed shared secret. The size in bytes of the shared secret depends on the used <code>curve</code> . For the given configuration, the buffer type <code>bk_ecc_shared_secret_t</code> can hold all shared secrets. The shared secret will be equal to the X-coordinate of the commonly derived point on the elliptic curve.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_INVALID_PRIVATE_KEY_CODE`, `IID_INVALID_PUBLIC_KEY_CODE`, `IID_ECC_NOT_ALLOWED` or `IID_CURVE_MISMATCH` otherwise

2.1.6.19. `bk_generate_cryptogram()` `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`,
 uint32_t `plaintext_length`,
 uint8_t *const `cryptogram`,
 uint32_t *const `cryptogram_length`)

Generates a BK elliptic-curve cryptogram, providing message encryption and authentication.

This function packs a provided plaintext into a BK-specific protected cryptogram format, based on an elliptic curve hybrid encryption scheme. The cryptogram format offers protection for confidentiality, integrity, sender authentication and replay. 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 ECDH/encryption.



Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>receiver_public_key_code</i>	<p>Pointer to an input buffer for a <code>bk_ecc_public_key_code_t</code> which holds the elliptic curve public key code of the receiver to whom the generated cryptogram will be sent.</p> <p>Note: a public key code used for cryptogram generation shall have been computed with <code>bk_compute_public_from_private_key()</code> or imported with <code>bk_import_public_key()</code>, with <code>purpose flags</code> allowing its use for ECDH/encryption operations.</p> <p>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 <code>receiver_public_key_code</code> shall have been verified in an independent manner (e.g. through certificate validation), or it shall come from an independent trusted source.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p>
in	<i>sender_private_key_code</i>	<p>Pointer to an input buffer for a <code>bk_ecc_private_key_code_t</code> which holds the elliptic curve private key code of the sender whom will send the generated cryptogram.</p> <p>Note: a private key code used for cryptogram generation shall have been created with <code>bk_create_private_key()</code>, with <code>purpose flags</code> allowing its use for ECDH/encryption operations.</p> <p>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.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p>
in	<i>cryptogram_type</i>	<p>Value which specifies the cryptogram type to be generated. It must be a valid value of the <code>bk_ecc_cryptogram_type_t</code> enumeration.</p>
in, out	<i>counter64</i>	<p>Pointer to an input/output buffer with a size of 8 bytes (interpreted as a uint64), which as an input holds the current 64-bit monotonic counter used for cryptogram replay protection, and as an output will hold the new counter value after successful function completion.</p> <p>Note: a separate counter buffer shall be used for each distinct sender-receiver key pair. The calling application needs to retrieve this buffer from persistent storage before each call to <code>bk_generate_cryptogram()</code>, and store back its updated value in persistent storage after the function completes successfully (return code is <code>IID_SUCCESS</code>). Upon first use of a counter buffer for a sender-receiver pair, the counter buffer needs to be initialized to all-zero bytes, after the initial validation of the receiver's public key.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p>



Parameters

in	<i>plaintext</i>	Pointer to an input buffer of size <code>plaintext_length</code> bytes which holds the plaintext which will be packed in the cryptogram. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0. Note: the length of this buffer must be a multiple of 4 bytes.
in	<i>plaintext_length</i>	Value which specifies the size in bytes of <code>plaintext</code> . Its value must be positive (> 0) and a multiple of 4.
out	<i>cryptogram</i>	Pointer to an output buffer of size <code>cryptogram_length</code> bytes which will hold the generated cryptogram. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
in, out	<i>cryptogram_length</i>	Pointer to an input/output buffer for a uint32, which as an input holds the allocated size in bytes of the <code>cryptogram</code> output buffer, and as an output will hold the exact size in bytes of the returned cryptogram. If the size provided by the input is smaller then the required size as determined by the used [cryptogram type] (<code>bk_ecc_cryptogram_type_t</code>) and <code>curve</code> , the function returns with <code>IID_INVALID_PARAMETERS</code> but still sets the output value at this pointer to the required buffer size.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_INVALID_COUNTER`, `IID_INVALID_PRIVATE_KEY_CODE`, `IID_INVALID_PUBLIC_KEY_CODE`, `IID_ECC_NOT_ALLOWED` or `IID_CURVE_MISMATCH` otherwise

```
2.1.6.20. bk_process_cryptogram() 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,  
    uint32_t cryptogram_length,  
    uint8_t *const plaintext,  
    uint32_t *const plaintext_length )
```

Processes a received BK elliptic-curve cryptogram to retrieve the contained message.

This function processes a received cryptogram in a BK-specific 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. 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 ECDH/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()`.



Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>receiver_private_key_code</i>	Pointer to an input buffer for a <code>bk_ecc_private_key_code_t</code> which holds the elliptic curve private key code of the receiver by whom the cryptogram is processed. Note: a private key code used for cryptogram processing shall have been created with <code>bk_create_private_key()</code> , with [purpose flags] (<code>bk_defgroup_keypurpose</code>) allowing its use for ECDH/encryption operations. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
in	<i>sender_public_key_code</i>	Pointer to an input buffer for a <code>bk_ecc_public_key_code_t</code> which holds the elliptic curve public key code of the sender from whom the to-be-processed cryptogram was received. Note: a public key code used for cryptogram processing shall have been created with <code>bk_compute_public_from_private_key()</code> or <code>bk_import_public_key()</code> , with [purpose flags] allowing its use for ECDH/encryption operations. 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 <code>sender_public_key_code</code> shall have been verified in an independent manner (e.g. through certificate validation), or it shall come from an independent trusted source. 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. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
out	<i>cryptogram_type</i>	Pointer to an output buffer for a <code>bk_ecc_cryptogram_type_t</code> which will hold the cryptogram type which was used to generate the cryptogram.



Parameters

in, out	<i>counter64</i>	<p>Pointer to an input/output buffer with a size of 8 bytes (interpreted as a uint64), which as an input holds the current 64-bit monotonic counter used for cryptogram replay protection, and as an output will hold the new counter value after successful function completion.</p> <p>Note: a separate counter buffer shall be used for each distinct sender-receiver key pair. The calling application needs to retrieve this buffer from persistent storage before each call to <code>bk_process_cryptogram()</code>, and store back its updated value in persistent storage after the function completes successfully (return code is <code>IID_SUCCESS</code>). Upon first use of a counter buffer for a sender-receiver pair, the counter buffer needs to be initialized to all-zero bytes, after the initial validation of the sender's public key.</p> <p>Note: <code>bk_process_cryptogram()</code> 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 as input by <code>counter64</code>. This prevents replay of old cryptograms, but also obstructs the ability to process multiple consecutive cryptograms out of order.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p>
in	<i>cryptogram</i>	<p>Pointer to an input buffer of size <code>cryptogram_length</code> bytes which holds the full cryptogram to be processed.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p>
in	<i>cryptogram_length</i>	<p>Value which specifies the size in bytes of the presented <code>cryptogram</code>.</p> <p>Note: this must be the exact size of the cryptogram to be processed, as specified by the used <code>cryptogram type</code> and <code>curve</code>.</p>
out	<i>plaintext</i>	<p>Pointer to an output buffer of size <code>plaintext_length</code> bytes, which will hold the decrypted plaintext.</p> <p>Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.</p>
in, out	<i>plaintext_length</i>	<p>Pointer to an input/output buffer for a uint32, which as an input holds the allocated size in bytes of the <code>plaintext</code> output buffer, and as an output will hold the exact size in bytes of the returned plaintext. If the size provided by the input is smaller then the required size as determined by the used [cryptogram type] (<code>bk_ecc_cryptogram_type_t</code>) and <code>curve</code>, the function returns with <code>IID_INVALID_PARAMETERS</code> but still sets the output value at this pointer to the required buffer size.</p>

Returns

`IID_SUCCESS`, `IID_INVALID_CRYPTOGAM` or `IID_INVALID_SENDER` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_INVALID_COUNTER`, `IID_INVALID_PRIVATE_KEY_CODE`, `IID_INVALID_PUBLIC_KEY_CODE`, `IID_ECC_NOT_ALLOWED` or `IID_CURVE_MISMATCH` otherwise

2.1.6.21. `bk_get_public_key_from_cryptogram()` `iid_return_t` `bk_get_public_key_from_cryptogram` (
 `bool` `use_point_compression`,
 `bk_ecc_curve_t` `curve`,



```
const uint8_t *const cryptogram,  
uint32_t cryptogram_length,  
uint8_t *const std_public_key )
```

Extracts the sender's public key embedded in a BK elliptic-curve cryptogram.

This *helper* function extracts the sender's public key from a received BK elliptic-curve 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 the 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.

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.

Warning

If this function is used to extract a sender's public key, it is 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.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK.

Parameters

in	<i>use_point_compression</i>	This flag is present for future use compatibility, but is not used for this product version of BK. For this product version, this value has to be set to False (no point compression). Any other values will result in the return code IID_INVALID_PARAMETERS .
in	<i>curve</i>	Specifies the named elliptic curve on which the cryptogram is defined. It must be a valid curve type of the bk_ecc_curve_t enumeration.
in	<i>cryptogram</i>	Pointer to an input buffer of size <i>cryptogram_length</i> bytes, which holds the full cryptogram. Note: the address must be at a 32-bit boundary, so the lowest two bits of the address must be 0.
in	<i>cryptogram_length</i>	Value which specifies the size in bytes of the presented <i>cryptogram</i> . Note: this must be the exact size of the cryptogram to be processed, as specified by the used cryptogram type and curve .
out	<i>std_public_key</i>	Pointer to an output buffer which will hold the extracted public key in X9.62 binary format. The size in bytes is determined by the used curve . For the given configuration, the buffer type bk_ecc_std_public_key_t can hold all X9.62 binary format public keys.



