PSoC™ 64 provisioning specification

About this document

Scope and purpose

This document provides information and reference flows to enable provisioning of the PSoC™ 64 series of devices. It documents the format of necessary objects, including provisioning packets, JSON Web Tokens, and keys used in the provisioning process. The necessary system calls for provisioning are also defined.

Intended audience

This document is aimed at customers and provisioning partners for implementing provisioning algorithms.

Table of contents

1 Definition of terms

- **Original Equipment Manufacturer (OEM)**: A customer who purchases PSoC™ 64 and uses them in their product.
- **Hardware Security Module (HSM)**: A physical computing device that safeguards and manages digital keys for strong authentication, and that provides cryptographic processing. In the context of the PSoC™ 64 "Secure Boot" MCU, the HSM is a device-programming engine placed in a physically secured facility.
- **Root-of-Trust (RoT)**: This is an immutable process or identity used as the first entity in a trust chain. No ancestor entity can provide a trustable attestation (in digest or other form) for the initial code and data state of the RoT.
- **JavaScript Object Notation (JSON)** is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value).
- **JSON Schema** is a JSON-based format for describing the structure of JSON data (**https://json-schema.org/specification.html**).
- **JWT**: JSON Web Token (JWT) is an open, industry standard (**RFC 7519**) method to securely represent claims between two parties.
- **JWK**: JSON Web Key (JWK) is a **RFC7517**-compliant data structure that represents a cryptographic key.
- **Policies**: Policies are a collection of pre-defined (name, value) pairs that describe what is and is not allowed on the device. Most policies are enforced during boot-time by the RoT firmware in the device, while some can be interpreted and enforced by higher layers of software like CyBootloader.
- **Secured boot**: Refers to a bootup process where the firmware being run by the chip is trusted by using strong cryptographic schemes and an immutable RoT.
- **Elliptic-curve cryptography** (**ECC**) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields.
- **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) offers a variant of the Digital Signature Algorithm (DSA) which uses elliptic curve cryptography.
- **Unique device secret (UDS)** secrets or keys that are unique to the device.
- **Base64** is a group of binary-to-text encoding schemes that represent binary data (more specifically, a sequence of 8-bit bytes) in an ASCII string format by translating it into a radix-64 representation. Defined in **RFC4648**.
- **Base64url Encoding** is a Base64 encoding using the URL- and filename-safe character set.

2 Introduction

A PSoC™ 64 “Secure Boot” MCU device, after leaving the factory, has only the immutable boot code programmed with the Infineon-owned Root-Of-Trust (RoT) embedded into it. At this stage, the PSoC™ 64 device does not allow secured boot of a customer application.

To enable complete secured boot functionality of the PSoC™ 64, the OEM must perform the following actions:

- Replace the Infineon RoT with the OEM RoT to change the device ownership and allow all subsequent interactions to be signed by the OEM.
- Create the device identity by generating or injecting unique keys to the device.
- Inject a set of cryptographic keys which will be used to verify the OEM application.
- Inject various OEM assets like bootloader image, policies which define the product behavior, and certificates to provide a chain-of-trust to a higher certifying authority.

All these steps must be securely executed; the PSoC™ 64 device provides services to complete them. This process is called **provisioning**.

3 Data formats

This chapter describes the essential data types used for provisioning.

3.1 Keys format

The PSoC™ 64 secured boot process is based on the public-key cryptography (or asymmetric cryptography) system. The cryptographic system uses pairs of keys: **public keys**, which may be freely shared; and **private keys**, which are known only to the owner.

PSoC™ 64 supports only a public-key cryptography variant based on elliptic curves, particularly the ECDSA256 digital signing algorithm.

Cryptographic keys can be represented using several formats. PSoC™ 64 supports only the JSON Web Key (JWK) format, which is defined in **RFC7517**.

The following is an example of a JWK used by PSoC™ 64. The example JWK declares that the key is an Elliptic Curve key, it is used with the P-256 Elliptic Curve, and its x and y coordinates are the base64url-encoded values as shown below:

```
{
       "oem_priv_key": {
              "crv": "P-256",
              "d": "JVozA1oRvg-zSotMUbrGebV3oBhBaF1mqUyEn_Fdcqc",
              "kty": "EC",
              "use": "sig",
              "kid": "5",
              "x": "vfb7_jewTxpFVINcXdrZQJBArC5igrN0BLc783FigrM",
              "y": "9rBBUKXzpj1A5K7fxPtEaJdsfo7Jj_wsF7LTZLc-sPM"
       }
}
```

To distinguish keys, a specific JWK must be assigned to the JSON object with a defined name depending on the key purpose. In the above example, the name is "oem_priv_key". The following names are supported for input data during the provisioning process:

Key name	Description
"dev_priv_key"	Device private key; may be injected during identity creation. Required for secured identification of the device.
"grp_priv_key"	Group private key; may be injected during identity creation. Required for secured identification of the group of devices.
"hsm_pub_key"	HSM public key; used to sign all provisioning packets.
"oem_pub_key"	OEM public key. After the RoT transition from Infineon to the OEM, the RoT will be based on this key.
"cy_pub_key"	Infineon public key. The RoT of the device when it comes out of the factory is based on this key.
"custom_pub_key"	Customer public key that will be used to verify OEM applications or any other purpose defined by the OEM.

List of the members of a JWK JSON object:

JWK member name	Description
`"kty"`	Key type. Only "EC" (Elliptic Curve) is supported.
`"use"`	Intended use of the key. Only "sig" (signature) is supported
`"crv"`	Cryptographic curve used with the key. Only "P-256" is supported.
`"x"`	The "x" (x coordinate) parameter that contains the x coordinate for the Elliptic Curve point. It is represented as the base64url encoding of the octet string representation of the coordinate.
`"y"`	The "y" (y coordinate) parameter that contains the y coordinate for the Elliptic Curve point. It is represented as the base64url encoding of the octet string representation of the coordinate.
`"d"`	The "d" (ECC private key) parameter that contains the Elliptic Curve private key value. It is represented as the base64url encoding of the octet string representation of the private key value. This parameter is present only in private keys.
`"kid"`	The "kid" (key ID) parameter is used to match a specific key. The "kid" represents a slot number in the device key storage. Allowed values in provisioning packets are: • "3" - Infineon public key • "4" - HSM public key • "5" - OEM public key • "6"..."10" - Customer public keys • "12" - Group key
`"kver"`	Optional JWK member, which is present only in the Infineon public key. It is intended to differentiate the keys issued for different batches of devices.

3.2 Policy format

A policy is a JSON file that defines the device's operation modes, debug access, code updates etc. The policy is injected into the device during provisioning.

The policy consists of a collection of JSON tokens defined by Infineon and a number of tokens defined by the OEM. Most tokens from the policy are interpreted and enforced by the “Secure Boot” firmware in the device, but some are interpreted and enforced by a customer application.

The policy consists of two main parts:

- The **Debug Policy** describes what debug capabilities are exposed by the device.
- The **Boot and Upgrade Policy** specifies the number of images present in the system and their characteristics, and configures the bootloader and the policy upgrade feature.

The policy example is shown below:

```
{
    "debug": {
        "m0p": {
            "permission": "enabled",
            "control": "firmware",
            "key": 5
        },
        "m4": {
            "permission": "allowed",
            "control": "firmware",
            "key": 5
        },
        "system": {
            "permission": "enabled",
            "control": "firmware",
            "key": 5,
            "flashw": true,
            "flashr": true
        },
        "rma": {
            "permission": "allowed",
            "destroy_fuses": [
                {
                    "start": 888,
                    "size": 136
                }
            ],
            "destroy_flash": [
                {
                    "start": 268435456,
                    "size": 512
                }
            ],
            "key": 5
        }
    },
    "boot_upgrade": {
        "title": "upgrade_policy",
        "firmware": [
            {
                "boot_auth": [
                    5
                ],
                "id": 0,
                "launch": 1,
                "acq_win": 100,
                "monotonic": 0,
                "clock_flags": 578,
                "protect_flags": 1,
                "upgrade": false,
                "upgrade_mode": "swap",
                "resources": [
                    {
                        "type": "FLASH_PC1_SPM",
                        "address": 270336000,
                        "size": 65536
                    },
                    {
                        "type": "SRAM_SPM_PRIV",
                        "address": 135135232,
                        "size": 65536
                    },
                    {
                        "type": "SRAM_DAP",
                        "address": 135184384,
                        "size": 16384
                    },
                    {
                        "type": "STATUS_PARTITION",
                        "address": 270303232,
                        "size": 32768
                    }
                ]
            },
            {
                "boot_auth": [
                    8
                ],
                "id": 1,
                "monotonic": 0,
                "smif_id": 1,
                "acq_win": 100,
                "multi_image": 1,
                "upgrade": true,
                "version": "0.1",
                "rollback_counter": 0,
                "encrypt": false,
                "encrypt_key_id": 1,
                "resources": [
                    {
                        "type": "BOOT",
                        "address": 268435456,
                        "size": 786432
                    },
                    {
                        "type": "UPGRADE",
                        "address": 402653184,
                        "size": 786432
                    }
                ]
            },
            {
                "boot_auth": [
                    8
                ],
                "id": 16,
                "monotonic": 8,
                "smif_id": 1,
                "multi_image": 2,
                "upgrade": true,
                "version": "0.1",
                "rollback_counter": 0,
                "encrypt": false,
                "encrypt_key_id": 1,
                "resources": [
                    {
                        "type": "BOOT",
                        "address": 269221888,
                        "size": 786432
                    },
                    {
                        "type": "UPGRADE",
                        "address": 403439616,
                        "size": 786432
                    }
                ]
            }
        ],
        "reprogram": [
            {
                "start": 270336000,
                "size": 65536
            }
        ],
        "reprovision": {
            "boot_loader": true,
            "keys_and_policies": true
        }
    }
}
```

The policy structure and the meaning of specific tokens are described in the **“Secure Boot” SDK user guide**.

The policy file must comply with the policy templates, which are JSON schema files provided by Infineon. There are two separate policy templates:

- **Debug Policy template**
- **Boot & Upgrade Policy template**

The policy is not directly programmed into the device. It is injected as a part of the provisioning JWT packet, which is described in the following chapter.

3.3 JWT

Almost all input and output data used during the provisioning process is represented using JWT packets or tokens. JWT is a standard (**RFC7519**) URL-safe way of storing and transmitting any JSON object. A JWT packet is digitally signed to authenticate a sender and ensure the integrity of the payload.

A JWT is represented as a sequence of URL-safe parts separated by period ('.') characters. Each part contains a base64url-encoded value. PSoC™ 64 supports JWTs that consist of three parts: header, payload, and signature.

A JWT header is a base64url-encoded JSON object that has the following format:

```
{
  "alg": "ES256",
  "typ": "JWT"
}
```

The `"alg"` field defines the digital signature algorithm used for packet signing. `"ES256"` is the only supported value, which is a ECDSA256 digital signing algorithm.

A JWT payload is a base64url-encoded JSON object that contains a provisioning command or response (specific commands and responses will be described later).

A JWT signature is a base64url-encoded digital signature produced by the ECDSA256 algorithm.

The following is an example of one of the JWTs used for provisioning:

```
eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0eXBlIjoiT0VNX1JPVF9BVVRIIiwib2VtX3B1Yl9rZX
kiOnsiY3J2IjoiUC0yNTYiLCJrdHkiOiJFQyIsInVzZSI6InNpZyIsImtpZCI6IjUiLCJ4IjoidmZiN19qZ
XdUeHBGVklOY1hkclpRSkJBckM1aWdyTjBCTGM3ODNGaWdyTSIsInkiOiI5ckJCVUtYenBqMUE1SzdmeFB0
RWFKZHNmbzdKal93c0Y3TFRaTGMtc1BNIn0sImhzbV9wdWJfa2V5Ijp7ImNydiI6IlAtMjU2Iiwia3R5Ijo
iRUMiLCJ1c2UiOiJzaWciLCJraWQiOiI0IiwieCI6InNKTXNOLTJKbzI3a2M1MXdWSzd4SjJmUDlCRGt6QW
MyZlpFWk1sb2hIWEEiLCJ5IjoiTVdsdXptWGdYT3ZkUVFEWVgzeXkxVGs5UW9ITC05RFpoc3dacFkwWGU1V
SJ9LCJwcm9kX2lkIjoibXlfdGhpbmcifQ.0jQ9mDwjTAYNuJNSDIJCq4FvDptgSlnpyeBgxYkF2uabpfZhq
IXSr2vZ-l_5Vr58qfhrRwsrhaRGwJjVN9eVWw
```

Header	eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9	base64UrlEncode({ "alg": "ES256", "typ": "JWT" })