Returns

`IID_SUCCESS` or `IID_INVALID_CRYPTOGRAM` if successful, or `IID_INVALID_PARAMETERS` or `IID_NOT_ALLOWED` otherwise

2.1.6.22. `bk_maxsizeof_csr()` `iid_return_t` `bk_maxsizeof_csr` (
 const `bk_ecc_private_key_code_t` *const `private_key_code`,
 const bool `use_point_compression`,
 const `bk_certificate_subject_t` *const `csr_subjects`,
 uint16_t *const `maxcsr_length`)

Precomputes the (maximum) byte size of a certificate signing request (CSR).

This function computes the maximum possible size a DER-encoded binary `PKCS#10`-formatted certificate signing request (CSR) can have for a presented elliptic curve private key (code) and a given set of subject information fields identifying the owner of that key.

Note

Due to a combination of the way DER-encoding works, and the non-determinism involved in the ECDSA-signature on the CSR, the precise size in bytes of a CSR is also non-deterministic and can only be exactly determined once the CSR is generated with `bk_create_csr()`. However, based on a given set of inputs, it is possible to precompute the maximum possible byte size of the CSR, which is what `bk_maxsizeof_csr()` does. This is the size which needs to be allocated in memory for the `csr` output buffer of `bk_create_csr()`. The exact byte size of a generated CSR is returned by `bk_create_csr()` as `csr_length`.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by `IID_SUCCESS`, this function will have written the maximum size in bytes. On error the output buffer content is undefined

Parameters

in	<code>private_key_code</code>	Pointer to an input buffer for a <code>bk_ecc_private_key_code_t</code> which holds the elliptic curve private key for which the CSR size needs to be computed. Note: a private key code used for CSR creation shall have been created with <code>bk_create_private_key()</code> , with <code>purpose flags</code> allowing its use for ECDSA operations.
in	<code>use_point_compression</code>	This flag is present for future use compatibility, but is not used for this product version of BK. For this product version, this value has to be set to False (no point compression). Any other values will result in the return code <code>IID_INVALID_PARAMETERS</code> .
in	<code>csr_subjects</code>	Pointer to an input buffer for a <code>bk_certificate_subject_t</code> structure. The elements of this structure contain the values which will be included in the distinguished name of the CSR's Subject field.
out	<code>maxcsr_length</code>	Pointer to a uint16 which will hold the maximal possible size in bytes of a CSR generated on the provided inputs with <code>bk_create_csr()</code> .



Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED` or `IID_INVALID_PRIVATE_KEY_CODE` otherwise

2.1.6.23. `bk_create_csr()` `iid_return_t` `bk_create_csr` (
 const `bk_ecc_private_key_code_t` *const `private_key_code`,
 const bool `use_point_compression`,
 const `bk_certificate_subject_t` *const `csr_subjects`,
 uint8_t *const `csr`,
 uint16_t *const `csr_length`)

Creates a certificate signing request (CSR) for an elliptic curve key pair.

This function creates a `PKCS#10`-formatted certificate signing request (CSR) for a presented elliptic curve private key (code) and a limited set of subject information fields identifying the owner of that key. The CSR is returned as a DER-encoded binary string.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by `IID_SUCCESS`, this function will have written the CSR as a DER-encoded byte sequence to the output buffer and have returned its size in bytes. On error the output buffer content is undefined

Parameters

in	<code>private_key_code</code>	Pointer to an input buffer for a <code>bk_ecc_private_key_code_t</code> which holds the elliptic curve private key for which a CSR will be created. The public key corresponding to this private key will be included in the CSR, while the private key will be used to (self-) sign the CSR. Note: a private key code used for CSR generation shall have been created with <code>bk_create_private_key()</code> , with <code>purpose flags</code> allowing its use for ECDSA operations.
in	<code>use_point_compression</code>	This flag is present for future use compatibility, but is not used for this product version of BK. For this product version, this value has to be set to False (no point compression). Any other values will result in the return code <code>IID_INVALID_PARAMETERS</code> .
in	<code>csr_subjects</code>	Pointer to an input buffer for a <code>bk_certificate_subject_t</code> structure. The elements of this structure contain the values which will be included in the distinguished name of the CSR's Subject field.
out	<code>csr</code>	Pointer to an output buffer of byte size <code>csr_length</code> , which will hold the DER-encoded binary CSR.
in, out	<code>csr_length</code>	Pointer to an input/output buffer for a uint16, which as an input holds the allocated size in bytes of the <code>csr</code> output buffer, and as an output (upon successful execution) will hold the exact size in bytes of the returned CSR. The exact size of a CSR is not entirely deterministic from its inputs, but the maximum possible size (which needs to be allocated for <code>csr</code>) can be precomputed using <code>bk_maxsizeof_csr()</code> .



Returns

[IID_SUCCESS](#) if successful, or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#), [IID_INVALID_PRIVATE_KEY_CODE](#) or [IID_PKI_BUFFER_TOO_SMALL](#) otherwise

2.1.6.24. [bk_maxsizeof_selfsigned_certificate\(\)](#) [iid_return_t](#) [bk_maxsizeof_selfsigned_certificate](#) (
 const [bk_ecc_private_key_code_t](#) *const *private_key_code*,
 const bool *use_point_compression*,
 const uint8_t *const *serial*,
 const uint16_t *serial_length*,
 const [bk_certificate_subject_t](#) *const *ssc_subjects*,
 uint16_t *const *maxcertificate_length*)

Precomputes the (maximum) byte size of a self-signed certificate (SSC).

This function computes the maximum possible size a DER-encoded binary [X.509](#)-formatted self-signed certificate can have for a presented elliptic curve private key (code) and a given set of certificate information fields and subject information fields identifying the owner of that key.

Note

Due to a combination of the way DER-encoding works, and the non-determinism involved in the ECDSA-signature on the self-signed certificate, the precise size in bytes of a self-signed certificate is also non-deterministic and can only be exactly determined once the certificate is generated with [bk_create_selfsigned_certificate\(\)](#). However, based on a given set of inputs, it is possible to precompute the maximum possible byte size of the certificate, which is what [bk_maxsizeof_selfsigned_certificate\(\)](#) does. This is the size which needs to be allocated in memory for the `cert` output buffer of [bk_create_selfsigned_certificate\(\)](#). The exact byte size of a generated certificate is returned by [bk_create_selfsigned_certificate\(\)](#) as `certificate_length`.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by [IID_SUCCESS](#), this function will have written the maximum size in bytes. On error the output buffer content is undefined

Parameters

in	<i>private_key_code</i>	Pointer to an input buffer for a bk_ecc_private_key_code_t which holds the elliptic curve private key for which the self-signed certificate size needs to be computed. Note: a private key code used for creating a self-signed certificate shall have been created with bk_create_private_key() , with [purpose flags] (bk_defgroup_keypurpose) allowing its use for ECDSA operations.
in	<i>use_point_compression</i>	This flag is present for future use compatibility, but is not used for this product version of BK. For this product version, this value has to be set to False (no point compression). Any other values will result in the return code IID_INVALID_PARAMETERS .



Parameters

in	<i>serial</i>	Pointer to an input buffer of byte size <i>serial_length</i> , which holds a binary string which will (optionally) be included as the serial number of the generated certificate. Note: this <i>certificate</i> serial number is not to be confused with the serial number which can be part of the subject's distinguished name, which is passed in the <i>subject_sn</i> field of <i>ssc_subjects</i> . The former is used to identify the certificate itself, while the latter is (part of) the identity of the certificate's owner. If <i>serial_length</i> is zero, <i>serial</i> must be zero.
in	<i>serial_length</i>	Value which specifies the size in bytes of the <i>serial</i> buffer. Note: When this value is zero, no serial number field will be included in the generated certificate. If <i>message_length</i> is zero, <i>message</i> must be zero.
in	<i>ssc_subjects</i>	Pointer to an input buffer for a bk_certificate_subject_t structure. The elements of this structure contain the values which will be included in the distinguished name of the certificate's Subject field. Note: since this is a <i>self-signed</i> certificate, the same values will be included in the <i>Issuer's</i> distinguished name as well.
out	<i>maxcertificate_length</i>	Pointer to a uint16 which will hold the maximal possible size in bytes of a self-signed certificate generated on the provided inputs with bk_create_selfsigned_certificate() .

Returns

[IID_SUCCESS](#) if successful, or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#) or [IID_INVALID_PRIVATE_KEY_CODE](#) otherwise

2.1.6.25. [bk_create_selfsigned_certificate\(\)](#) [iid_return_t](#) [bk_create_selfsigned_certificate](#) (
 const [bk_ecc_private_key_code_t](#) *const *private_key_code*,
 const bool *use_point_compression*,
 const uint8_t *const *serial*,
 const uint16_t *serial_length*,
 const char *const *valid_start*,
 const char *const *valid_end*,
 const [bk_certificate_subject_t](#) *const *ssc_subjects*,
 uint8_t *const *certificate*,
 uint16_t *const *certificate_length*)

Creates a self-signed certificate (SSC) for an elliptic curve key pair.

This function creates an [X.509](#)-formatted self-signed certificate for a presented elliptic curve private key (code) and a limited set of certificate information fields and subject information fields identifying the owner of that key. The self-signed certificate is returned as a DER-encoded binary string.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by [IID_SUCCESS](#), this function will have written the SSC as a DER-encoded byte sequence to the output buffer and have returned its size in bytes. On error the output buffer content is undefined



Parameters

in	<i>private_key_code</i>	Pointer to an input buffer for a bk_ecc_private_key_code_t which holds the elliptic curve private key for which a self-signed certificate will be created. The public key corresponding to this private key will be included in the certificate, while the private key will be used to (self-)sign the certificate. Note: a private key code used for creating a self-signed certificate shall have been created with bk_create_private_key() , with purpose flags allowing its use for ECDSA operations.
in	<i>use_point_compression</i>	This flag is present for future use compatibility, but is not used for this product version of BK. For this product version, this value has to be set to False (no point compression). Any other values will result in the return code IID_INVALID_PARAMETERS .
in	<i>serial</i>	Pointer to an input buffer of byte size <i>serial_length</i> , which holds a binary string which will (optionally) be included as the serial number of the generated certificate. Note: this <i>certificate</i> serial number is not to be confused with the serial number which can be part of the subject's distinguished name, which is passed in <i>subject_sn</i> field of <i>ssc_subjects</i> . The former is used to identify the certificate itself, while the latter is (part of) the identity of the certificate's owner. If <i>serial_length</i> is zero, <i>serial</i> must be zero.
in	<i>serial_length</i>	Value which specifies the size in bytes of the <i>serial</i> buffer. Note: When this value is zero, no serial number field will be included in the generated certificate. If <i>message_length</i> is zero, <i>message</i> must be zero.
in	<i>valid_start</i>	Pointer to an input buffer of byte size 15 which holds a \0-terminated ASCII character string specifying the start date of the certificate's <i>Validity</i> period. This string needs to be formatted as "YYYYMMDDhhmmss". Note: the presented string is interpreted and included in the certificate as GMT date/time.
in	<i>valid_end</i>	Pointer to an input buffer of byte size 15 which holds a \0-terminated ASCII character string specifying the end date of the certificate's <i>Validity</i> period. This string needs to be formatted as "YYYYMMDDhhmmss". Note: the presented string is interpreted and included in the certificate as GMT date/time.
in	<i>ssc_subjects</i>	Pointer to an input buffer for a bk_certificate_subject_t structure. The elements of this structure contain the values which will be included in the distinguished name of the certificate's Subject field. Note: since this is a <i>self-signed</i> certificate, the same values will be included in the <i>Issuer's</i> distinguished name as well.
out	<i>certificate</i>	Pointer to an output buffer of byte size <i>certificate_length</i> , which will hold the DER-encoded binary self-signed certificate.
in, out	<i>certificate_length</i>	Pointer to an input/output buffer for a uint16, which as an input holds the allocated size in bytes of the <i>certificate</i> output buffer, and as an output (upon successful execution) will hold the exact size in bytes of the returned certificate. The exact size of a certificate is not entirely deterministic from its inputs, but the maximum possible size (which needs to be allocated for <i>certificate</i>) can be precomputed using bk_maxsizeof_selfsigned_certificate() .

Returns

[IID_SUCCESS](#) if successful, or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#), [IID_INVALID_PRIVATE_KEY_CODE](#) or [IID_PKI_BUFFER_TOO_SMALL](#) otherwise



2.1.6.26. bk_write_ec_private_key() `iid_return_t` bk_write_ec_private_key (
 const `bk_ecc_curve_t` curve,
 uint8_t *const private_key,
 uint8_t *const ECPrivateKey,
 uint16_t *const ECPrivateKey_length)

Creates a DER-encoded representation of an elliptic curve private key and its associated public key following the ECPrivateKey ASN.1 syntax specified in [RFC 5915](#).

This function creates a DER-encoded representation of an elliptic curve private key and its associated public key following the ECPrivateKey ASN.1 syntax specified in [RFC 5915](#).

Note: This format is equal to OpenSSL output of an EC keypair generated with the option "-outform der"

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by `IID_SUCCESS`, this function will have written the private key as a DER-encoded byte sequence to the output buffer and have returned its size in bytes. On error the output buffer content is undefined

Parameters

in	curve	Specifies the named elliptic curve on which the private key is defined. It must be a valid curve type of the <code>bk_ecc_curve_t</code> enumeration.
in	private_key	Pointer to an input buffer which holds the raw binary elliptic curve private key (e.g. as returned by <code>bk_get_private_key()</code>) to be written, with a size specified by the <code>curve</code> parameter. For the given configuration, the buffer type <code>bk_ecc_private_key_t</code> can hold all raw private keys.
out	ECPrivateKey	Pointer to an output buffer of byte size <code>ECprivatekey_length</code> , which will hold the DER-encoded binary private key and its associated public key.
in, out	ECPrivateKey_length	Pointer to an input/output buffer for a uint16, which as an input holds the allocated size in bytes of the <code>ECPrivateKey</code> output buffer, and as an output (upon successful execution) will hold the exact size in bytes of the returned <code>ECPrivateKey</code> . The exact size of an <code>ECPrivateKey</code> is determined by the used <code>curve</code> . For the given configuration, the buffer type <code>bk_der_private_key_info_t</code> can hold all DER-encoded representations of an elliptic curve private key .

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PRIVATE_KEY`, `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED` or `IID_PKI_BUFFER_TOO_SMALL` otherwise



2.1.6.27. bk_write_subject_public_key_info() `iid_return_t bk_write_subject_public_key_info (`
`const bk_ecc_curve_t curve,`
`uint8_t *const std_public_key,`
`uint8_t *const subject_public_key_info,`
`uint16_t *const subject_public_key_info_length)`

Creates a DER-encoded representation of an elliptic curve public key following the `subject_public_key_info` ASN.1 syntax specified in [RFC 5480](#).

This function creates a DER-encoded representation of an elliptic curve public key following the `SubjectPublicKey` ASN.1 syntax specified in [RFC 5480](#) as can be used in [X.509](#) certificates.

Note: This format is equal to OpenSSL public key output with the option "-pubout -outform der".

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by `IID_SUCCESS`, this function will have written the public key as a DER-encoded byte sequence to the output buffer and have returned its size in bytes. On error the output buffer content is undefined

Parameters

in	<i>curve</i>	Specifies the named elliptic curve on which the public key is defined. It must be a valid curve type of the <code>bk_ecc_curve_t</code> enumeration.
in	<i>std_public_key</i>	Pointer to an input buffer which holds the elliptic curve public key to be written. The expected input is in X9.62 binary format (e.g. as returned by <code>bk_derive_public_key()</code> or <code>bk_export_public_key()</code>). Its size in bytes is determined by the used <i>curve</i> . For the given configuration, the buffer type <code>bk_ecc_std_public_key_t *</code> can hold all X9.62 binary format public keys.
out	<i>subject_public_key_info</i>	Pointer to an output buffer of byte size <code>subject_public_key_info_length</code> , which will hold the DER-encoded binary public key.
in, out	<i>subject_public_key_info_length</i>	Pointer to an input/output buffer for a <code>uint16</code> , which as an input holds the allocated size in bytes of the <code>subject_public_key_info</code> output buffer, and as an output (upon successful execution) will hold the exact size in bytes of the returned <code>subject_public_key_info</code> . The exact size of an <code>subject_public_key_info</code> is determined by the used <i>curve</i> . For the given configuration, the buffer type <code>bk_der_public_key_info_t</code> can hold all DER-encoded representations of an elliptic curve public key.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PUBLIC_KEY`, `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED` or `IID_PKI_BUFFER_TOO_SMALL` otherwise.



2.1.6.28. bk_write_ecdsa_sig_value() `iid_return_t bk_write_ecdsa_sig_value (`
 `const bk_ecc_curve_t curve,`
 `const uint8_t *const signature,`
 `uint8_t *const ecdsa_sig_value,`
 `uint16_t *const ecdsa_sig_value_length)`

Creates a DER-encoded representation of an ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax specified in [RFC 3279](#).

This function creates a DER-encoded representation of an ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax specified in [RFC 3279](#) as can be used in [X.509](#) certificates.

Note: This format is equal to the default OpenSSL ECDSA signature output.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by `IID_SUCCESS`, this function will have written the ECDSA signature as a DER-encoded byte sequence to the output buffer and have returned its size in bytes. On error the output buffer content is undefined

Parameters

in	<i>curve</i>	Specifies the named elliptic curve on which the ECDSA signature was generated. It must be a valid curve type of the <code>bk_ecc_curve_t</code> enumeration.
in	<i>signature</i>	Pointer to an input buffer which holds the raw binary ECDSA signature (e.g. as returned by <code>bk_ecdsa_sign()</code>) to be written, with a size in bytes is determined by the used <code>curve</code> . For the given configuration, the buffer type <code>bk_ecc_signature_t</code> can hold all signatures.
out	<i>ecdsa_sig_value</i>	Pointer to an output buffer of byte size <code>ecdsa_sig_value_length</code> , which will hold the DER-encoded binary ECDSA signature.
in, out	<i>ecdsa_sig_value_length</i>	Pointer to an input/output buffer for a <code>uint16</code> , which as an input holds the allocated size in bytes of the <code>ecdsa_sig_value</code> output buffer, and as an output (upon successful execution) will hold the exact size in bytes of the returned <code>ecdsa_sig_value</code> . The exact size of an <code>ecdsa_sig_value</code> depends on the used [curve] (<code>bk_ecc_curve_t</code>) as well as the signature data. For the given configuration, the buffer type <code>bk_der_sig_value_t</code> can hold all DER-encoded representations of an elliptic curve signature.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED` or `IID_PKI_BUFFER_TOO_SMALL` otherwise



2.1.6.29. `bk_read_ec_private_key()` `iid_return_t` `bk_read_ec_private_key` (
 `const uint8_t *const ECPrivateKey,`
 `uint8_t *const private_key,`
 `uint16_t * private_key_length,`
 `bk_ecc_curve_t *const curve`)

Reads a DER-encoded representation of an elliptic curve private key and its associated public key following the ECPrivateKey ASN.1 syntax specified in [RFC 5915](#).

This function reads a DER-encoded representation of an elliptic curve private key and its associated public key following the ECPrivateKey ASN.1 syntax specified in [RFC 5915](#).

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by `IID_SUCCESS`, this function will have read the private key and its associated X9.62 binary format public key from a DER-encoded byte sequence to the output buffers and have returned the `curve_id`. On error the output buffer content is undefined. This function can return an error for corrupted DER encoded input, but also if it refers to unsupported data or method identifiers.