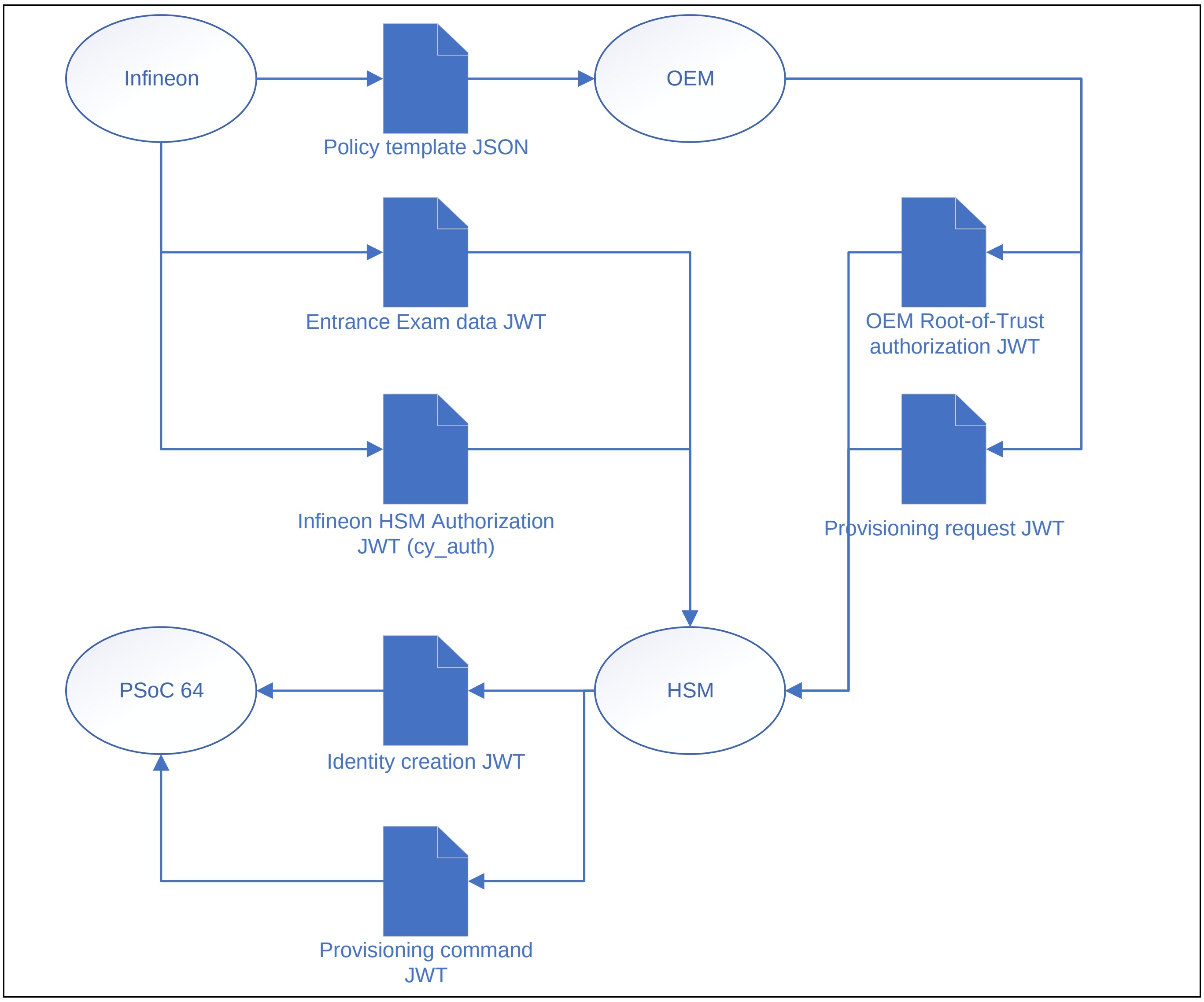
Payload	eyJ0eXBlIjoiT0VNX1JPVF9BVVRIIiwib2VtX3B1Yl9rZXk
	iOnsiY3J2IjoiUC0yNTYiLCJrdHkiOiJFQyIsInVzZSI6In
	NpZyIsImtpZCI6IjUiLCJ4IjoidmZiN19qZXdUeHBGVklOY
	1hkclpRSkJBckM1aWdyTjBCTGM3ODNGaWdyTSIsInkiOiI5
	ckJCVUtYenBqMUE1SzdmeFB0RWFKZHNmbzdKal93c0Y3TFR
	aTGMtc1BNIn0sImhzbV9wdWJfa2V5Ijp7ImNydiI6IlAtMj
	U2Iiwia3R5IjoiRUMiLCJ1c2UiOiJzaWciLCJraWQiOiI0I
	iwieCI6InNKTXNOLTJKbzI3a2M1MXdWSzd4SjJmUDlCRGt6
	QWMyZlpFWk1sb2hIWEEiLCJ5IjoiTVdsdXptWGdYT3ZkUVF
	EWVgzeXkxVGs5UW9ITC05RFpoc3dacFkwWGU1VSJ9LCJwcm
	9kX2lkIjoibXlfdGhpbmcifQ
		base64UrlEncode(
		{
		"type": "OEM_ROT_AUTH",
		"oem_pub_key": {
		"crv": "P-256",
		"kty": "EC",
		"use": "sig",
		"kid": "5",
		"x":
		"vfb7_jewTxpFVINcXdrZQJB
		ArC5igrN0BLc783FigrM",
		"y":
		"9rBBUKXzpj1A5K7fxPtEaJd
		sfo7Jj_wsF7LTZLc-sPM"
		},
		"hsm_pub_key": {
		"crv": "P-256",
		"kty": "EC",
		"use": "sig",
		"kid": "4",
		"x": "sJMsN-
		2Jo27kc51wVK7xJ2fP9BDkzA
		c2fZEZMlohHXA",
		"y":
		"MWluzmXgXOvdQQDYX3yy1Tk
		9QoHL-9DZhswZpY0Xe5U"
		},
		"prod_id": "my_thing"
		})

Signature	`0jQ9mDwjTAYNuJNSDIJCq4FvDptgSlnpyeBgxYkF2uabpfZhqIXSr2vZ-l_5Vr58qfhrRwsrhaRGwJjVN9eVWw`	`ECDSASHA256(`
		
		`base64UrlEncode(header) + "." +`
		
		`base64UrlEncode(payload), KEY)`

4 Provisioning data flow

This chapter explains the interactions between the actors (Infineon, OEM, and HSM) and data transfer between them in the form of JWT packets.

The security of the provisioning process is ensured by the fact that all data exchange happens using JWT packets signed with each actor's private key. Therefore, each side can validate the authenticity and integrity of the input data and requested operations. The following diagram shows the provisioning data flow.



4.1 Entrance exam packet

It is assumed that any step in a supply and manufacturing chain of the PSoC™ 64 device is not trusted. This means that any manipulation during these steps that modifies eFuses or flash memory bits must lead to rejection of devices before they are provisioned with their credentials. An "entrance exam" is required before provisioning the device that determines that there are no modifications of eFuses or flash memory bits and the device is "trustworthy".

The entrance exam is conducted by reading specific memory regions and comparing the read values to the reference values provided by Infineon. Infineon provides the entrance exam reference data as a JWT packet signed with the Infineon private key. Therefore, the HSM can verify the authenticity of the entrance exam JWT, and then using the JWT, verify that the device is genuine.

The entrance exam JWT allows the verification of the following:

- device ID
- hash of the immutable boot code
- immutable boot code version
- immutable boot code execution status
- absence of any applications in the flash memory

The entrance exam JWT also contains the information required to execute system calls such as the IPC structure number, and the address and size of the parameter region in the RAM.

The following is a reduced-for-clarity example of the entrance exam JSON object (the payload of the entrance exam JWT):

```
{
  "type": "ENTRANCE_EXAM",
  "ahb_reads": [
    {
      "value": "0xF0000000",
      "description": "Firmware status: 0xFx in MSB-no firmware image, 23-bit set - 
Asset hash failure",
      "address": "0x080FE000",
      "mask": "0xF0800000"
    },
    {
      "value": "0xE4531200",
      "description": "SI_ID: Indicates Silicon ID and Rev of the device",
      "address": "0x16000000",
      "mask": "0x0000FF00"
    }
  ],
  "ahb_reads8": [
    {
      "value": "0x08",
      "description": "ASSET_HASH byte 0",
      "address": "0x402C0840",
      "mask": "0xFF"
    },
    {
      "value": "0xE3",
      "description": "ASSET_HASH byte 1",
      "address": "0x402C0841",
      "mask": "0xFF"
    }
  ],
  "region_hashes": [
    {
      "size": "0x001E0000",
      "description": "Run syscall in compare mode to check that user flash is
empty.",
      "hash_id": 255,
      "address": "0x10000000",
      "value": "00"
    }
  ],
  "parameter_region": "0x080EC000",
  "parameter_region_size": "0x00004000",
  "ipc_address": "0x40220040",
  "cy_pub_key": {
    "crv": "P-256",
    "kid": "3",
    "kty": "EC",
    "x": "Zwavlq5SiAXmtCvjhy0g6Hl4h1d7KslRuiXvwAa7BnM",
    "y": "HJxqUKgpWAJYNrrpaU0fmQ6eU1rLadHrofZ-n_H2hos",
    "use": "sig",
    "kver": "B0-0"
  }
}
```

The following table explains the meaning of tokens in the entrance exam JSON object:

Token name	Token type	Description
`"type"`	string	Always `"ENTRANCE_EXAM"`
`"ahb_reads"`	Array of objects	The array of objects where each object represents one 32-bit-wide read operation. The objects consist of the following members: • `"address"` - address of the read to be performed (hex) • `"mask"` - mask to be ANDed with the value before checking the result (hex) • `"value"` - result expected from the read command (hex) • `"description"` - explains the meaning of the object
`"ahb_reads8"`	Array of objects	• The array of objects where each object represents one 8-bit-wide read operation. The objects consist of the following members: • `"address"` - address of the read to be performed (hex) • `"mask"` - mask to be ANDed with the value before checking the result (hex) • `"value"` - result expected from the read command (hex)

Token name	Token type	Description
		• `"description"` - explains the meaning of the object. For the PSoC™ CYB06447BZI device, the objects with description ASSET_HASH are required for an additional step in the entrance exam algorithm.
`"region_hashes"`	Array of objects	The array of objects where each object represents one call of RegionHash system calls. The objects consist of the following members: • `"address"` - start address of the region to pass to the system call (hex) • `"size"` - size of the address region to pass to the system call (hex) • `"value"` – hex value of the expected hash (MSB first, no prefix). • `"hash_id"` - ID of the hash algorithm to be used (a number, passed as parameter to the system call)
`"parameter_region"`	String	Address of the SRAM region to be used for system call parameters and results. A string that represents a hexadecimal number.
`"parameter_region_size"`	String	Size of the SRAM region to be used for system call parameters and results. A string that represents a hexadecimal number.
`"ipc_address"`	String	Address of the `IPC_STRUCT` structure for making system calls. A string that represents a hexadecimal number.
`"cy_pub_key"`	JWK object	The Infineon public key. This key must match the key used to sign the entrance exam JWT.

4.2 Authorization tokens for provisioning

The HSM is a separate entity that performs provisioning. Therefore, it must be trusted by both Infineon and the OEM. To establish trust, Infineon and the OEM issue JWT tokens that authorize the HSM identified by its public key. The HSM later includes the authorization tokens in provisioning tokens so that the device can verify that then provisioning is performed by the authorized entity.

4.2.1 Infineon HSM authorization

This token signifies that Infineon authorizes the HSM to provision Infineon devices. The token is signed with the Infineon private key and transferred to the HSM. It may contain information to limit the applicability of the token to specific device types or specific ranges of the die.

The following is an example of the Infineon HSM authorization JWT token payload:

```
{
  "type": "CY_AUTH_HSM",
  "auth": {
    "dev_id": {
      "include": [
        "E701.12.105",
        "E701.11.105"
      ],
      "exclude": [
        "E70D.12.105"
      ]
    },
    "die_id": {
      "max": {
        "day": 255,
        "lot": 16777215,
        "month": 255,
        "wafer": 255,
        "xpos": 255,
        "year": 255,
        "ypos": 255
      },
      "min": {
        "day": 0,
        "lot": 0,
        "month": 0,
        "wafer": 0,
        "xpos": 0,
        "year": 0,
        "ypos": 0
      }
    }
  },
  "cy_pub_key": {
    "crv": "P-256",
    "kid": "3",
    "kty": "EC",
    "x": "Zwavlq5SiAXmtCvjhy0g6Hl4h1d7KslRuiXvwAa7BnM",
    "y": "HJxqUKgpWAJYNrrpaU0fmQ6eU1rLadHrofZ-n_H2hos",
    "use": "sig",
    "kver": "B0-0"
  },
  "exp": 1618537730,
  "hsm_pub_key": {
    "crv": "P-256",
    "kid": "4",
    "kty": "EC",
    "use": "sig",
    "x": "sJMsN-2Jo27kc51wVK7xJ2fP9BDkzAc2fZEZMlohHXA",
    "y": "MWluzmXgXOvdQQDYX3yy1Tk9QoHL-9DZhswZpY0Xe5U"
  }
```

```
}
```

The following table explains the meaning of tokens in the Infineon HSM authorization JSON object.

Token name	Token type	Description
`"type"`	String	Always "CY_AUTH_HSM".
`"auth"`	Object	An authorization object that limits this token to specific device types or specific ranges of the die
`"cy_pub_key"`	JWK object	Infineon public key. This key must match the key used to sign the Infineon HSM authorization JWT.
`"hsm_pub_key"`	JWK object	Public key of the HSM that is authorized by this token
`"exp"`	Number	Expiration time represented as Unix time

4.2.1.1 Authorization object ("auth")

Consists of two members: **dev_id object** and **die_id object**.

- Both `dev_id` and `die_id` objects are optional members of the authorization object. If these objects are empty, the packet allows unlimited authorization (any device type, any die ID).
- If `dev_id` is present, either an include or exclude list must be present in that object. If both are present, the include list is used and the exclude list is ignored.
- If an include list is present, the `dev_id` of the device being provisioned must appear in that list. If an exclude list is present, the `dev_id` of the device being provisioned must not appear in that list.
- The `die_id` object may contain min/max constraints. A device being provisioned must comply with both the min and max constraints if both are given.
- The `“auth”` limits are checked as part of the `ProcessProvisionCmd` **system call**.