Parameters

in	<code>ECPrivateKey</code>	Pointer to an input buffer with the ECPrivateKey byte stream, which holds the DER-encoded binary private key.
out	<code>private_key</code>	Pointer to an output buffer which will hold the read elliptic curve private key. The output will be in raw format (e.g. as accepted by <code>bk_create_private_key()</code>).
in, out	<code>private_key_length</code>	Pointer to an input/output buffer for a uint16, which as an input holds the allocated size in bytes of the <code>private_key</code> output buffer, and as an output (upon successful execution) will hold the exact size in bytes of the returned private key. The size of the private key is determined by the elliptic curve ID which is encoded in the DER input. For the given configuration, the buffer type <code>bk_ecc_private_key_t</code> can hold all raw private keys.
out	<code>curve</code>	Pointer to a buffer to hold the specifier of the named elliptic curve on which the private key is defined. It must be a valid curve type of the <code>bk_ecc_curve_t</code> enumeration.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PRIVATE_KEY`, `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_PKI_INVALID_DER_INPUT` or `IID_PKI_BUFFER_TOO_SMALL` otherwise

2.1.6.30. `bk_read_subject_public_key_info()` `iid_return_t` `bk_read_subject_public_key_info` (
 `const uint8_t *const subject_public_key_info,`



```
uint8_t *const std_public_key,  
uint16_t * std_public_key_length,  
bk_ecc_curve_t *const curve )
```

Reads a DER-encoded representation of an elliptic curve public key following the `subject_public_key_info` ASN.1 syntax specified in [RFC 5480](#).

This function reads a DER-encoded representation of an elliptic curve public key following the `SubjectPublicKey` ASN.1 syntax specified in [RFC 5480](#), as can be used in [X.509](#) certificates.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code `IID_NOT_ALLOWED`, without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by `IID_SUCCESS`, this function will have read the public key from a DER-encoded byte sequence to the output buffer and have returned the `curve_id`. On error the output buffer content is undefined. This function can return an error for corrupted DER encoded input, but also if it refers to unsupported data or method identifiers.

Parameters

in	<i>subject_public_key_info</i>	Pointer to an input buffer with the <code>subject_public_key_info</code> byte stream, which holds the DER-encoded binary public key.
out	<i>std_public_key</i>	Pointer to an output buffer which will hold the read elliptic curve public key. The output will be in X9.62 binary format (e.g. as accepted by bk_import_public_key()).
in, out	<i>std_public_key_length</i>	Pointer to an input/output buffer for a <code>uint16</code> , which as an input holds the allocated size in bytes of the <code>std_public_key</code> output buffer, and as an output (upon successful execution) will hold the exact size in bytes of the returned public key. The size of the public key is determined by the elliptic curve ID which is encoded in the DER input. For the given configuration, the buffer type bk_ecc_std_public_key_t can hold all X9.62 binary format public keys.
out	<i>curve</i>	Pointer to a buffer to hold the specifier of the named elliptic curve on which the X9.62 binary format public key is defined. It must be a valid curve type of the bk_ecc_curve_t enumeration.

Returns

`IID_SUCCESS` if successful, or `IID_INVALID_PUBLIC_KEY`, `IID_INVALID_PARAMETERS`, `IID_NOT_ALLOWED`, `IID_PKI_INVALID_DER_INPUT` or `IID_PKI_BUFFER_TOO_SMALL` otherwise

2.1.6.31. `bk_read_ecdsa_sig_value()` `iid_return_t` `bk_read_ecdsa_sig_value` (
 const [bk_ecc_curve_t](#) *curve*,



```
const uint8_t *const ecdsa_sig_value,  
uint8_t *const signature )
```

Reads a DER-encoded representation of an ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax specified in [RFC 3279](#).

This function reads a DER-encoded representation of an ECDSA signature following the ECDSA-Sig-Value ASN.1 syntax specified in [RFC 3279](#), as is used in [X.509](#) certificates.

Precondition

A call to this function is only allowed when BK is in the *Enrolled* or *Started* state. Called from any other operational state, this function will return immediately with return code [IID_NOT_ALLOWED](#), without taking any action.

Postcondition

A call to this function will not change the operational state of BK. If successful, signalled by [IID_SUCCESS](#), this function will have read the ECDSA signature from a DER-encoded byte sequence to the output buffer and have returned its size in bytes. On error the output buffer content is undefined. This function can return an error for corrupted DER encoded input, but also if it refers to unsupported data or method identifiers.

Parameters

in	<i>curve</i>	Specifies the named elliptic curve on which the ECDSA signature was generated. It must be a valid curve type of the bk_ecc_curve_t enumeration. Note: this reader function needs to know the curve on which the signature was generated in order to correctly parse it (since the DER signature itself does not contain the curve information). The (expected) curve can be obtained from the public key (code) with which one intends to validate this signature.
in	<i>ecdsa_sig_value</i>	Pointer to an input buffer holding the DER-encoded binary ECDSA signature.
out	<i>signature</i>	Pointer to an output buffer which will hold the ECDSA signature (e.g. as accepted by bk_ecdsa_verify()) to be read, with a size in bytes determined by the used [curve] (bk_ecc_curve_t). For the given configuration, the buffer type bk_ecc_signature_t can hold all signatures.

Returns

[IID_SUCCESS](#) if successful, or [IID_INVALID_PARAMETERS](#), [IID_NOT_ALLOWED](#), [IID_PKI_INVALID_DER_INPUT](#) or [IID_PKI_BUFFER_TOO_SMALL](#) otherwise

2.2. Return Codes

Return codes returned by the top-level functions.

Enumerations

- enum [iid_return_base_t](#) {
 [IID_RETURN_BASE](#) = 0x01u ,



```

IID_RETURN_BK = 0x20u ,
IID_RETURN_ECC = 0x30u ,
IID_RETURN_PKI = 0x40u ,
IID_AES_BASE = 0xE0u }

```

Return code base values per component.

```

• enum iid_return_t {
    IID_SUCCESS = 0x00u ,
    IID_NOT_ALLOWED = ((uint32_t) IID_RETURN_BASE + 0x00u) ,
    IID_INVALID_PARAMETERS = ((uint32_t) IID_RETURN_BASE + 0x01u) ,
    IID_NO_FREE_HANDLE = ((uint32_t) IID_RETURN_BASE + 0x02u) ,
    IID_INVALID_HANDLE = ((uint32_t) IID_RETURN_BASE + 0x03u) ,
    IID_INVALID_CONTEXT = ((uint32_t) IID_RETURN_BASE + 0x04u) ,
    IID_INVALID_MAC = ((uint32_t) IID_RETURN_BASE + 0x05u) ,
    IID_DATA_EXCEPTION = ((uint32_t) IID_RETURN_BASE + 0x0Du) ,
    IID_PROGRAM_EXCEPTION = ((uint32_t) IID_RETURN_BASE + 0x0Eu) ,
    IID_DEVICE_EXCEPTION = ((uint32_t) IID_RETURN_BASE + 0x0Fu) ,
    IID_ERROR_STARTUP_DATA = ((uint32_t) IID_RETURN_BK + 0x00u) ,
    IID_INVALID_AC = ((uint32_t) IID_RETURN_BK + 0x01u) ,
    IID_INVALID_KEY_CODE = ((uint32_t) IID_RETURN_BK + 0x02u) ,
    IID_SYM_NOT_ALLOWED = ((uint32_t) IID_RETURN_BK + 0x03u) ,
    IID_INVALID_SYMMETRIC_KEY_CODE = ((uint32_t) IID_RETURN_BK + 0x04u) ,
    IID_INVALID_PRIVATE_KEY = ((uint32_t)(IID_RETURN_ECC + 0x00)) ,
    IID_INVALID_PUBLIC_KEY = ((uint32_t)(IID_RETURN_ECC + 0x01)) ,
    IID_INVALID_SIGNATURE = ((uint32_t)(IID_RETURN_ECC + 0x02)) ,
    IID_INVALID_PUBLIC_KEY_CODE = ((uint32_t)(IID_RETURN_ECC + 0x03)) ,
    IID_INVALID_PRIVATE_KEY_CODE = ((uint32_t)(IID_RETURN_ECC + 0x04)) ,
    IID_ECC_NOT_ALLOWED = ((uint32_t)(IID_RETURN_ECC + 0x05)) ,
    IID_CURVE_MISMATCH = ((uint32_t)(IID_RETURN_ECC + 0x09)) ,
    IID_INVALID_COUNTER = ((uint32_t)(IID_RETURN_ECC + 0x06)) ,
    IID_INVALID_CRYPTGRAM = ((uint32_t)(IID_RETURN_ECC + 0x07)) ,
    IID_INVALID_SENDER = ((uint32_t)(IID_RETURN_ECC + 0x08)) ,
    IID_PKI_BUFFER_TOO_SMALL = ((uint32_t)(IID_RETURN_PKI + 0x00)) ,
    IID_PKI_INVALID_DER_INPUT = ((uint32_t)(IID_RETURN_PKI + 0x01)) ,
    IID_BUSY = 0xFFu ,
    IID_RETURN_ENUM_32BITS = 0xFFFFFFFFFu }

```

Return code values.

2.2.1. Detailed Description

Return codes returned by the top-level functions.

These return codes are used by the functions to return status information. They are grouped by component, which helps users to better understand the root cause of a potential issue when running the product.

2.2.2. Enumeration Type Documentation

2.2.2.1. iid_return_base_t `enum iid_return_base_t`

Return code base values per component.



Enumerator

IID_RETURN_BASE	Generic return codes base. Value used internally as the base value for all generic return codes.
IID_RETURN_BK	BK basic operation return codes base. Value used internally as the base value for all BK's basic operation specific return codes.
IID_RETURN_ECC	BK elliptic curve operations return codes base. Value used internally as the base value for all BK's ECC specific return codes.
IID_RETURN_PKI	BK PKI operations return codes base. Value used internally as the base value for all BK's PKI specific return codes.
IID_AES_BASE	AES return codes base. Value used internally as the base value for all AES return codes

2.2.2.2. iid_return_t enum iid_return_t

Return code values.