4.2.2 OEM Root-of-Trust authorization

This token signifies that the OEM authorizes the HSM to establish a new RoT based on the OEM public key. It is created by the OEM (signed with the OEM private key) and transferred to the HSM. The HSM includes this token with the provisioning tokens that are injected to the device. The OEM key from this token will be stored on the device as a base for RoT.

The following is an example of the OEM Root-of-Trust authorization JWT token payload:

```
{
  "type": "OEM_ROT_AUTH",
  "oem_pub_key": {
    "crv": "P-256",
    "kty": "EC",
    "use": "sig",
    "kid": "5",
    "x": "vfb7_jewTxpFVINcXdrZQJBArC5igrN0BLc783FigrM",
    "y": "9rBBUKXzpj1A5K7fxPtEaJdsfo7Jj_wsF7LTZLc-sPM"
  },
  "hsm_pub_key": {
    "crv": "P-256",
    "kty": "EC",
    "use": "sig",
    "kid": "4",
    "x": "sJMsN-2Jo27kc51wVK7xJ2fP9BDkzAc2fZEZMlohHXA",
    "y": "MWluzmXgXOvdQQDYX3yy1Tk9QoHL-9DZhswZpY0Xe5U"
  },
  "prod_id": "my_thing"
}
```

The following table explains the meaning of tokens in the OEM Root-of-Trust Authorization JSON object.

Token name	Token type	Description
`"type"`	String	Always `"OEM_ROT_AUTH"`
`"oem_pub_key"`	JWK object	OEM public key. This key must match the key used to sign the OEM Root-of-Trust Authorization JWT.
`"hsm_pub_key"`	JWK object	Public key of the HSM that is authorized by this token. The HSM public key in this token must match the HSM public key in the Infineon HSM authorization token.
`"prod_id"`	String	A product identifier of an arbitrary format with a maximum length of 64 bytes. It is given by the OEM to uniquely identify the product. The product identifier is stored on the device and becomes a part of the device identity.

4.3 Provisioning command

The provisioning command is a JWT with a number of claims in the payload that determine which provisioning step or steps are to be executed on the device as a result of receiving the token. The steps are:

1. Create the device identity
2. Transfer the RoT
3. Inject the Chain-of-Trust certificate
4. Inject the assets (policy, customer keys)

All steps may be performed at once or separately in any combination. If a step is performed separately, the provisioning command JWT must contain only necessary tokens for the step.

The token is created by an HSM and signed with its private key.

The OEM must provide the following objects contained within the provisioning command JWT:

1. RoT Auth (“`rot_auth`”)
2. All certificates contained in the Chain-of-Trust (“`chain_of_trust`”)
3. Provisioning request packet (“`prov_req`”)
4. The image certificate (“`image_cert`”)

Note: If the policy settings “`reprovision: boot_loader`” and “`reprovision: keys_and_policies`” are both “`false`”, then the provisioner must verify that the

“`SRAM_DAP`” “`address`” and “`size`” fields are configured correctly for the corresponding device. The required settings are as follows:

For 2M devices (CYS064xA, CYB064xA):

```
{
    "type": "SRAM_DAP",
    "address": 135192576,
    "size": 8192
}
```

For 1M or 512K devices (CYB064x7, CYB064x5):

```
{
    "type": "SRAM_DAP",
    "address": 134406144,
    "size": 8192
}
```

Note: *If the “`provision_group_private_key`” field is “true” in the policy, the OEM must also provide the “`prov_priv_key.jwt`” to the HSM. The HSM must verify the “`prov_priv_key`” JWT with the OEM public key and extract the “`grp_priv_key`” from the token. The HSM can then add the “`grp_priv_key`” to the final provisioning command JWT. The following is an example of the “`prov_priv_key.jwt`” contents:*

```
{
    "grp_priv_key": {
        "crv": "P-256",
        "d": "5fUZFiVR1dsMT4I3LQ2xkunieoFCmjveYJxn5XyxmQo",
        "kty": "EC",
        "use": "sig",
        "x": "9y0Ysj48RHKcZGg5gA25FzTgSfld49VIHV__CXZEgdc",
        "y": "K1AaVsV-MGCNE5787zEBhdlAlOK8XdOGZyp7jzBzZ1k"
    },
    "type": "OEM_GRP_PRIV_KEY"
}
```

The following table shows the possible payload tokens in the provisioning command JWT.

Token name	Token type	Step	Description
`"type"`	String	All	Always `"HSM_PROV_CMD"`
`"cy_auth"`	JWT string	All	Contains the Infineon HSM authorization JWT described above. Using this token, the device can verify that the HSM that executes the provisioning command is authorized by Infineon.

Token name	Token type	Step	Description
`"create_identity"`	Bool	Identity creation	If this token is present and set to `true`, creation of the device identity is initiated.
`"dev_priv_key"`	JWK object	Identity creation	During the creation of the device identity, its private key may be either generated locally in the device or passed using this token. This field is not typically present. The `"create_identity"` token must be present and set to `true` for this token to have an effect.
`"grp_priv_key"`	JWK object	Identity creation	If this token is present, the group private key is stored on the device as part of the identity creation. This key is shared between a group of devices. It can be used to derive the key to decrypt a bootloader image if an encrypted programming feature is used.
`"rot_auth"`	JWT string	RoT transfer	Contains the OEM Root-of-Trust authorization JWT described above. If this token is present, it initiates the RoT transfer step. During the step, the OEM and HSM public keys that are contained in the token will be stored on the device.
`"chain_of_trust"`	List of X.509 certificates	Certificate issue	Each element of the list is a string. It may contain any certificates needed on the device; for example, the device certificate for TLS or identity. No restrictions are placed on this field's contents and the chain-of-trust is considered an opaque object. If this token is present, the certificates will be stored on the device. The maximum size of the certificates is 5120 bytes.
`"prov_req"`	JWT string	Asset injection	The provisioning request is a JWT that contains the policy and customer keys in its payload. The OEM provides the provisioning request to the HSM. If this token is present, the provisioning request will be stored on the device.
`"image_cert"`	JWT string	Asset injection	The image certificate is a JWT that contains the bootloader metadata: start address, size, and hash. The OEM provides the image certificate to the HSM along with the bootloader image. If this token is present, the bootloader image stored in the device flash memory is validated against the certificate. If validation is successful, the image certificate will be stored on the device.
`"complete"`	Bool	Completion	If this token is present and set to `true`, the "Completion" provisioning step is performed, which means that device will enforce provisioned policies after the completion of this command, and attempt to securely boot upon next reset/power cycle.

The following is an example of the provisioning command JWT payload that contains all possible tokens (JWT strings are reduced for clarity):

```
{
    "type": "HSM_PROV_CMD",
    "cy_auth": "eyJhbGciOiJFUzI1NiJ9...",
    "create_identity": true,
    "dev_priv_key": {
        "crv": "P-256",
        "d": "iaHQORcqDLvsl-h5QKaLIeJwIgmJFwmdgM0FKaG1p94",
        "kty": "EC",
        "use": "sig",
        "x": "N1DhaEYSZ8kGdqTN3mJnog5naNpIFsJzJk-UTC3leZA",
        "y": "aBeESN31DuaGSMiBGOVbTAYKD2CobxF5MY7aPK_Brbc"
    },
    "grp_priv_key": {
        "crv": "P-256",
        "d": "5fUZFiVR1dsMT4I3LQ2xkunieoFCmjveYJxn5XyxmQo",
        "kty": "EC",
        "use": "sig",
        "x": "9y0Ysj48RHKcZGg5gA25FzTgSfld49VIHV__CXZEgdc",
        "y": "K1AaVsV-MGCNE5787zEBhdlAlOK8XdOGZyp7jzBzZ1k"
    },
    "rot_auth": "eyJhbGciOiJFUzI1NiIs...",
    "chain_of_trust": [
        "-----BEGIN CERTIFICATE-----\nMIIBNTCB2aADAgECAhQB...\n-----END CERTIFICATE--
---\n",
        "-----BEGIN CERTIFICATE-----\nMIIBNTCB2aADAgECAhQB...\n-----END CERTIFICATE--
---\n"
    ],
    "prov_req": "eyJhbGciOiJFUzI1NiJ9...",
    "image_cert": "eyJhbGciOiJFUzI1NiJ9...",
    "complete": true
}
```

4.3.1 Device identity

The following steps are involved in creating the device identity:

1. Generate a unique device secret (UDS) and store it in the device eFuses.
2. Generate the device private key if it is not provided with the "`dev_priv_key`" token.
3. Receive the group private key if it is provided with the "`grp_priv_key`" token.
4. Encrypt the device private key and the group private key (if provided) with a key derived from UDS and store the encrypted data in the device.

The device public key and the group public key are made public to anyone inquiring about the identity of the device. The private keys are kept in the device encrypted and never exposed. They are used only to sign the attestation tokens about the device identity.

It may be useful to initiate the step separately before other provisioning steps, because as a result, the device public key will be generated, which can be included in the chain-of-trust certificate for the following step.

To initiate the device identity creation step separately, the provisioning command payload should include only the following tokens:

- `"create_identity"`: true
- `"cy_auth"` with the Infineon HSM authorization JWT
- `"dev_priv_key"` (optional)
- `"grp_priv_key"` (optional)

Once this step is issued, the device identity becomes immutable.

4.3.2 Image certificate

The image certificate is a JWT that contains the information required to validate the bootloader image. It can also contain additional information about the bootloader.

The trust of the bootloader is based on the fact that the hash of the bootloader image matches with the hash provided in the certificate signed by the bootloader provider.

The following is an example of the image certificate token payload:

```
{
  "image_id": 0,
  "image_hash": [
    65, 125, 251, 58, 59, 3, 208, 207, 229, 239, 123, 49, 141, 21, 126, 119,
    231, 240, 170, 230, 226, 148, 161, 165, 5, 31, 171, 62, 11, 13, 121, 205
  ],
  "image_file": "InfineonBootloader_CM0p.hex",
  "image_address": 270336000,
  "image_size": 65224,
  "image_version": "2.0.0.3257",
  "iat": 1605061545,
  "exp": 1924984800
}
```

The following table explains the meaning of tokens in the image certificate JSON object.

Token name	Token type	Mandatory	Description
`"image_hash"`	List of numbers	Yes	List of numbers where each number represents one byte of the bootloader image hash. The hash is calculated using the SHA-256 algorithm. Therefore, the list consists of 32 numbers.
`"image_address"`	Number	Yes	Address in the device memory where the bootloader is located
`"image_size"`	Number	Yes	Size of the image in bytes
`"image_id"`	Number	No	ID of the image; for bootloader, it is always 0
`"image_file"`	String	No	Name of the bootloader image file

Token name	Token type	Mandatory	Description
`"iat"`	Number	No	Issue time represented as Unix time
`"exp"`	Number	No	Expiration time represented as Unix time. The HSM can check this field to verify the validity of the bootloader certificate.

The bootloader image hash is calculated for a memory range started from `"image_address"` to `"image_address" + "image_size"`.

The image certificate must be signed with the key defined in the policy in the `"firmware"` list item 0 (`"id" : 0`). The following is a fragment of the policy that defines that a key with ID 5 is to be used for the bootloader image certificate validation (see **Keys format** and **Policy format** sections):

```
{
       "boot_upgrade": {
        "title": "upgrade_policy",
        "firmware": [
            {
                "boot_auth": [
                    5
                ],
                "id": 0,
                            ...
                     },
                     ...
              ]
       }
}
```

Because the validation of the image certificate token relies on the policy, the provisioning command must contain both `"image_cert"` and `"prov_req"` tokens.

4.3.3 Provisioning request

The provisioning request is a JWT provided by the OEM to the HSM. It contains the policies and a list of customer keys to be installed on the device. The provisioning request JWT is signed by the OEM private key.

The following is an example of a payload of the provisioning request JWT (policies are excluded for clarity because their format is described in the **Policy format** section.):

```
{
  "debug": {
    ...
  },
  "boot_upgrade": {
    ...
  },
  "prod_id": "my_thing",
  "custom_pub_key": [
    {
      "crv": "P-256",
      "kid": "6",
      "kty": "EC",
      "use": "sig",
      "x": "ouTC_gXC7-rzcLn2IJsdQuokCQNeF6-5Mugq5PPpwgw",
      "y": "SZ04p3z_jyilL8J18zZoxGTcWDoL7XPSBsJ5EZx9MAo"
    },
    {
      "crv": "P-256",
      "kty": "EC",
      "use": "sig",
      "x": "_za6DQEnUxqOm0vK9Pgvt9GHBtFi1XIVrPvfQ5zq90k",
      "y": "mhqW_r-kI0hWvAW_cqQmyaxlRs02bF4w-v4iV8YY-DQ",
      "kid": "8"
    }
  ]
}
```

The following table explains the meaning of tokens in the provisioning request JSON object.

Token name	Token type	Description
`"debug"`	Object	Debug Policy JSON object; see the **Policy format** section.
`"boot_upgrade"`	Object	Boot & Upgrade Policy JSON object; see the **Policy format** section.
`"prod_id"`	String	A product identifier of an arbitrary format given by the OEM to uniquely identify the product. Must match the one from the OEM Root-of-Trust Authorization JWT.
`"custom_pub_key"`	List of JWK objects	A list of customer public keys which can be used for application validation during secured boot or any other purpose. The list can contain up to five keys with IDs in a range [6...10].

4.3.4 Device response

The device returns a response as a result of receiving the provisioning command JWT. The device response is also a JWT, which is signed using the device private key. If no device private key yet exists (the device identity creation step has not been executed), the response JWT has truncated format: only the header and payload are present; the signature is missing.

The device response JWT payload tokens are explained in the following table.

Token name	Token type	Description
`"type"`	String	Always `"DEV_RSP"`
`"dev_id"`	String	Device ID; allows identity of a part number. It is injected during Infineon manufacturing.

Token name	Token type	Description
`"die_id"`	Object	Unique die ID (lot/wafer/xpos/ypos/day/month/year) created by Infineon during manufacturing
`"dev_pub_key"`	JWK object	Device public key generated or injected during the identity creation step. Not present if the identity has not yet been created.
`"grp_pub_key"`	JWK object	Group public key injected during the identity creation step. Not present if the identity has not yet been created.
`"oem_pub_key"`	JWK object	The OEM public key, injected during the RoT transfer step. Not present if the RoT transfer has not yet happened.
`"prod_id"`	String	A product identifier defined by the OEM to uniquely identify the product, injected during the RoT transfer step. Not present if the RoT transfer has not yet happened.
`"complete"`	Bool	Set to `true` if the Completion step was executed. Provisioned policies are in effect.

If issuing of the provisioning command is divided into several steps, the device response includes only the tokens that are already present in the device. For example, if the RoT transfer step has not been executed, the device response will not contain the OEM public key.

The following is an example of the payload of the complete device response JWT:

```
{
  "type": "DEV_RSP",
  "dev_id": "silicon_id=E70D.12, family_id=105",
  "die_id": {
    "lot": 9001031,
    "wafer": 23,
    "xpos": 27,
    "ypos": 41,
    "day": 21,
    "month": 2,
    "year": 120
  },
 "dev_pub_key": {
    "crv": "P-256",
    "kty": "EC",
    "use": "sig",
    "kid": "2",
    "x": "sko3TUSOAOg8dYgGa-9jArQ3id5OGIdkKPZ5G-oIrTo",
    "y": "5LfVGpWCCq1-EG9W9yMqPpj0uHExk_17U1RNV-c8NOU"
  },
  "grp_pub_key": {
    "crv": "P-256",
    "kty": "EC",
    "use": "sig",
    "kid": "12",
    "x": "9y0Ysj48RHKcZGg5gA25FzTgSfld49VIHV__CXZEgdc",
    "y": "K1AaVsV-MGCNE5787zEBhdlAlOK8XdOGZyp7jzBzZ1k"
  },
  "oem_pub_key": {
    "crv": "P-256",
    "kid": "5",
    "kty": "EC",
    "use": "sig",
    "x": "vfb7_jewTxpFVINcXdrZQJBArC5igrN0BLc783FigrM",
    "y": "9rBBUKXzpj1A5K7fxPtEaJdsfo7Jj_wsF7LTZLc-sPM"
  },
  "prod_id": "my_thing",
  "complete": true
}
```

5 Provisioning algorithm

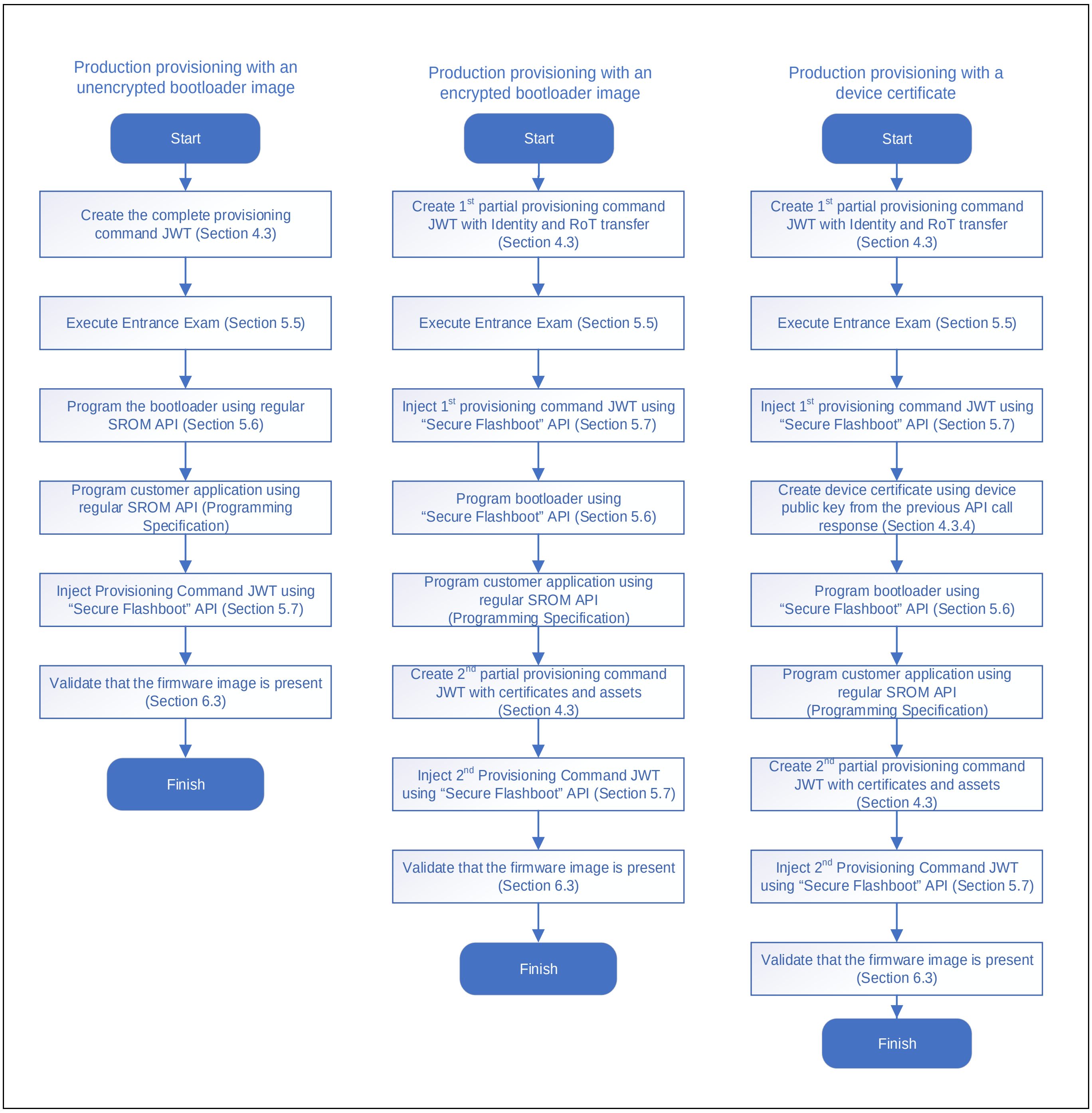
This chapter describes the provisioning algorithm of PSoC™ 64 devices.

5.1 High-level provisioning algorithm

The following diagrams show the sequence of steps that must be performed to provision the PSoC™ 64 device. Some steps are different depending on whether the bootloader image is distributed as encrypted or unencrypted (as plaintext), and whether a device certificate is to be provisioned. The following cases are shown:

1. Production provisioning with an unencrypted bootloader image
2. Production provisioning with an encrypted bootloader image
3. Production provisioning with a device certificate

The following steps involve communication with the device through the Program and Debug Interface and execution of the SROM and "Secure Flashboot" APIs. A description of the communication through the Program and Debug Interface and execution of the SROM APIs is out of the scope of this document. See the **PSoC™ 64 Programming Specifications**. The "Secure Flashboot" APIs are described in the following sections.


Production provisioning with an unencrypted bootloader image
Start
Create the complete provisioning command JWT (Section 4.3)
Execute Entrance Exam (Section 5.5)
Program the bootloader using regular SROM API (Section 5.6)
Program customer application using regular SROM API (Programming Specification)
Inject Provisioning Command JWT using "Secure Flashboot" API (Section 5.7)
Validate that the firmware image is present (Section 6.3)
Finish
Production provisioning with an encrypted bootloader image
Start
Create 1st partial provisioning command JWT with Identity and RoT transfer (Section 4.3)
Execute Entrance Exam (Section 5.5)
Inject 1st provisioning command JWT using "Secure Flashboot" API (Section 5.7)
Program bootloader using "Secure Flashboot" API (Section 5.6)
Program customer application using regular SROM API (Programming Specification)
Create 2nd partial provisioning command JWT with certificates and assets (Section 4.3)
Inject 2nd Provisioning Command JWT using "Secure Flashboot" API (Section 5.7)
Validate that the firmware image is present (Section 6.3)
Finish
Production provisioning with a device certificate
Start
Create 1st partial provisioning command JWT with Identity and RoT transfer (Section 4.3)
Execute Entrance Exam (Section 5.5)
Inject 1st provisioning command JWT using "Secure Flashboot" API (Section 5.7)
Create device certificate using device public key from the previous API call response (Section 4.3.4)
Program bootloader using "Secure Flashboot" API (Section 5.6)
Program customer application using regular SROM API (Programming Specification)
Create 2nd partial provisioning command JWT with certificates and assets (Section 4.3)
Inject 2nd Provisioning Command JWT using "Secure Flashboot" API (Section 5.7)
Validate that the firmware image is present (Section 6.3)
Finish

The main difference between the cases is that the provisioning command JWT is created and injected in one step or split into two parts and injected in two steps.

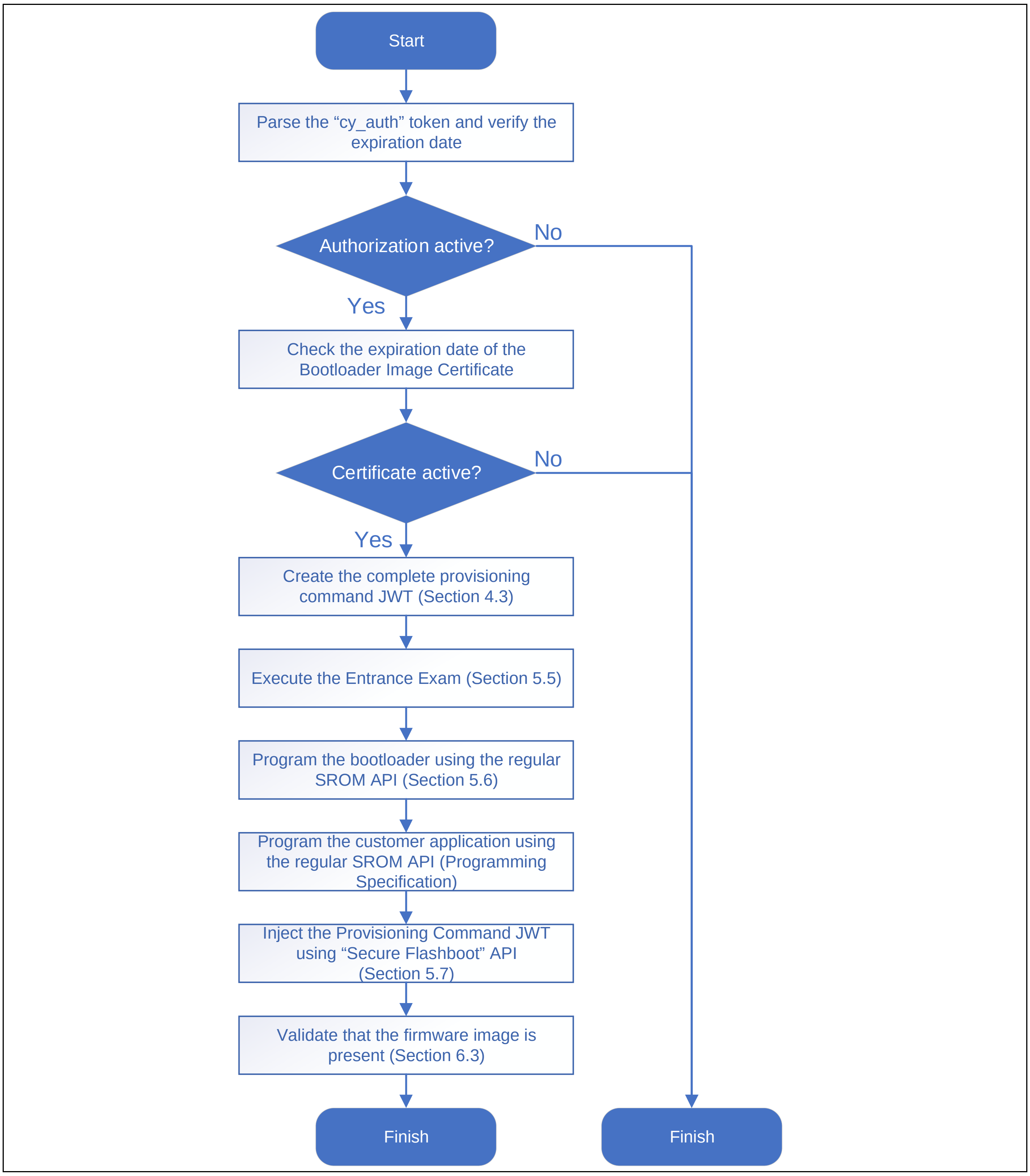
In the first case (unencrypted bootloader image), the provisioning command JWT is created containing all the tokens described in the **Provisioning command** section. The `"complete"` token should be set to `true` to finalize the provisioning process.

In the second case (encrypted bootloader image), the first provisioning command JWT is created containing only the tokens required to create the device identity and transfer RoT; the `"complete"` token absent or set to `false`. Injecting this provisioning command stores the device private key, group private key (optional), and OEM and HSM public keys in the device. These keys are required for the encrypted programming API operation.

After the encrypted bootloader image programming, the second provisioning command JWT (contains the remaining tokens and the `"complete"` token set to `true`) is injected to finalize the provisioning process.

The third case is suitable for both encrypted or unencrypted bootloader images. The reason for splitting the provisioning command JWT is that the device certificate must be created and provisioned into the device. To create the device certificate, the device public key is required. The device public key is created during the identity creation step and is returned in the device response to the provisioning command injection. The contents of the both provisioning command JWTs is the same as for the second case, except for the `"chain_of_trust"` token that contains the device certificate.