Enumerator

IID_SUCCESS	Successful execution. The called function ran successfully.
IID_NOT_ALLOWED	Successful execution. The given function call is not allowed in the current state.
IID_INVALID_PARAMETERS	Invalid function parameters. At least one of the parameters passed as argument to the 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.
IID_NO_FREE_HANDLE	No free handle available. A handle cannot be allocated for the specified streaming interface.
IID_INVALID_HANDLE	Invalid streaming interface handle. The streaming interface handle provided to this function does not point to an existing and opened streaming interface, or a call to open a new streaming interface could not allocate sufficient memory for creating it.
IID_INVALID_CONTEXT	Invalid streaming interface context. The context pointed by the provided handle does not exist, has been closed, or is corrupted
IID_INVALID_MAC	Authentication of streaming interface data failed. A call to close a streaming interface with an authentication verification (e.g. a MAC or ECIES stream) ends with a data authentication failure of the processed stream, thus a sign of data corruption.
IID_DATA_EXCEPTION	Program data error. The internal program data is corrupted or undetermined.
IID_PROGRAM_EXCEPTION	Program flow error. The internal program state is corrupted or undetermined.
IID_DEVICE_EXCEPTION	Device integrity error. The internal device state is corrupted or undetermined.



Enumerator

IID_ERROR_STARTUP_DATA	<p>Error startup data. The appointed SRAM address does not contain qualitative start-up data that can be used as an SRAM PUF by BK.</p> <p>Remarks</p> <p>This return value with high likelihood indicates a blocking error, since one is not able to get out of the current state (recalling the operation that returned this value will likely just return the same error code). A device repower is anyway required to get out of this situation, in combination with a resolution of the cause of this problem, if possible.</p>
IID_INVALID_AC	Invalid activation code. The activation code provided to <code>bk_start()</code> is not valid for this device, i.e. it was not generated by a successful call to <code>bk_enroll()</code> on the same device.
IID_INVALID_KEY_CODE	Invalid key code. The key code provided to <code>bk_unwrap()</code> or <code>bk_create_symmetric_key()</code> is not valid for this device, i.e. it was not generated by a successful call to <code>bk_wrap()</code> on the same device in the same cryptographic context.
IID_SYM_NOT_ALLOWED	Presented symmetric key code is not allowed to be used for this operation. The provided symmetric key code input does not have the right [key purpose flags](<code>bk_defgroup_keypurpose</code>) set for being used by the called function.
IID_INVALID_SYMMETRIC_KEY_CODE	Invalid symmetric key code. The provided symmetric key code input is not valid for the called function on this device, i.e. it was not generated by <code>bk_create_symmetric_key()</code> on this device.
IID_INVALID_PRIVATE_KEY	Invalid private key value. The provided private key (to <code>bk_create_private_key()</code> or <code>bk_derive_public_key()</code>) is invalid for the specified elliptic curve.
IID_INVALID_PUBLIC_KEY	Invalid public key value. The provided public key (to <code>bk_import_public_key()</code>) is invalid for the specified elliptic curve.
IID_INVALID_SIGNATURE	Invalid signature. The signature input provided to <code>bk_ecdsa_verify()</code> is not valid for the provided message under the provided public key.
IID_INVALID_PUBLIC_KEY_CODE	Invalid public key code. The provided public key code is not valid for the called function on this device, i.e. it was not generated by <code>bk_compute_public_from_private_key()</code> or <code>bk_import_public_key()</code> on this device.
IID_INVALID_PRIVATE_KEY_CODE	Invalid private key code. The provided private key code is not valid for the called function on this device, i.e. it was not generated by <code>bk_create_private_key()</code> on this device.
IID_ECC_NOT_ALLOWED	Presented elliptic curve key code is not allowed to be used for this operation. The provided private and/or public key code inputs do not have the right [key purpose flags](<code>bk_defgroup_keypurpose</code>) set for being used by the called function.
IID_CURVE_MISMATCH	Curve mismatch. The elliptic curves in the simultaneously provided private and public key code do not match.
IID_INVALID_COUNTER	Invalid cryptogram counter. The <code>counter64</code> input provided to <code>bk_generate_cryptogram()</code> results in a counter overflow, or the <code>counter64</code> input provided to <code>bk_process_cryptogram()</code> 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.



Enumerator

IID_INVALID_CRYPTOGRAM	Invalid cryptogram. 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	Invalid sender of cryptogram. 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.
IID_PKI_BUFFER_TOO_SMALL	Insufficient PKI output buffer space. The provided output buffer for the requested PKI structure (CSR of self-signed certificate) is too small to hold all data.
IID_PKI_INVALID_DER_INPUT	Invalid DER input. The provided DER encoded byte stream is either corrupted or holds unsupported data or method indicators.
IID_BUSY	Busy executing. The execution of the called function has not terminated.
IID_RETURN_ENUM_32BITS	Unused value, placed here to force the compiler to use 32 bits to represent values of this type.

2.3. Compiler Attributes

Compiler Attributes (for alignment and data packing)

Macros

- `#define EXPLICIT_FALLTHROUGH`
Attribute macro to avoid implicit fallthrough warnings.
- `#define PRE_PACKED`
Macro to force packing of data structures (prefix)
- `#define POST_PACKED __attribute__((packed))`
Macro to force packing of data structures (postfix)
- `#define PRE_ALIGN32`
Macro to force memory alignment on 32-bit (prefix)
- `#define PRE_ALIGN64`
Macro to force memory alignment on 64-bit (prefix)
- `#define PRE_ALIGN128`
Macro to force memory alignment on 128-bit (prefix)
- `#define PRE_ALIGN`
Macro to force memory alignment on the platform word size (prefix)
- `#define POST_ALIGN32 __attribute__((aligned(4)))`
Macro to force memory alignment on 32-bit (postfix)
- `#define POST_ALIGN64 __attribute__((aligned(8)))`
Macro to force memory alignment on 64-bit (postfix)
- `#define POST_ALIGN128 __attribute__((aligned(16)))`
Macro to force memory alignment on 128-bit (postfix)
- `#define POST_ALIGN __attribute__((aligned(4)))`
Macro to force memory alignment on the platform word size (postfix)

2.3.1. Detailed Description

Compiler Attributes (for alignment and data packing)



2.3.2. Macro Definition Documentation

2.3.2.1. EXPLICIT_FALLTHROUGH `#define EXPLICIT_FALLTHROUGH`

Attribute macro to avoid implicit fallthrough warnings.

2.3.2.2. PRE_PACKED `#define PRE_PACKED`

Macro to force packing of data structures (prefix)

2.3.2.3. POST_PACKED `#define POST_PACKED __attribute__((packed))`

Macro to force packing of data structures (postfix)

2.3.2.4. PRE_ALIGN32 `#define PRE_ALIGN32`

Macro to force memory alignment on 32-bit (prefix)

2.3.2.5. PRE_ALIGN64 `#define PRE_ALIGN64`

Macro to force memory alignment on 64-bit (prefix)

2.3.2.6. PRE_ALIGN128 `#define PRE_ALIGN128`

Macro to force memory alignment on 128-bit (prefix)

2.3.2.7. PRE_ALIGN `#define PRE_ALIGN`

Macro to force memory alignment on the platform word size (prefix)

Examples

[iid_bk_examples_standalone.c](#), and [iid_bk_examples_wolfssl.c](#).

**2.3.2.8. POST_ALIGN32** #define POST_ALIGN32 __attribute__((aligned(4)))

Macro to force memory alignment on 32-bit (postfix)

2.3.2.9. POST_ALIGN64 #define POST_ALIGN64 __attribute__((aligned(8)))

Macro to force memory alignment on 64-bit (postfix)

2.3.2.10. POST_ALIGN128 #define POST_ALIGN128 __attribute__((aligned(16)))

Macro to force memory alignment on 128-bit (postfix)

2.3.2.11. POST_ALIGN #define POST_ALIGN __attribute__((aligned(4)))

Macro to force memory alignment on the platform word size (postfix)

Examples

[iid_bk_examples_standalone.c](#), and [iid_bk_examples_wolfssl.c](#).