5.2 Production provisioning with an unencrypted bootloader


Start
Parse the "cy_auth" token and verify the expiration date
Authorization active?
No
Yes
Check the expiration date of the Bootloader Image Certificate
Certificate active?
No
Yes
Create the complete provisioning command JWT (Section 4.3)
Execute the Entrance Exam (Section 5.5)
Program the bootloader using the regular SROM API (Section 5.6)
Program the customer application using the regular SROM API (Programming Specification)
Inject the Provisioning Command JWT using "Secure Flashboot" API (Section 5.7)
Validate that the firmware image is present (Section 6.3)
Finish
Finish

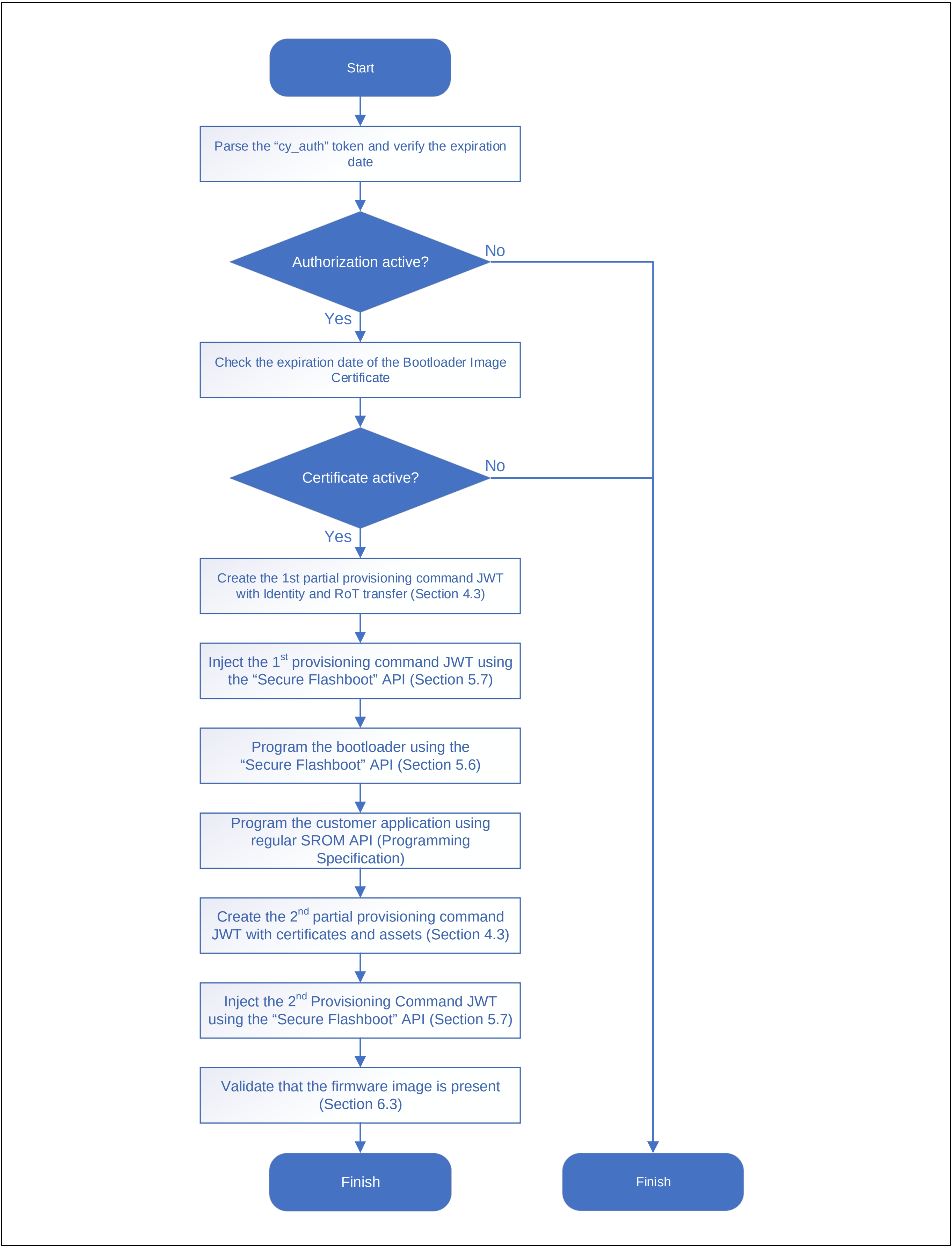
1. Confirm that the "`cy_auth`" token expiration date is in the future. If it is, proceed with the provisioning flow. If it is not, do not proceed.
2. Check the "exp" field in the **bootloader image certificate**. If the expiration date is in the future or if the field is not present, proceed with the provisioning flow. If the certificate is expired, do not proceed.

3. Create a provisioning command JWT. The following objects must be created and filled in to the provisioning command token. Additional fields contained in the **provisioning command JWT** can optionally be filled in based on the format present in the **Provisioning command** section.

Object name	Object type	Description/value
`“type”`	String	Always `“HSM_PROV_CMD”`
`“cy_auth”`	JWT string	Contains the Infineon HSM authorization JWT
`“create_identity”`	Bool	`“true”`
`“rot_auth”`	JWT string	OEM Root-of-Trust authorization token
`“prov_req”`	JWT string	Provisioning request JWT
`“complete”`	Bool	`“true”`

4. Run an entrance exam to verify the integrity of the device. See the **Entrance exam** section for an example flow for running the entrance exam.
5. Program the bootloader. Follow the steps listed in the **PSoC™ 64 Programming Specifications**.
6. Program the application. Follow the steps listed in the **PSoC™ 64 Programming Specifications**.
7. Program the provisioning command JWT using the `ProcessProvisionCmd` system call. See the **System Calls API Reference** section for more information.
8. Verify that the application is present and launches on device reset. See the **Firmware Presence Status** section for more information.

5.3 Production provisioning with an encrypted bootloader


Start
Parse the "cy_auth" token and verify the expiration date
Authorization active?
No
Yes
Check the expiration date of the Bootloader Image Certificate
Certificate active?
No
Yes
Create the 1st partial provisioning command JWT with Identity and RoT transfer (Section 4.3)
Inject the 1st provisioning command JWT using the "Secure Flashboot" API (Section 5.7)
Program the bootloader using the "Secure Flashboot" API (Section 5.6)
Program the customer application using regular SROM API (Programming Specification)
Create the 2nd partial provisioning command JWT with certificates and assets (Section 4.3)
Inject the 2nd Provisioning Command JWT using the "Secure Flashboot" API (Section 5.7)
Validate that the firmware image is present (Section 6.3)
Finish
Finish

Provisioning algorithm

1. Confirm that the "cy_auth" token expiration date is in the future. If it is, proceed with the provisioning flow. If it is not, do not proceed.
2. Check the "exp" field in the **bootloader image certificate**. If the expiration date is in the future or if the field is not present, proceed with the provisioning flow. If the certificate is expired, do not proceed.
3. Create a provisioning command JWT containing only the following objects:

Object Name	Object Type	Description/Value
"type"	String	Always **"HSM_PROV_CMD"**
"cy_auth"	JWT string	Contains the Infineon HSM authorization JWT
"create_identity"	Bool	"true"
"dev_priv_key" (optional)	JWK	Device private key. If empty and the "create_identity" field is "true", the device generates this value internally. Passing a private key value in this field sets the device private key to the passed value.
"grp_priv_key" (optional)	JWK	A private key shared between a group of devices. Can be used for decryption of an encrypted bootloader image or key derivation.
"complete"	Bool	"false"

4. Run an entrance exam to verify the integrity of the device. See the **Entrance exam** section for an example flow for running the entrance exam.
5. Program the device identity
   a. Program the provisioning command JWT using the ProcessProvisionCmd system call. This transfers the RoT and create the device public key.
   b. Store the returned device response JWT. This is returned from the ProcessProvisionCmd system call at SRAM_SCRATCH2 + 0x04.
6. Program the encrypted bootloader image using the "Secure Flashboot" API. Pseudocode in the **Bootloader image programming** section demonstrates how this can be done.
7. Program the application image by following the steps in the **PSoC™ 64 Programming Specifications**.
8. Create the 2nd provisioning packet containing the remaining objects and setting the "complete" field to "true".
9. Program the complete provisioning command JWT using the ProcessProvisionCmd system call.
10. Verify that the application is present and launches on device reset. See the **Firmware Presence Status** section for more information.

5.4 System call execution

Many operations required for the device provisioning are implemented as system calls. These system calls are executed out of "Secure Flashboot"; they are similar to the SROM APIs described in the **Technical Reference Manuals** and **PSoC™ 64 MCU Programming Specifications**. System calls are executed inside the CM0+ Non-Maskable Interrupt (NMI) in Protection Context 0. The system call interface makes use of the IPC to initiate an NMI to CM0+.

System calls can be performed by CM0+, CM4, or the Debug Access Port (DAP). Each has a reserved IPC structure through which they can request the execution of a system call. Different system calls are identified by their opcodes (8-bit number). The caller acquires the specific IPC structure, writes the opcode and argument to the data register of the IPC structure, and notifies a dedicated IPC interrupt structure. This results in an NMI interrupt in CM0+ where the system call is executed. When the system call execution is finished, the IPC structure is released; the caller can read a return code and output data (if provided by the system call) in the data register of the IPC structure.

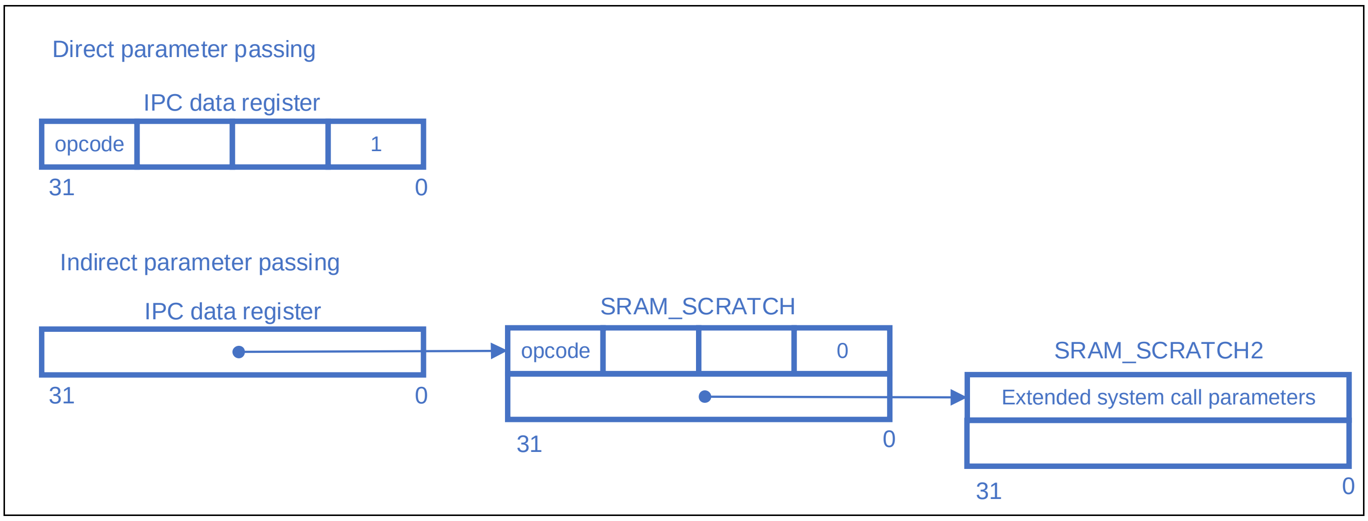
External programmers request system call execution through the DAP; the IPC structure #2 is reserved for this purpose. The IPC structure address and SRAM region that should be used to pass parameters to system calls through the DAP are available in the entrance exam JWT provided by Infineon.

Note that the size of the SRAM region for system call parameters given in the entrance exam JWT is for unprovisioned devices. After the completion of the provisioning process, the size of the region reduces by half; the lower addresses of the region became unavailable.

For example, the usual size for unprovisioned devices is 16 KB; for the provisioned devices, it is 8 KB. The region 0x080EC000-0x080F0000 is available for the parameters before provisioning is completed and the region 0x080EE000-0x080F0000 is available after provisioning completion.

Only a single 32-bit argument can be passed through a data register of the IPC structure. This argument is either a pointer to the SRAM or a formatted opcode and/or argument values that cannot be a valid SRAM address. To indicate that all parameters are passed through the data register, the least significant bit of the data field is set to '1' because valid SRAM addresses must be aligned on a word boundary.