3. Example Documentation

3.1. iid_bk_examples_standalone.c

```
/*
 * Copyright 2022 Intrinsic ID B.V. All rights reserved.
 *
 * Usage of this software is permitted under a valid written license agreement
 * between you and Intrinsic ID B.V.
 */
/*
 * This example shows how to use some functions of BK
 * (init, enroll, start, stop, get_key, get_private_key)
 */
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include "iid_bk.h"
#include "iid_bk_examples_data.h"
/* Define the start address of the SRAM used for BK */
#define SRAM_PUF_ADDRESS    &sram_example
/* Define the start address of the AC in NVM */
#define AC_NVM_ADDRESS      0x12345678
void write_to_nvm()
{
}
/*
 * !!!!!!!!!!!
 *
 * IMPORTANT:
 * Each of these example functions assumes BK is in Uninitialized state at the beginning of the example function
 */
```




```
!!!!!!!
*/
int iid_bk_example_init(void)
{
    iid_return_t ret_val;
    /******
     * Initialize BK
     ******
    ret_val = bk_init(
        (uint8_t * const) SRAM_PUF_ADDRESS,
        BK_SRAM_PUF_SIZE_BYTES);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

int iid_bk_example_enroll_stop(void)
{
    iid_return_t ret_val;
    /* PRE_ALIGN and POST_ALIGN macros can be used to align a variable address to 32 bits */
    PRE_ALIGN uint8_t activation_code[BK_AC_SIZE_BYTES] POST_ALIGN;
    /******
     * Initialize BK
     ******
    ret_val = bk_init(
        (uint8_t * const) SRAM_PUF_ADDRESS,
        BK_SRAM_PUF_SIZE_BYTES);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Enroll the Device
     ******
    ret_val = bk_enroll(activation_code);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_enroll - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Stop BK
     ******
    ret_val = bk_stop();
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_stop - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /* Store the AC in NVM */
    write_to_nvm(AC_NVM_ADDRESS, activation_code, BK_AC_SIZE_BYTES);
    return EXIT_SUCCESS;
}

int iid_bk_example_start_stop(void)
{
    /* Enrollment must have been performed before the following code, as shown in the previous example */
    iid_return_t ret_val;
    /******
     * Initialize BK
     ******
    ret_val = bk_init(
        (uint8_t * const) SRAM_PUF_ADDRESS,
        BK_SRAM_PUF_SIZE_BYTES);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Start BK
     ******
    ret_val = bk_start(activation_code_example);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_start - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Stop BK
     ******
}
```



```
*****/
ret_val = bk_stop();
if (IID_SUCCESS != ret_val) {
    /* ... handle error ... */
    printf("Error: bk_stop - %u\n", ret_val);
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}
int iid_bk_example_get_key(void)
{
    /* To call bk_get_key, BK must be in Enrolled or Started state.
       For this example, we put BK in Started state before calling bk_get_key.
    */
    iid_return_t ret_val;
    /******
     * Initialize BK
     * *****/
    ret_val = bk_init(
        (uint8_t * const) SRAM_PUF_ADDRESS,
        BK_SRAM_PUF_SIZE_BYTES);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Start BK
     * *****/
    ret_val = bk_start(activation_code_example);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_start - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Get Key
     * *****/
    uint8_t key_type = BK_SYM_KEY_TYPE_256;
    uint8_t index = 0;
    uint8_t key[32];
    ret_val = bk_get_key(key_type, index, key);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_get_key - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
int iid_bk_example_get_private_key(void)
{
    /* To call bk_get_private_key, BK must be in Enrolled or Started state.
       For this example, we put BK in Started state before calling bk_get_private_key.
    */
    iid_return_t ret_val;
    bk_ecc_curve_t curve = BK_ECC_CURVE_NIST_P256;
    /* PRE_ALIGN and POST_ALIGN macros can be used to align a variable address to 32 bits */
    PRE_ALIGN uint8_t private_key[BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_BYTES] POST_ALIGN;
    /******
     * Initialize BK
     * *****/
    ret_val = bk_init(
        (uint8_t * const) SRAM_PUF_ADDRESS,
        BK_SRAM_PUF_SIZE_BYTES);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Start BK
     * *****/
    ret_val = bk_start(activation_code_example);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_start - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Get Private Key
     * *****/
    ret_val = bk_get_private_key(
```



```
    curve,
    NULL, /* no usage_context is used */
    0,    /* no usage context is used */
    BK_KEY_SOURCE_PUF_DERIVED,
    private_key);
if (IID_SUCCESS != ret_val) {
    /* ... handle error ... */
    printf("Error: bk_get_private_key - %u\n", ret_val);
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}

int iid_bk_example_generate_random(void)
{
    /* To call bk_generate_random, BK must be in Enrolled or Started state.
       For this example, we put BK in Started state before calling bk_generate_random.
    */
    iid_return_t ret_val;
    /******
     * Initialize BK
     * *****/
    ret_val = bk_init(
        (uint8_t * const) SRAM_PUF_ADDRESS,
        BK_SRAM_PUF_SIZE_BYTES);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Start BK
     * *****/
    ret_val = bk_start(activation_code_example);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_start - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    /******
     * Generate Random
     * *****/
    uint8_t data_buffer[32];
    ret_val = bk_generate_random(32, data_buffer);
    if (IID_SUCCESS != ret_val) {
        /* ... handle error ... */
        printf("Error: bk_generate_random - %u\n", ret_val);
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

3.2. iid_bk_examples_mbedtls.c

```
/*
 * Copyright 2022 Intrinsic ID B.V. All rights reserved.
 *
 * Usage of this software is permitted under a valid written license agreement
 * between you and Intrinsic ID B.V.
 */
/*
    This example shows how BK can be used to generate a PUF-derived elliptic curve key pair.
    This key pair is then used by Mbed TLS to compute an ECDSA signature.
    This code is compatible with Mbed TLS v2.x and v3.x.
*/
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include "iid_bk.h"
#include "iid_attributes.h"
#include "mbedtls/version.h"
#if (MBEDTLS_VERSION_MAJOR == 3)
#include "mbedtls/compat-2.x.h"
#else
#include "mbedtls/ecp.h"
#define MBEDTLS_PRIVATE(member) member
typedef enum {
    ECP_TYPE_NONE = 0,
    ECP_TYPE_SHORT_WEIERSTRASS, /*  $y^2 = x^3 + a x + b$  */
}
```



```
    ECP_TYPE_MONTGOMERY,          /* y^2 = x^3 + a x^2 + x */
} ecp_curve_type;
/*
 * Get the type of a curve
 */
static inline ecp_curve_type ecp_get_type(const mbedtls_ecp_group * grp)
{
    if (grp->G.X.p == NULL)
        return(ECP_TYPE_NONE);
    if (grp->G.Y.p == NULL)
        return(ECP_TYPE_MONTGOMERY);
    else
        return(ECP_TYPE_SHORT_WEIERSTRASS);
}
#endif
#include "mbedtls/ecdsa.h"
#include "mbedtls/sha256.h"
#include "mbedtls/platform.h"
#include "mbedtls/aes.h"
#include "iid_bk_examples_data.h"
#if !defined(MBEDTLS_ECDSA_C)
#error MBEDTLS_ECDSA_C must be defined
#endif
#if !defined(MBEDTLS_SHA256_C)
#error MBEDTLS_SHA256_C must be defined
#endif
/* Use secp256r1 curve */
#define ECPARAMS    MBEDTLS_ECP_DP_SECP256R1
/*****
 *      H E L P E R   F U N C T I O N S
 *****/
/* This function converts an Mbed TLS group ID to a BK curve ID */
bk_ecc_curve_t get_bk_curve_from_mbedtls_group_id(mbedtls_ecp_group_id mbedtls_group_id)
{
    bk_ecc_curve_t bk_curve;
    switch(mbedtls_group_id)
    {
#ifdef BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_BYTES
        case MBEDTLS_ECP_DP_SECP192R1:
            bk_curve = BK_ECC_CURVE_NIST_P192;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_BYTES
        case MBEDTLS_ECP_DP_SECP224R1:
            bk_curve = BK_ECC_CURVE_NIST_P224;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_BYTES
        case MBEDTLS_ECP_DP_SECP256R1:
            bk_curve = BK_ECC_CURVE_NIST_P256;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP256K1_PRIVATE_KEY_BYTES
        case MBEDTLS_ECP_DP_SECP256K1:
            bk_curve = BK_ECC_CURVE_NIST_K256;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_BYTES
        case MBEDTLS_ECP_DP_SECP384R1:
            bk_curve = BK_ECC_CURVE_NIST_P384;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_BYTES
        case MBEDTLS_ECP_DP_SECP521R1:
            bk_curve = BK_ECC_CURVE_NIST_P521;
            break;
#endif
        default:
            bk_curve = -1;
            break;
    }
    return bk_curve;
}
/* This function:
 * - uses BK to generate a PUF-derived (deterministic) private key
 * - stores this key in an Mbed TLS MPI
 */
int iid_bk_get_puf_derived_mbedtls_privkey(
    const mbedtls_ecp_group * grp,
    mbedtls_mpi * d,
    const uint8_t * usage_context,
    uint32_t usage_context_length)
```



```
{
    bk_ecc_curve_t bk_curve;
    iid_return_t iid_ret_val;
    int mbedtls_ret_val;
#ifdef MBEDTLS_ECP_MONTGOMERY_ENABLED /* Mbed TLS v3 */
    if (mbedtls_ecp_get_type(grp) == MBEDTLS_ECP_TYPE_MONTGOMERY) {
        return MBEDTLS_ERR_ECP_FEATURE_UNAVAILABLE;
    }
#elif (MBEDTLS_VERSION_MAJOR == 2) /* Mbed TLS v2 */
    if (ecp_get_type(grp) == ECP_TYPE_MONTGOMERY) {
        return MBEDTLS_ERR_ECP_FEATURE_UNAVAILABLE;
    }
#endif
#ifdef MBEDTLS_ECP_SHORT_WEIERSTRASS_ENABLED /* Mbed TLS v3 */
    if (mbedtls_ecp_get_type(grp) == MBEDTLS_ECP_TYPE_SHORT_WEIERSTRASS) {
#elif (MBEDTLS_VERSION_MAJOR == 2) /* Mbed TLS v2 */
    if (ecp_get_type(grp) == ECP_TYPE_SHORT_WEIERSTRASS) {
#endif
        mbedtls_mpi_grow(d, grp->N.MBEDTLS_PRIVATE(n));
        bk_curve = get_bk_curve_from_mbedtls_group_id(grp->id);
        assert(((size_t) d->MBEDTLS_PRIVATE(p)) % sizeof(int) == 0);
        iid_ret_val = bk_get_private_key(
            bk_curve,
            usage_context,
            usage_context_length,
            BK_KEY_SOURCE_PUF_DERIVED,
            (uint8_t *) d->MBEDTLS_PRIVATE(p));
        if (IID_SUCCESS == iid_ret_val) {
            mbedtls_ret_val = MBEDTLS_EXIT_SUCCESS;
        } else {
            mbedtls_ret_val = MBEDTLS_EXIT_FAILURE;
        }
        return mbedtls_ret_val;
    }
    return MBEDTLS_ERR_ECP_BAD_INPUT_DATA;
}

/* This function is an Mbed TLS-compatible callback function using BK's Random Number Generator.
Some Mbed TLS function can be given a pointer to a RNG callback function (e.g. mbedtls_ecdsa_genkey),
this function can be used for the f_rng parameter, with a NULL context for p_rng.

For example, to generate a random key pair using Mbed TLS and BK, the following call can be used:
mbedtls_ecdsa_genkey(&ctx, ECPARAMS, iid_zrng_mbedtls_wrapper, NULL);
*/
int iid_bk_generate_random_mbedtls_wrapper(
    void * p_rng,
    unsigned char * output,
    size_t output_len)
{
    iid_return_t iid_ret_val;
    int mbedtls_ret_val;
    /* No context required */
    (void) p_rng;
    assert(output_len <= UINT16_MAX);
    iid_ret_val = bk_generate_random((uint16_t) output_len, output);
    if (IID_SUCCESS == iid_ret_val) {
        mbedtls_ret_val = MBEDTLS_EXIT_SUCCESS;
    } else {
        mbedtls_ret_val = MBEDTLS_EXIT_FAILURE;
    }
    return mbedtls_ret_val;
}

/*
!!!!!!!!!!