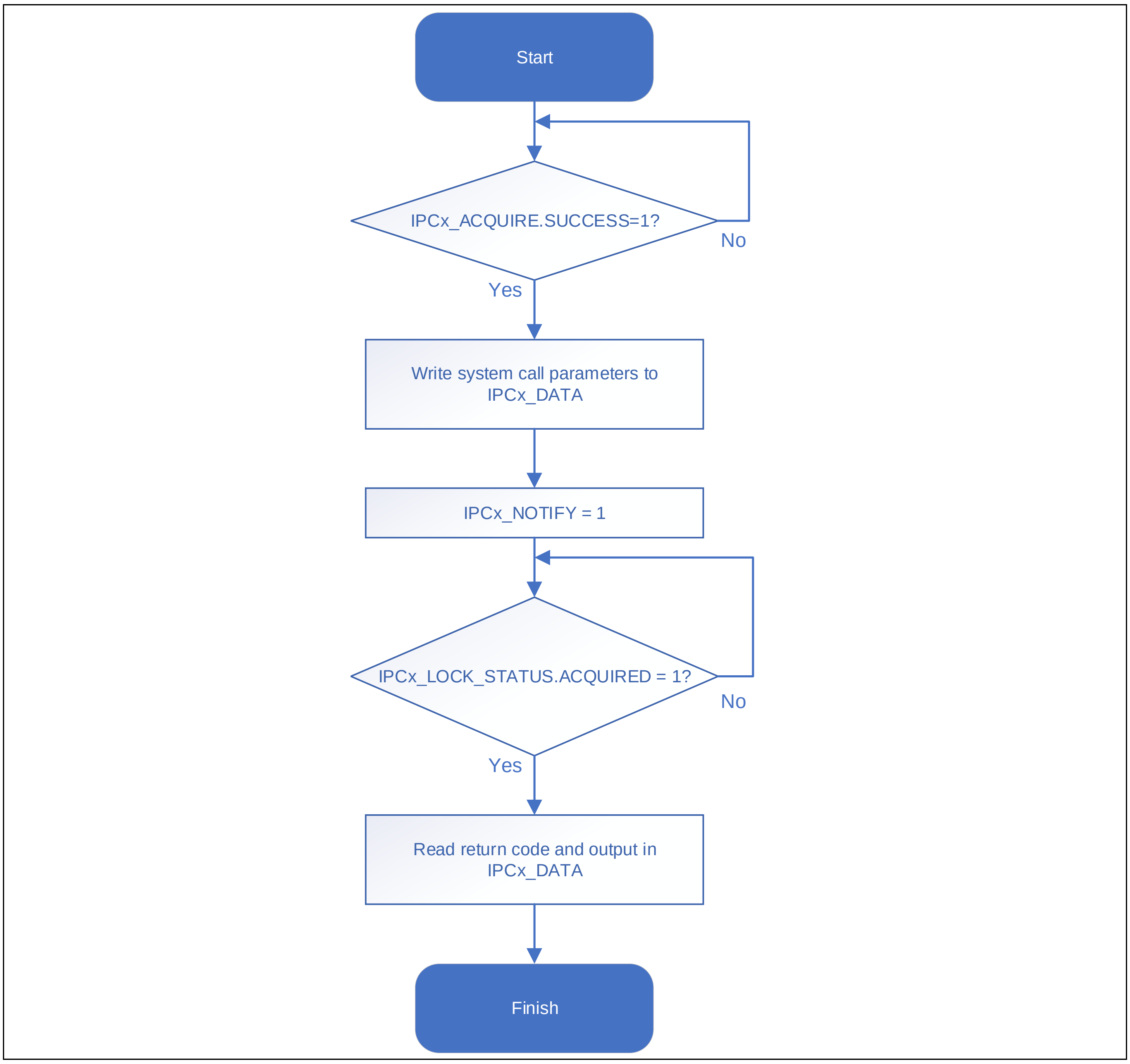
The following diagram shows two possible ways of passing the parameters to "Secure Flashboot" system calls. SRAM_SCRATCH and SRAM_SCRATCH2 are the SRAM regions where the parameters for the system call are stored.



The system call return code is a 32-bit value. The return code and output data are passed in a similar way as input parameters. The data register of the IPC structure can contain either the return code or a pointer to the SRAM where the return code and output data are stored (if they are provided by the system call).

The way the input parameters and the return code are passed depends on the specific system call and it is described in the **System Calls API** reference section.

The following diagram illustrates the system call execution request algorithm:


Start
IPCx_ACQUIRE.SUCCESS=1?
No
Yes
Write system call parameters to IPCx_DATA
IPCx_NOTIFY = 1
IPCx_LOCK_STATUS.ACQUIRED = 1?
No
Yes
Read return code and output in IPCx_DATA
Finish

5.5 Entrance exam

The entrance exam procedure consists of the following steps:

1. Decode and validate the entrance exam JWT.
2. Acquire the device.
3. Iterate through all items in the `"ahb_reads8"` list. The pass condition for each item is `Read8(address) & mask == value`.
4. Iterate through all items in the `"ahb_reads"` list. The pass condition for each item is `Read32(address) & mask == value`.
5. Iterate through all items in the `"region_hashes"` list. Execute the RegionHash system call for each item in the list. The pass condition for each item is `RegionHash(value, hash_id, address, size) == SUCCESS`.

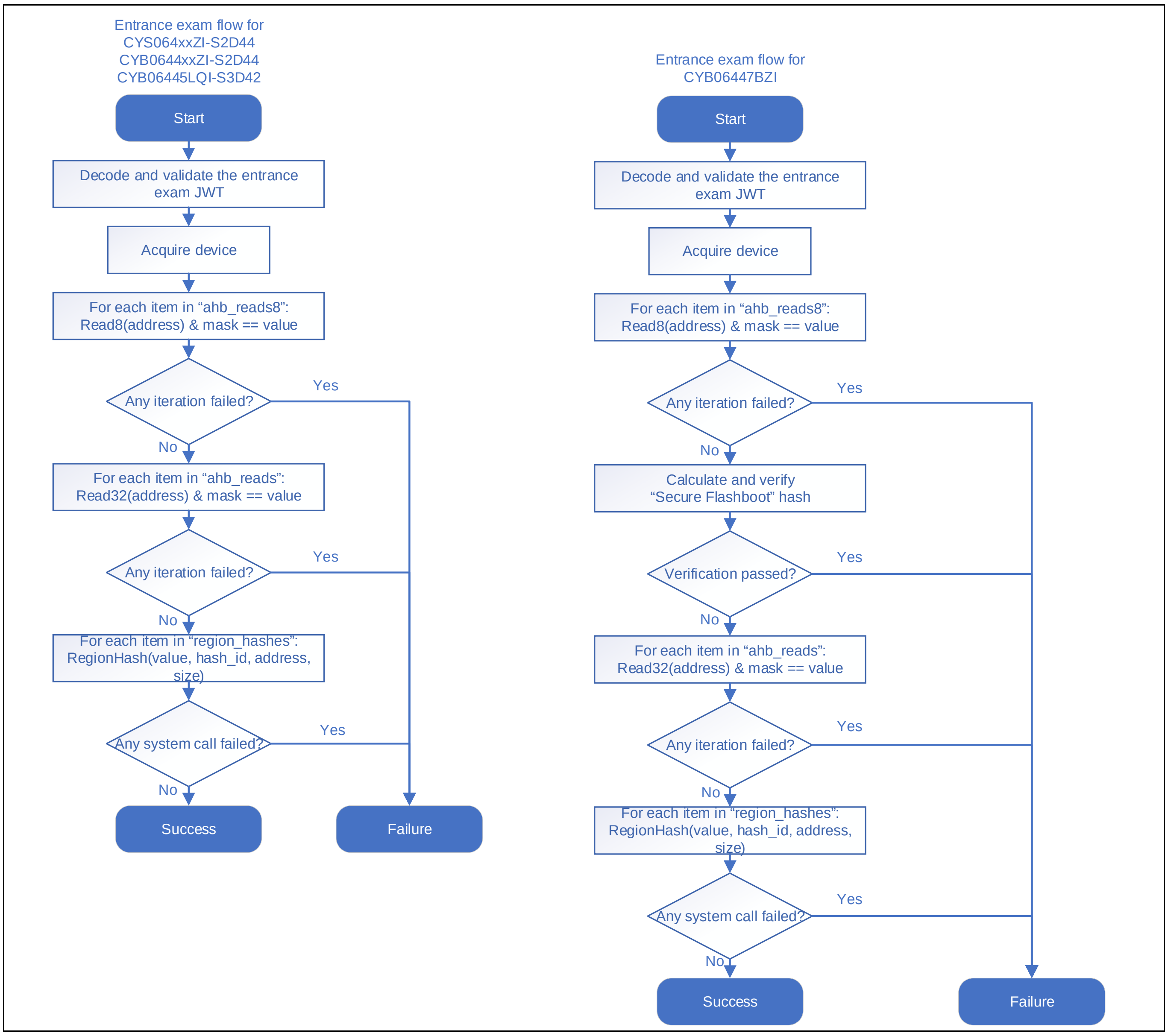
Provisioning algorithm

It is important to perform read operations first because they allow you to verify the integrity of the boot code. After that, you can trust the boot code to execute system calls.

Use `"ipc_address"`, `"parameter_region"`, and `"parameter_region_size"` tokens from the entrance exam JWT to execute the system calls.

Failure at any step leads to the rejection of the device.

The following diagram illustrates the entrance exam algorithm:


Entrance exam flow for CYS064xxZI-S2D44 CYB0644xxZI-S2D44 CYB06445LQI-S3D42
Start
Decode and validate the entrance exam JWT
Acquire device
For each item in "ahb_reads8": Read8(address) & mask == value
Any iteration failed?
Yes
No
For each item in "ahb_reads": Read32(address) & mask == value
Any iteration failed?
Yes
No
For each item in "region_hashes": RegionHash(value, hash_id, address, size)
Any system call failed?
Yes
No
Success
Failure
Entrance exam flow for CYB06447BZI
Start
Decode and validate the entrance exam JWT
Acquire device
For each item in "ahb_reads8": Read8(address) & mask == value
Any iteration failed?
Yes
No
Calculate and verify "Secure Flashboot" hash
Verification passed?
Yes
No
For each item in "ahb_reads": Read32(address) & mask == value
Any iteration failed?
Yes
No
For each item in "region_hashes": RegionHash(value, hash_id, address, size)
Any system call failed?
Yes
No
Success
Failure

The algorithm implemented by the programmer tool might look like this pseudocode:

```
// Connect to the device system access port
device = programmer.connect(SYSAP)

// Read the Infineon public key from the device
cy_public_key = device.read_public_key()
```

```
// Decode and validate JWT signature with Infineon public key
jwt_plaintext = parse_jwt(entrance_exam.jwt)
status = validate_jwt(jwt_plaintext, cy_public_key)

// Convert jwt to readable json
json_data = convert_to_data(jwt_plaintext)

// Verify all items in ahb_reads8 and ahb_reads32 list
if status == pass:
    for item in json_data.ahb_reads8:
        if(json_data.ahb_reads8.value ==
            ahb_reads8[json_data.ahb_reads8.address] &
                      json_data.ahb_reads8.mask):
                  status == pass
         else:
            status == fail
if status == pass:
    for item in json_data.ahb_reads32:
        if(json_data.ahb_reads32.value ==
            ahb_reads32[json_data.ahb_reads32.address] & json_data.ahb_reads8.mask):
                  status == pass
         else:
              status == fail

// Verify all items in the region_hashes list with RegionHash system call
if status == pass:
       for item in json_data.region_hashes:
              status = region_hash(compare_value, hash_id, address, size)
if status == pass:
return SUCCESS
else:
       return FAILURE
```

5.5.1 Entrance exam for PSoC™ CYB06447BZI

The entrance exam algorithm for the CYB06447BZI devices contain an additional step compared to other devices. The “Secure Flashboot” hash is verified by the SROM code for CYS064xxZI-S2D44, CYB0644xxZI-S2D44, and CYB06445LQI-S3D42 devices, but CYB06447BZI devices do not have this feature. Therefore, the entrance exam procedure for CYB06447BZI includes the calculation and verification of the “Secure Flashboot” code.

The calculated hash is verified against the value provided in the entrance exam JWT. The hash byte values are provided in the `"ahb_reads8"` array in objects with a description `"ASSET_HASH byte X"`, where X is a byte number. The hash algorithm is SHA256, for which the digest is truncated to 128 bits (16 bytes). Therefore, values of X are in the range 0…15.

The “Secure Flashboot” code is placed in multiple locations. The complete list of locations is available in TOC1 (Table of contents). The address of the TOC1 is 0x16007800. The following table describes the TOC1 contents for the CYB06447BZI device (values that are required for the hash calculation are in bold):

Offset	Value example	Purpose
0x00	0x200-4	TOC1 object size in bytes starting from offset 0x00 until the last entry in TOC
0x04	0x01211219	Magic Number (TOC Part 1, ID)
0x08	**10**	Number of objects starting from offset 0xC
0x0C	0x16000200u	Trim Table address
0x10	0x16000600u	DIE ID address
0x14	**0x16002000u**	**Flashboot code in Supervisory Flash with object size, which includes the Infineon Pub Key region**
0x18	**0x16000004u**	**System call table address. The length is constant - 4 bytes.**
0x1C	**0x100E0000u**	**Flashboot main code in flash with the object size**
0x20	**0x1605a000u**	**Infineon Public Key object**
0x24	**0x16007800u**	**TOC1 object address**
0x28	**0x16007A00u**	**Redundant TOC1 object address**
0x2C	**0x16007C00u**	**TOC2 object address**
0x30	**0x16007E00u**	**Redundant TOC2 object address**
0x200-4	0xXXXXXXXX	CRC is calculated and inserted by the CYELF tool run with the `--sign` option

Locations that are included in the hash calculation start at offset 0x14. The number of objects can vary; it is calculated as the value given at offset 0x08 minus 2. For the example above, it is 8.

At each offset, the address of the code location is given. The first word at the address contains the region size in bytes. The only exception is the offset 0x18 (system call table location). Its size is fixed to four bytes; therefore, the address itself is included in the hash calculation. The following pseudocode explains the hash calculation algorithm:

```
For each offset in TOC1:
    if offset is SysCall table:
        address = TOC1[offset]
        hash = SHA256(address, 4)
    else:
        address = TOC1[offset]
        size = address[1]
        hash = SHA256(address, size)
```

5.6 Bootloader image programming

The bootloader provider includes a bootloader **image certificate** that can optionally contain an “exp” field that specifies the bootloader image expiration time. Before programming the bootloader image, the programmer or HSM should confirm that the bootloader image is still valid by checking the expiration date. If the field is left empty or is not included at all, the bootloader image is assumed to be valid.