IMPORTANT:
Each of these example functions assumes BK is in Uninitialized state at the beginning of the example function

!!!!!!!!!!

*/
/*****
 *                               E X A M P L E   C O D E
 *****/
/* This function is the 'main' function of this example file.

It:
- initializes BK (using an SRAM image for the example)
- starts BK (using a pre-generated activation code for the example)
- generates a Mbed TLS private key (using BK)
- computes the associated public key
- computes an ECDSA signature over a given message using the key pair
- verifies the generated ECDSA signature
*/
```



```
int iid_bk_example_mbedtls_ecdsa_signature(void)
{
    int ret_val;
    iid_return_t iid_ret_val;
    int mbedtls_ret_val = MBEDTLS_EXIT_FAILURE;
    mbedtls_ecdsa_context ctx_sign, ctx_verify;
    uint8_t hash[32];
    uint8_t signature[MBEDTLS_ECDSA_MAX_LEN] = {0};
    size_t signature_length;
    mbedtls_ecdsa_init(&ctx_sign);
    mbedtls_ecdsa_init(&ctx_verify);
    /******
     * Initialize and start BK *
     *****/
    /* This example uses a fixed SRAM image, but a real-case scenario requires uninitialized SRAM */
    iid_ret_val = bk_init(sram_example, sizeof(sram_example));
    if (IID_SUCCESS != iid_ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", iid_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
    /* This example assumes an activation code was generated previously during the enrollment phase */
    iid_ret_val = bk_start(activation_code_example);
    if (IID_SUCCESS != iid_ret_val) {
        /* ... handle error ... */
        printf("Error: bk_start - %u\n", iid_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
    /******
     * Generate a key pair for the signature *
     *****/
    mbedtls_ret_val = mbedtls_ecp_group_load(&(ctx_sign.MBEDTLS_PRIVATE(grp)), ECPARAMS);
    if (IID_SUCCESS != mbedtls_ret_val) {
        /* ... handle error ... */
        printf("Error: mbedtls_ecp_group_load - %d\n", mbedtls_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
    /* Generate a PUF-derived (deterministic) private key (under the form of an Mbed TLS MPI) using BK.
     For this example, we are using an empty usage context,
     but users may choose anything else for the usage context if they need to generate multiple keys.
     */
    mbedtls_ret_val = iid_bk_get_puf_derived_mbedtls_privkey(
        &(ctx_sign.MBEDTLS_PRIVATE(grp)),
        &(ctx_sign.MBEDTLS_PRIVATE(d)),
        NULL, /* no usage_context is used */
        0); /* no usage_context is used */
    if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
        /* ... handle error ... */
        printf("Error: iid_bk_get_puf_derived_mbedtls_privkey - %d\n", mbedtls_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
    /* Compute public key (from private key generated above) */
    mbedtls_ret_val = mbedtls_ecp_mul(
        &(ctx_sign.MBEDTLS_PRIVATE(grp)),
        &(ctx_sign.MBEDTLS_PRIVATE(Q)),
        &(ctx_sign.MBEDTLS_PRIVATE(d)),
        &(ctx_sign.MBEDTLS_PRIVATE(grp).G),
        iid_bk_generate_random_mbedtls_wrapper,
        NULL);
    if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
        /* ... handle error ... */
        printf("Error: mbedtls_ecp_mul - %d\n", mbedtls_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
    /******
     * Compute the SHA-256 hash of the message *
     *****/
    mbedtls_ret_val = mbedtls_sha256_ret(message_example, sizeof(message_example), hash, 0);
    if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
        /* ... handle error ... */
        printf("Error: mbedtls_sha256 - %d\n", mbedtls_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
    /******
     * Compute ECDSA signature over the message hash *
     *****/
}
```



```
/* This function also uses BK as a Random Number Generator for the ECDSA signature
   (by using the iid_bk_generate_random_mbedtls_wrapper callback function)
*/
mbedtls_ret_val = mbedtls_ecdsa_write_signature(
    &ctx_sign,
    MBEDTLS_MD_SHA256,
    hash,
    sizeof(hash),
    signature,
    sizeof(signature),
    #if (MBEDTLS_VERSION_MAJOR == 3)
    sizeof(signature),
    #endif
    &signature_length,
    iid_bk_generate_random_mbedtls_wrapper,
    NULL);
if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
    /* ... handle error ... */
    printf("Error: mbedtls_ecdsa_write_signature - %d\n", mbedtls_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
/*****
 * Create a separate verifying context
 *****/
/* Technically speaking a signature verification could be performed using ctx_sign,
   but for the sake of completeness this example uses a separate verification context ctx_verify,
   which only contains the public key.
*/
mbedtls_ret_val = mbedtls_ecp_group_copy(&ctx_verify.MBEDTLS_PRIVATE(grp), &ctx_sign.MBEDTLS_PRIVATE(grp));
if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
    /* ... handle error ... */
    printf("Error: mbedtls_ecp_group_copy - %d\n", mbedtls_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
mbedtls_ret_val = mbedtls_ecp_copy(&ctx_verify.MBEDTLS_PRIVATE(Q), &ctx_sign.MBEDTLS_PRIVATE(Q));
if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
    /* ... handle error ... */
    printf("Error: mbedtls_ecp_copy - %d\n", mbedtls_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
/*****
 * Verify ECDSA signature *
 *****/
mbedtls_ret_val = mbedtls_ecdsa_read_signature(
    &ctx_verify,
    hash,
    sizeof(hash),
    signature,
    signature_length);
if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
    /* ... handle error ... */
    printf("Error: mbedtls_ecdsa_read_signature - %d\n", mbedtls_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
ret_val = EXIT_SUCCESS;
exit:
mbedtls_ecdsa_free(&ctx_verify);
mbedtls_ecdsa_free(&ctx_sign);
return ret_val;
}
/* example showing encryption of 48 bytes of data,
   includes a test which shows how the data can be decrypted
   without passing the key */
int iid_bk_example_mbedtls_aes_encrypt(void)
{
    int ret_val;
    iid_return_t iid_ret_val;
    int mbedtls_ret_val = MBEDTLS_EXIT_FAILURE;
    mbedtls_aes_context enc;
    unsigned char aes_key[16];
    unsigned char original_data[48] = "reallylongstringfortestingpurposes";
    unsigned char encrypted_data[128];
    /*****
     * Initialize and start BK
     *****/
    iid_ret_val = bk_init(sram_example, sizeof(sram_example));
    if (IID_SUCCESS != iid_ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", iid_ret_val);
    }
}
```



```
    ret_val = EXIT_FAILURE;
    goto exit;
}
iid_ret_val = bk_start(activation_code_example);
if (IID_SUCCESS != iid_ret_val) {
    /* ... handle error ... */
    printf("Error: bk_start - %u\n", iid_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
/* get deterministic key from BK */
iid_ret_val = bk_get_key(BK_SYM_KEY_TYPE_128, 0, (uint8_t *)aes_key);
if (IID_SUCCESS != iid_ret_val) {
    /* ... handle error ... */
    printf("Error: aes key opening failing - %u\n", iid_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
mbedtls_ret_val = mbedtls_aes_setkey_enc( &enc, aes_key, 128 );
if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
    /* ... handle error ... */
    printf("Error: setting AES key with deterministic key - %u\n", mbedtls_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
mbedtls_ret_val = mbedtls_aes_crypt_ecb( &enc, MBEDTLS_AES_ENCRYPT, original_data, encrypted_data );
if (MBEDTLS_EXIT_SUCCESS != mbedtls_ret_val) {
    /* ... handle error ... */
    printf("Error: AES encryption - %u\n", mbedtls_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
ret_val = EXIT_SUCCESS;
exit:
    return ret_val;
}
```

3.3. iid_bk_examples_wolfssl.c

```
/*
 * Copyright 2022 Intrinsic ID B.V. All rights reserved.
 *
 * Usage of this software is permitted under a valid written license agreement
 * between you and Intrinsic ID B.V.
 */
/*
 * This example shows how BK can be used to generate a PUF-derived elliptic curve key pair.
 * This key pair is then used by WolfSSL to compute an ECDSA signature.
 * This code was written for WolfSSL v5.x.
 */
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include "iid_bk.h"
#include <wolfssl/options.h>
#include "config.h"
#include <wolfssl/wolfcrypt/wc_port.h>
#include <wolfssl/wolfcrypt/logging.h>
#include <wolfssl/wolfcrypt/types.h>
#include <wolfssl/wolfcrypt/ecc.h>
#include <wolfssl/wolfcrypt/signature.h>
#include <wolfssl/wolfcrypt/integer.h>
#include <wolfssl/wolfcrypt/error-crypt.h>
#include <wolfssl/wolfcrypt/aes.h>
#include "iid_bk_examples_data.h"
/*****
 *
 * H E L P E R F U N C T I O N S
 *
 *****/
/* bk_generate_random function can be used by WolfSSL to get random data.
 * To do so, WolfSSL MUST be built with -DWOLFSSL_USER_SETTINGS
 * and the wolfssl subdirectory of this example be added to its include path.
 */
#define USE_BK_RNG_FOR_WOLFSSL
#ifdef USE_BK_RNG_FOR_WOLFSSL
#ifdef WOLFSSL_USER_SETTINGS
#error "WolfSSL MUST be built with -DWOLFSSL_USER_SETTINGS"
#endif
#include "user_settings.h"
```




```
/* This function is called by WolfSSL inside its wc_RNG_GenerateBlock function. */
int iid_bk_rng_wolfssl_wrapper(unsigned char * output, unsigned int sz)
{
    iid_return_t iid_ret_val;
    int wolfssl_ret_val;
    assert(sz <= UINT16_MAX);
    iid_ret_val = bk_generate_random((uint16_t) sz, output);
    if (IID_SUCCESS == iid_ret_val) {
        wolfssl_ret_val = 0;
    } else {
        wolfssl_ret_val = RNG_FAILURE_E;
    }
    return wolfssl_ret_val;
}

/* This function is called by WolfSSL internally.
It is empty since the seeding of BK's RNG is done when calling bk_init. */
int iid_custom_rand_generate_seed(unsigned char * output, unsigned int sz)
{
    (void) output;
    (void) sz;
    return 0;
}

#endif /* USE_BK_RNG_FOR_WOLFSSL */

void zeroize_buffer(
    volatile void * const dst,
    size_t nbytes)
{
    volatile char * p = dst;
    while (nbytes-- > 0) {
        *p++ = 0;
    }
}

/* This function converts a WolfSSL curve ID to a BK curve ID */
bk_ecc_curve_t get_bk_curve_from_wolfssl_curve_id(ecc_curve_id curve_id)
{
    bk_ecc_curve_t bk_curve;
    switch (curve_id) {
#ifdef BK_ECC_CURVE_SECP192R1_PRIVATE_KEY_BYTES
        case ECC_SECP192R1:
            bk_curve = BK_ECC_CURVE_NIST_P192;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP224R1_PRIVATE_KEY_BYTES
        case ECC_SECP224R1:
            bk_curve = BK_ECC_CURVE_NIST_P224;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_BYTES
        case ECC_SECP256R1:
            bk_curve = BK_ECC_CURVE_NIST_P256;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP256K1_PRIVATE_KEY_BYTES
        case ECC_SECP256K1:
            bk_curve = BK_ECC_CURVE_NIST_K256;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP384R1_PRIVATE_KEY_BYTES
        case ECC_SECP384R1:
            bk_curve = BK_ECC_CURVE_NIST_P384;
            break;
#endif
#ifdef BK_ECC_CURVE_SECP521R1_PRIVATE_KEY_BYTES
        case ECC_SECP521R1:
            bk_curve = BK_ECC_CURVE_NIST_P521;
            break;
#endif
        default:
            bk_curve = -1;
            break;
    }
    return bk_curve;
}

/* This function:
- uses BK to generate a PUF-derived (deterministic) private key
- stores this key in an Mbed TLS MPI
- computes the associated public key
- checks the validity of the key pair
*/
int iid_bk_get_puf_derived_wolfssl_key(
    WC_RNG * rng,
    int keysize,
```



```
ecc_key * key,
int curve_id,
const uint8_t * usage_context,
uint32_t usage_context_length)
{
    int wolfssl_ret_val = MP_OKAY;
    bk_ecc_curve_t bk_curve;
    iid_return_t iid_ret_val;
    PRE_ALIGN bk_ecc_private_key_t bk_private_key POST_ALIGN;
    bk_curve = get_bk_curve_from_wolfssl_curve_id(curve_id);
    iid_ret_val = bk_get_private_key(
        bk_curve,
        usage_context,
        usage_context_length,
        BK_KEY_SOURCE_PUF_DERIVED,
        bk_private_key);
    if (IID_SUCCESS != iid_ret_val) {
        wolfssl_ret_val = ECC_PRIV_KEY_E;
    }
    if (MP_OKAY == wolfssl_ret_val) {
        wolfssl_ret_val = wc_ecc_set_curve(key, keysize, curve_id);
    }
    if (MP_OKAY == wolfssl_ret_val) {
        wolfssl_ret_val = mp_read_unsigned_bin(&(key->k), bk_private_key, keysize);
    }
    if (MP_OKAY == wolfssl_ret_val) {
        wolfssl_ret_val = wc_ecc_make_pub_ex(key, NULL, rng);
    }
    if (MP_OKAY == wolfssl_ret_val) {
        wolfssl_ret_val = wc_ecc_check_key(key);
    }
    zeroize_buffer(bk_private_key, sizeof(bk_private_key));
    return wolfssl_ret_val;
}
/*
!!!!!!!!!!!!

IMPORTANT:
Each of these example functions assumes BK is in Uninitialized state at the beginning of the example function

!!!!!!!!!!!!

*/
/*****
 *                               E X A M P L E   C O D E                               *
 *****/
/* This function is the 'main' function of this example file.

It:
- initializes BK (using an SRAM image for the example)
- starts BK (using a pre-generated activation code for the example)
- generates a WolfSSL key pair (using BK)
- computes an ECDSA signature over a given message using the key pair
- verifies the generated ECDSA signature

*/
int iid_bk_example_wolfssl_ecdsa_signature(void)
{
    int ret_val;
    iid_return_t iid_ret_val;
    int wolfssl_ret_val = 1;
    uint8_t signature[ECC_MAX_SIG_SIZE] = { 0 };
    word32 signature_size;
    int signature_type = WC_SIGNATURE_TYPE_ECC;
    ecc_key key;
    WC_RNG rng;
    /*****
     * Initialize and start BK *
     *****/
    iid_ret_val = bk_init(sram_example, sizeof(sram_example));
    if (IID_SUCCESS != iid_ret_val) {
        /* ... handle error ... */
        printf("Error: bk_init - %u\n", iid_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
    iid_ret_val = bk_start(activation_code_example);
    if (IID_SUCCESS != iid_ret_val) {
        /* ... handle error ... */
        printf("Error: bk_start - %u\n", iid_ret_val);
        ret_val = EXIT_FAILURE;
        goto exit;
    }
}
/*****/
```



```

    * Initialize WolfSSL RNG structure *
    *****/
wolfssl_ret_val = wc_InitRng(&rng);
if (0 != wolfssl_ret_val) {
    /* ... handle error ... */
    printf("Error: wc_InitRng - %d\n", wolfssl_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
/******
 * Generate a key pair for the signature *
*****/
wolfssl_ret_val = wc_ecc_init(&key);
if (0 != wolfssl_ret_val) {
    /* ... handle error ... */
    printf("Error: wc_ecc_init - %d\n", wolfssl_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
/* For the PUF-derived key in this example, an empty usage context is used */
wolfssl_ret_val = iid_bk_get_puf_derived_wolfssl_key(
    &rng,
    BK_ECC_CURVE_SECP256R1_PRIVATE_KEY_BYTES,
    &key,
    ECC_SECP256R1,
    NULL,
    0);
if (0 != wolfssl_ret_val) {
    /* ... handle error ... */
    printf("Error: iid_bk_get_puf_derived_wolfssl_key - %d\n", wolfssl_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
/******
 * Compute ECDSA signature over a SHA-256 hash of the message *
*****/
wolfssl_ret_val = wc_SignatureGetSize(signature_type, &key, sizeof(key));
if (BAD_FUNC_ARG == wolfssl_ret_val) {
    printf("Error: wc_SignatureGetSize - %d\n", wolfssl_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
signature_size = wolfssl_ret_val;
assert(signature_size <= sizeof(signature));
wolfssl_ret_val = wc_SignatureGenerate(
    WC_HASH_TYPE_SHA256,
    signature_type,
    message_example,
    sizeof(message_example),
    signature,
    &signature_size,
    &key,
    sizeof(key),
    &rng);
if (0 != wolfssl_ret_val) {
    /* ... handle error ... */
    printf("Error: wc_SignatureGenerate - %d\n", wolfssl_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
/******
 * Verify ECDSA signature *
*****/
wolfssl_ret_val = wc_SignatureVerify(
    WC_HASH_TYPE_SHA256,
    signature_type,
    message_example,
    sizeof(message_example),
    signature,
    signature_size,
    &key,
    sizeof(key));
if (0 != wolfssl_ret_val) {
    /* ... handle error ... */
    printf("Error: wc_SignatureVerify - %d\n", wolfssl_ret_val);
    ret_val = EXIT_FAILURE;
    goto exit;
}
ret_val = EXIT_SUCCESS;
exit:
    wc_ecc_free(&key);
    wc_FreeRng(&rng);

```



```
        return ret_val;
    }
    /* how to encrypt data with AES using the get_key function */
    int iid_bk_example_wolfssl_aes_encrypt(void)
    {
        int ret_val;
        iid_return_t iid_ret_val;
        int wolfssl_ret_val = 1;
        unsigned char aes_key[16];
        unsigned char iv[16];
        unsigned char original_data[48] = "reallylonggstringfortestingpurposes";
        unsigned char encrypted_data[128];
        Aes enc;
        /******
         * Initialize and start BK *
         *****/
        iid_ret_val = bk_init(sram_example, sizeof(sram_example));
        if (IID_SUCCESS != iid_ret_val) {
            /* ... handle error ... */
            printf("Error: bk_init - %u\n", iid_ret_val);
            ret_val = EXIT_FAILURE;
            goto exit;
        }
        iid_ret_val = bk_start(activation_code_example);
        if (IID_SUCCESS != iid_ret_val) {
            /* ... handle error ... */
            printf("Error: bk_start - %u\n", iid_ret_val);
            ret_val = EXIT_FAILURE;
            goto exit;
        }
        /* get deterministic key from BK */
        iid_ret_val = bk_get_key(BK_SYM_KEY_TYPE_128, 0, (uint8_t *)aes_key);
        if (IID_SUCCESS != iid_ret_val) {
            /* ... handle error ... */
            printf("Error: aes key opening failing - %u\n", iid_ret_val);
            ret_val = EXIT_FAILURE;
            goto exit;
        }
        /*iv is not set because it is not used for single block encryption (wc_AesEncryptDirect) used in this example
        wolfssl_ret_val = wc_AesSetKey(&enc, aes_key, 16, iv, AES_ENCRYPTION);
        if (wolfssl_ret_val != 0) {
            /* ... handle error ... */
            printf("Error: setting AES key with deterministic key - %u\n", wolfssl_ret_val);
            ret_val = EXIT_FAILURE;
            goto exit;
        }
        /* Does not have a return value */
        wc_AesEncryptDirect(&enc, encrypted_data, original_data);
        ret_val = EXIT_SUCCESS;
    exit:
        return ret_val;
    }
}
```