If the bootloader image is programmed unencrypted, use the regular flash memory programming procedures described in **PSoC™ 64 MCU Programming Specifications**.

This section describes the encrypted flash memory programming procedure.

The image data is transferred in encrypted form and is decrypted in the device. Application developers can use this process to protect their firmware from theft in the supply chain.

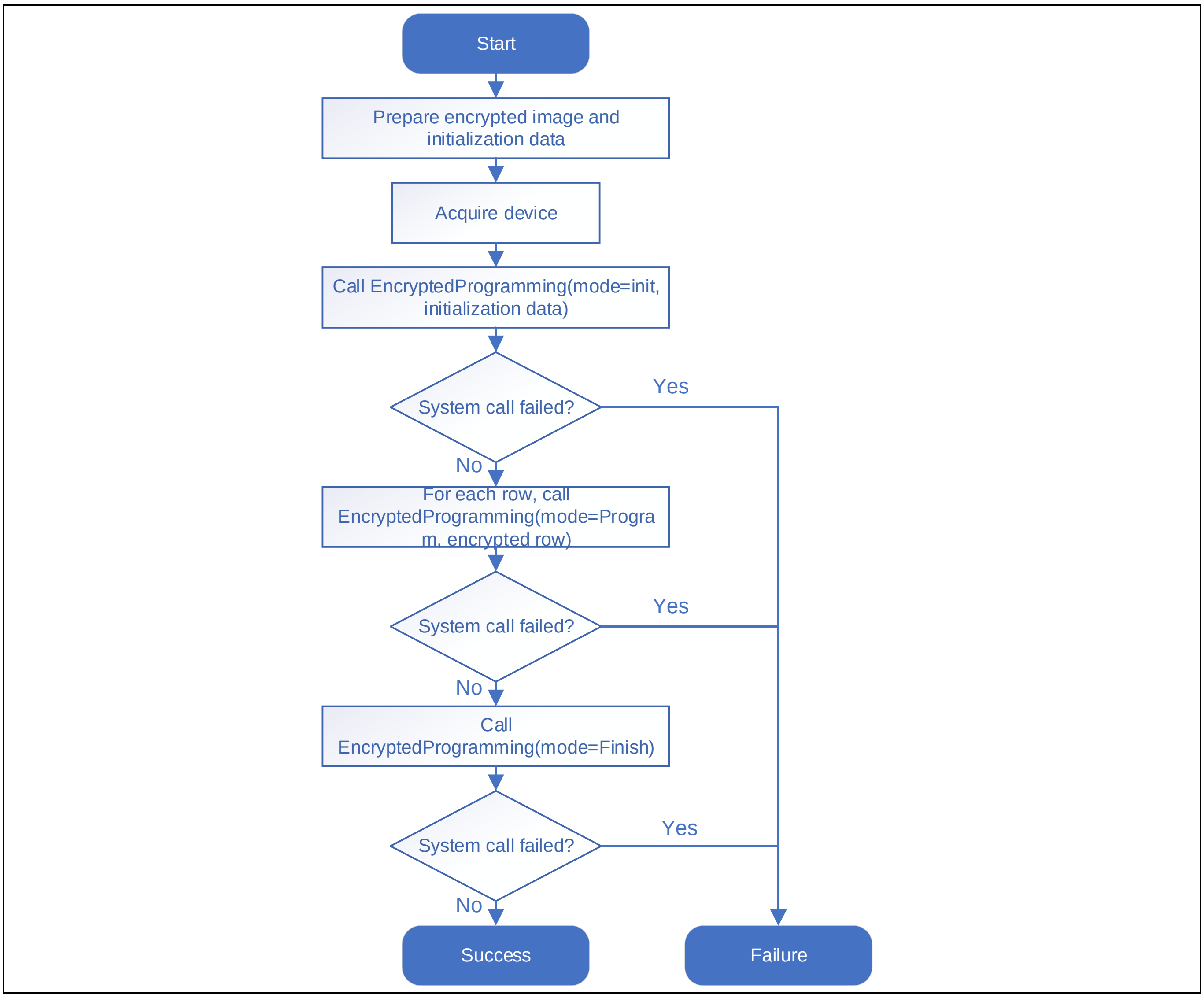
Note: ***Limitation:*** *The encrypted programming process is intended to protect the confidentiality of the binary images in transit. However, these images are decrypted inside the device and reside there in plain text. The decrypted image can be protected by disabling the debug ports.*

For encrypted programming, the AES CBC algorithm with a key size of 128 or 256 bits and PKCS#7 padding is used to encrypt the image. The encrypted programming feature is achieved by using the `EncryptedProgramming` system call to program the flash memory instead of the `WriteRow` system call for the conventional flash memory programming. The `EncryptedProgramming` system call programs the flash memory in rows and is called sequentially for each row. A row of the encrypted image is an independently encrypted block of 512 bytes of a plaintext row. The encrypted row length is 528 bytes (512 bytes of data and 16 bytes of padding required for PKCS#7).

The encrypted programming process consists of the following steps:

1. Prepare the encrypted bootloader image and initialization data (receive from an image developer or encrypt by themselves). Follow the steps listed in the **Image encryption algorithm** section.
2. Acquire the device.
3. Call `EncryptedProgramming(mode=Init, initialization data)` to initialize the process.
4. Call `EncryptedProgramming(mode=Program, encrypted row)` for each row in the encrypted image.
5. Call `EncryptedProgramming(mode=Finish)` to finish the process.

The following diagram illustrates the encrypted programming algorithm:


Start
Prepare encrypted image and initialization data
Acquire device
Call EncryptedProgramming(mode=init, initialization data)
System call failed?
Yes
No
For each row, call EncryptedProgramming(mode=Program, encrypted row)
System call failed?
Yes
No
Call EncryptedProgramming(mode=Finish)
System call failed?
Yes
No
Success
Failure

The following pseudocode demonstrates this flow:

```
// Prepare the encrypted assets, key K, init vector IV, shared secret
// encryptedSecret, and encrypted init data encrypted_header
K, IV = GenerateKey(AES-128)
K_shared, IV_shared = ECDH(SHA256 mode, host_priv_key, device_pub_key, salt, info)

struct aes_data_t aes_data = {K, IV, padding}

encryptedSecret = AES(CBC mode, K_shared, IV_shared, aes_data)

struct encrypted_programming_header_t encrypted_header = {16,
                                                          encryptedSecret,
                                                          salt,
                                                          info}

// Encrypt each block with AES using AES key K and IV
```

```
for block in image:
       image_encrypted[block] = AES(CBC mode, K, IV, PKCS7, block)

// Connect to the device system access port
device = programmer.connect(SYSAP)

// Call encrypted programming sys call with init arguments
status = device.EncryptedProgramming(mode=init, encrypted_header)

// Program the encrypted image block by block
if SUCCESS == status:
       for block in image_encrypted:
              status &= device.EncryptedProgramming(mode=program,
                                                    image_encrypted[block])

// End the encrypted programming operation
if SUCCESS == status:
       status = device.EncryptedProgramming(mode=Finish)

return status
```

5.5.2 Image encryption algorithm

The image encryption algorithm produces two outputs:

- The encrypted image. It is passed to the `EncryptedProgramming(mode=Program, encrypted row)` call.
- The initialization data for the Elliptic-curve Diffie–Hellman (ECDH) key agreement protocol. It is passed to the `EncryptedProgramming(mode=Init, initialization data)` call.

The key used to encrypt the image is passed to the device encrypted using a key derived with the help of the ECDH algorithm. The derived key is called a shared secret. For the ECDH algorithm, the host side (the side that does the encryption) uses its private key (whose public part is known to the device) and a device public key (whose private part is known to the device). The following key combinations are available:

- OEM or HSM private keys
- Device or group public keys

These options mean that:

- Encrypted programming is possible only after the RoT is transferred (OEM and HSM public keys are stored in the device) and the device identity is created (device and/or group private keys are stored in the device and their public parts are known to the host).
- Choice of which key the OEM or HSM should use depends on who does the image encryption: OEM or HSM, because only that side owns the private key.
- Choosing the device key means that only this particular device can decrypt the image.

- Choosing the group key means that the image can be decrypted by the multiple devices that share the same group key.

Do the following to encrypt the bootloader image:

1. Generate a random AES key K and initialization vector IV. The length of the key is 128 or 256; the length of the IV is always 128 bits.
2. Split the image into blocks of 512 bytes (rows). If the size of the image is not a multiple of 512, pad the last block with zeros.
3. Encrypt each block using the AES algorithm in CBC mode with PKCS#7 padding using key K and IV generated in the first step.

The initialization data is a block of binary data. Its length is 97 bytes and its structure is defined by the following C structure:

```
#define SECRET_SIZE     64
typedef struct
{
       uint8_t aesKeySize; /* 16 for 128-bit key, 32 for 256-bit key */
       uint8_t encryptedSecret[SECRET_SIZE];
       uint8_t salt[16];
       uint8_t info[16];
}encrypted_programming_header_t;
```

The `encryptedSecret` array contains the encrypted key K and IV generated previously and used to encrypt the image rows. The key and IV are packed in a structure defined as follows and then encrypted using the AES algorithm in CBC mode with no padding, using the key and initialization vector derived using the ECDH algorithm.

```
#define AES_KEY_SIZE    16 /* 16 for 128-bit key, 32 for 256-bit key */
#define IV_SIZE         16
typedef struct
{
       uint8_t key[AES_KEY_SIZE];
       uint8_t iv[IV_SIZE];
       uint8_t padding[SECRET_SIZE - AES_KEY_SIZE - IV_SIZE];
}aes_data_t;
```

Do the following to prepare the initialization data:

1. Generate the shared secret: `K_shared, IV_shared = ECDH(SHA256 mode, host private key, device public key, salt, info).`
2. Pack the key K and IV into the `aes_data_t` structure.
3. Encrypt the `aes_data_t` structure using the AES CBC with `K_shared` and `IV_shared`: `encryptedSecret = AES(CBC mode, K_shared, IV_shared, aes_data_t).`
4. Pack the `encryptedSecret`, `salt` and `info` into the `encrypted_programming_header_t` structure.

5.6 Provisioning command execution

The provisioning command JWT is injected into the device using the `ProcessProvisionCmd` system call. It can be called multiple times with the provisioning command JWTs containing different sets of tokens. Until the provisioning command contains the "complete" token set to true, the device is not considered as provisioned: the policies are not applied and it does not attempt to securely boot. Each call of the `ProcessProvisionCmd` system call is followed by the response, which contains the information about the device. The format of the provisioning command JWT and the device response are described in the previous sections.

6 System Calls API reference

6.1 System Calls list

The table below lists the provisioning-related system calls and describes them at a high level. Each call is detailed in the following sections.

System call	Opcode	Description	Category
`RegionHash`	0x31	Compute the hash of a given region using a given hash algorithm. Use this to verify that some region of the memory is empty (values are equal to some expected pattern).	Provisioning
`ProcessProvisionCmd`	0x33	Inject keys and policies including the OEM RoT key. These keys and policies are certified by Infineon and the OEM, and are verified by this function. This function converts the device from "SECURE UNCLAIMED" to "SECURE CLAIMED" life cycle.	Provisioning
`EncryptedProgramming`	0x34	Use an encrypted firmware image when programming. The data is transferred in the encrypted form and is decrypted on the device.	Provisioning
`GetProvDetails`	0x37	Returns the provisioning packet (JWT), templates, or public key strings in JSON format.	Other
`DAPControl`	0x3A	Controls the DAP access if it is allowed by the debug policy.	Debug

6.1.1 RegionHash

Computes the hash of a given region using a given hash algorithm.

In addition, it can compare a region against a value. Use this to determine whether a region is empty. When byte 1 of the `SRAM_SCRATCH` parameter is set to Compare, this call compares the specified region to the compare value. It returns the `CY_FB_INVALID_FLASH_OPERATION` error if the procedure fails.

This call checks whether the client has read access to the requested memory region by looking into DAP MPU and SMPUs. If called on an out-of-bounds flash region, or the client does not have read access, the `CY_FB_SYSCALL_ADDR_PROTECTED` status is returned.

Input Values

Address	Value to be written	Description
IPC_STRUCT.DATA		
Bits [31:0]	`SRAM_SCRATCH_ADDR`	32-bit aligned SRAM address where the short set of the call's parameters are stored
SRAM_SCRATCH		
Bits [31-24]	0x31	Opcode for this system call
Bits [23-16]	Compare value	

Address	Value to be written	Description
Bits [15-8]	0x02 – SHA256 0xFF – Compare	Type of operation
Bits [7-0]	0xXX	Not used
SRAM_SCRATCH +0x04		
Bits [31-0]	SRAM_SCRATCH2_ADDR	32-bit aligned SRAM address where the extended set of the call's parameters are stored
SRAM_SCRATCH2		
Bits [31-0]	Number of bytes of data	Memory region for which the hash is calculated
SRAM_SCRATCH2 +0x04		
Bits [31-0]	Data address	32-bit aligned system address of the first byte of the data

Return Values

Address	Return Value	Description
SRAM_SCRATCH		
Bits [31-24]	0xA0 = Success 0xF0 = Error	Status code for this system call
Bits [23-0] – for Success	0x00	

Bits [23-0] – for Error	Error Code	`CY_FB_SYSCALL_ADDR_PROTECTED`
		`CY_FB_SYSCALL_INVALID_ARGUMENT`
		`CY_FB_INVALID_CRYPTO_OPER`
		`CY_FB_INVALID_FLASH_OPERATION`

SRAM_SCRATCH +0x04		
Bits [31-0]	SRAM_SCRATCH2_ADDR	32-bit aligned SRAM address where the extended set of the call's parameters are stored; not used for the compare operation.
SRAM_SCRATCH2		
Bits [31-0]	Size in bytes of the Hash	Not used for the Compare operation
SRAM_SCRATCH2 +0x04		
Bits [31-0]	Hash address	32-bit aligned system address of the first byte of the hash; not used for the compare operation.

6.1.2 ProcessProvisionCmd

This function injects keys and policies. These keys and policies are certified by Infineon and the OEM, and verified by this function. The function uses a certificate chain to verify the Infineon and OEM signatures. For Infineon, the initial certificate is signed by the Hardware Security Module (HSM) using a private key generated by the HSM. The second certificate certifies the corresponding public key. The second certificate is signed by Infineon, whose public key is stored in the device. A similar scheme is used for the OEM certificate.

This function converts the device from the "SECURE UNCLAIMED" to the "SECURE CLAIMED" life cycle state.

When the system call handler receives the `ProcessProvisionCmd`, it parses the JWT payload and acts based on the claims contained in the payload. The structure of the provisioning command JWT is described in the **Provisioning command** section.

`cy_auth`:

- Validates the input packet
  - Verifies the CY signature of the `cy_auth` packet
  - Verifies the CY public key in the `cy_auth` packet
  - Imports the HSM public key
  - Verifies the HSM signature of the full `prov_cmd.jwt` packet.
  - Verifies the `DIE_ID` and `DEV_ID` range in the packet with the device identity

`create_identity = true`:

- Generates the UDS, device public, and private keys (if not received in same packet)
  - Encrypts the device and the received group key derived from the UDS key and stores to flash. AES-CBC with PKCS7 padding is used for encryption.
  - Stores the UDS key to eFuses that are protected by the PPU.
- Calculates and blows “SECURE_HASH2” eFuse from the encrypted package.

`rot_auth`:

- Validates the input packet
  - Imports the OEM public key
  - Verifies the OEM signature of the `rot_auth` packet
  - Compares the HSM public key in the `cy_auth` and `rot_auth` packets
- Stores the OEM public key and `prod_id` in flash

`prod_id`:

- Verifies “`prod_id`” with the “`product_id`” stored in flash during the `rot_auth` packet process

`“chain_of_trust”`, `“prov_req”`:

- Stores the JWT packet to flash when the OEM signature is correct

`image_cert`:

- Verifies the OEM signature of the input packet and stores it to flash
- Extracts the `"image_address"` and `"image_size"` values of the bootloader image
- Verifies the hash for the bootloader image with the hash value from the bootloader FW image distribution packet
- Bootloader image must be programmed to flash before this token execution

`complete = true`:

- Verifies the policies against templates (only min/max values and types)
- Calculates and preserves “SECURE_HASH3” in eFuses. “SECURE_HASH3” is calculated from:
  - OEM public key
  - Product ID
  - If “reprovision” > “keys_and_policies” = false: the provisioning JWT packet (`prov_req` and `chain_of_trust`)
  - If “reprovision” > “boot_loader” = false: the bootloader image and image certificate

- Calculates “SECURE_HASH4”, signs it with the device private key, and stores it to flash. “SECURE_HASH4” is calculated from:
  - If “reprovision” > “keys_and_policies” = true: provisioning the JWT packet (`prov_req` and `chain_of_trust`)
  - If “reprovision” > “boot_loader” = true: the bootloader image and image certificate

After parsing and action on the payload, the call generates a JWT response packet signed by the device private key with these items (if they are present in the device):

- Device ID (silicon ID, family ID)
- DIE ID (lot, wafer, xpos, ypos, day, month, year)
- Device public key
- Group public key
- OEM public key
- Product ID
- Complete status

After the device has been transitioned to the “SECURE CLAIMED” mode, “Secure Flashboot” performs the following tasks during every boot:

- Validates “SECURE_HASH3” before transferring control to the Infineon bootloader
- Protects the bootloader flash region to avoid unexpected clearing of this image
- Validates “SECURE_HASH4” if “reprovision”-> “keys_and_policies” = true or “reprovision”-> “boot_loader” = true

Input Values

Address	Value to be Written	Description
IPC_STRUCT.DATA		
Bits [31:0]	SRAM_SCRATCH_ADDR	32-bit aligned SRAM address where the short set of the call’s parameters are stored
SRAM_SCRATCH		
Bits [31-24]	0x33	Opcode for this system call
Bits [23-0]	Unused	
SRAM_SCRATCH +0x04		
Bits [31-0]	SRAM_SCRATCH2_ADDR	32-bit aligned SRAM address where the extended set of the call’s parameters are stored
SRAM_SCRATCH2		
Bits [31-0]	Number of bytes of data	Include these four bytes in the byte count
SRAM_SCRATCH2 +0x04		
Arbitrary length	JWT payload	The JWT packet, signed by HSM private key: JWT packet (`cy_auth`) signed by the Infineon private key with: HSM public key Infineon public key Optionally JWT packet (rot_auth) signed by the OEM private key with:

Address	Value to be Written	Description
		HSM public key OEM public key When the "complete" field in the JWT packet is set to true, the following items are required: JWT packet (prov_req) signed by the OEM private key for provisioning with policies JWT packet (image_cert) signed by any private key referenced and provisioned in prov_req->boot_upgrade->firmware->boot_auth with the bootloader image hash Optionally certificates (chain_of_trust)

Return Values

Address	Return Value	Description
SRAM_SCRATCH		
Bits [31-24]	0xA0 = Success 0xF0 = Error	Status code for this system call
Bits [23-0] – for Success	0x00	
Bits [23-0] – for Error	Error Code	CY_FB_SYSCALL_INVALID_ARGUMENT
SRAM_SCRATCH +0x04		
Bits [31-0]	SRAM_SCRATCH2_ADDR	32-bit aligned SRAM address where the extended set of the call's parameters are stored
SRAM_SCRATCH2		
Bits [31-0]	Number of bytes in the response	
SRAM_SCRATCH2 +0x04		
Bits [31-0]	Address where JWT response packet is stored.	32-bit aligned SRAM address of the first byte of the response

6.1.3 EncryptedProgramming

If you are using an encrypted firmware image, use this command to decrypt the image for programming on the production line. The data is transferred in the encrypted form and is decrypted on the device. This protects the firmware from theft in the supply chain.

Note: Encryption is intended to protect the confidentiality of the binary images in transit. However, after these images are decrypted inside the device, they are available in plain text. The firmware can be protected by disabling the debug ports.

There are three modes of operation for this command: Init, Program, and Finish.

During Init, this call validates the programming request, decrypts the decryption key, and sets up for encrypted programming.

During Program, the actual data programming happens using calls similar to the `WriteRow` syscall. This means that the normal programming flow can be used. However, the data must be programmed to an address inside the region set up during the Init step. The data step will be executed repeatedly, one row at a time, with the address set for each row, until the image is completely programmed.

During Finish, the command cleans up the initialized data.

Note: Encrypted programming (flash write) cannot be performed in Ultra Low Power mode (core voltage 0.9 V).

Init:

- Validates the programming request
- Decrypts the decryption key
  - ECDH key agreement by using the device or group private/public keys and HSM or OEM public key, the shared secret is generated
  - Generate the key material from the shared secret, salt and info: AES key and IV
  - Import AES key to the key storage
  - Init cipher decrypt setup
  - Set Initialization Vector to be used with the AES decryption key
  - Import decryption AES key to the key storage
- Sets up for encrypted programming

Program:

- Sets Initialization Vector
- Decrypts the data
- Programs it to flash

Finish:

- Cleans the initialized data

Input Values

Address	Value to be Written	Description
IPC_STRUCT.DATA		
Bits [31:0]	SRAM_SCRATCH_ADDR	32-bit aligned SRAM address where the short set of the call's parameters are stored
SRAM_SCRATCH		
Bits [31-24]	0x34	Opcode for this system call
Bits [23-16]	4 = HSM key 5 = OEM key	Peer (host) key ID (the key slot)
Bits [15-8]	1 = device key 12 = group key	Private key ID (the key slot)
Bits [7-0]	0x00 = Init 0x01 = Data 0x02 = Finish	Command mode
SRAM_SCRATCH +0x04		

Address	Value to be Written	Description
Bits [31-0]	SRAM_SCRATCH2_ADDR	32-bit aligned SRAM address where the extended set of the call's parameters are stored
SRAM_SCRATCH2		
Bits [31-0]	Number of bytes of data	Init mode = length of the encryption header Data mode = Data size to program
SRAM_SCRATCH2 +0x04		
Length varies	Init Mode: • AES key size (1 byte) – value 16 = 128-bit key; 32 = 256-bit key • encrypted AES key (16 or 32 bytes) – ECDH key agreement algorithm should be used with the device or group private/public keys and the HSM/OEM public key – for a 16-byte key, the first 16 bytes hold the key • encrypted initialization vector (16 bytes) • padding (16 bytes); required only if the key length is 128 bits. • salt and label to derive the key encryption key (16 bytes each, 32 total) Program Mode: • flash address to be programmed (4 bytes) • encrypted data array with padding to decode and program one flash row (512 bytes + 16 bytes padding)	

Return Values

Address	Return Value	Description
SRAM_SCRATCH		
Bits [31-24]	0xA0 = Success 0xF0 = Error	Status code for this system call
Bits [23-0] – for Success	0x00	
Bits [23-0] – for Error	Error Code	`CY_FB_SYSCALL_INVALID_ARGUMENT`

6.1.4 GetProvDetails

This call returns the provisioning packet (JWT), templates, or public key strings in JSON format.

Input Values

Address	Value to be Written	Description
IPC_STRUCT.DATA		
Bits [31:0]	SRAM_SCRATCH_ADDR	32-bit aligned SRAM address where the short set of the call's parameters are stored
SRAM_SCRATCH		
Bits [31-24]	0x37	Opcode for this system call

Address	Value to be Written	Description
Bits [23-0]	Unused	
SRAM_SCRATCH +0x04		
Bits [31-0]	`SRAM_SCRATCH2_ADDR`	32-bit aligned SRAM address where the extended set of the call's parameters are stored
SRAM_SCRATCH2		
Bits [31-0]	Index value	An index value for the desired information. Values 1-12 represent the contents of the numbered key slot maintained in the flashboot for Mbed crypto key storage. Additional options are (FB = flashboot) • 0x100 - `FB_POLICY_JWT` • 0x101 - `FB_POLICY_TEMPL_BOOT` • 0x102 - `FB_POLICY_TEMPL_DEBUG` • 0x2xx - `FB_POLICY_CERTIFICATE`, where xx is a certificate index in the `"chain_of_trust"` array of the provisioned packet. • 0x300 - `FB_POLICY_IMG_CERTIFICATE`

Return Values

Address	Return Value	Description
SRAM_SCRATCH		
Bits [31-24]	0xA0 = Success 0xF0 = Error	Status code for this system call
Bits [23-0] – for Success	0x00	
Bits [23-0] – for Error	Error Code	`CY_FB_SYSCALL_ADDR_PROTECTED` `CY_FB_SYSCALL_INVALID_ARGUMENT`
SRAM_SCRATCH +0x04		
Bits [31-0]	`` ` ``SRAM_SCRATCH2_ADDR	32-bit aligned SRAM address where the extended set of the call's parameters are stored
SRAM_SCRATCH2		
Bits [31-0]	Size in bytes of the response string	
SRAM_SCRATCH2+0x04		
Bits [31-0]	Address of the response string	

6.1.5 DAPControl

This SysCall allows firmware to disable or enable DAP access at run-time, if that is permitted by the debug policy. The table explains the various values that can be set for the properties in the debug policy that affect this call.

Permission	control	Notes
disabled	ignored	DAP is closed and cannot be opened. The syscall always returns `CY_FB_PD_PERM_NOT_ALLOWED`.
enabled	ignored	DAP is open, but may be closed or opened by this syscall
allowed	firmware	DAP is closed, but may be opened or closed by this syscall
	certificate	DAP is closed, but may be opened if a certificate is provided. The certificate must be signed by the key specified in the debug policy. The DAP may be closed without a certificate.

The required certificate is in JWT form. The input and return values for this syscall vary depending on whether a certificate is required.

Input Values (certificate required)

Address	Value to be Written	Description
IPC_STRUCT.DATA		
Bits [31:0]	SRAM_SCRATCH_ADDR	32-bit aligned SRAM address where the short set of the call's parameters are stored
SRAM_SCRATCH		
Bits [31-24]	0x3A	Opcode for this system call
Bits [23-16]	0 = Disable 1 = Enable	Desired state of DAP access; close or open the DAP
Bits [15-8]	0 = CM0+ 1 = CM4 2 = System	Which port to control
Bits [7-0]	0 = certificate required	
SRAM_SCRATCH +0x04		
Bits [31-0]	SRAM_SCRATCH2_ADDR	32-bit aligned SRAM address where the extended set of the call's parameters are stored
SRAM_SCRATCH2		
Bits [31-0]	Size in bytes of the JWT packet	
SRAM_SCRATCH2 +0x04		
Varies	Start of the JWT packet	The JWT includes a valid "`cy_auth`" claim with valid DIE_ID range. It must be signed by the private key specified in the debug policy key field for the port in question.

Return Values (certificate required)

Address	Return Value	Description
SRAM_SCRATCH		
Bits [31-24]	0xA0 = Success	Status code for this system call

Address	Return Value	Description
	0xF0 = Error	
Bits [23-0] – for Success	0x00	
Bits [23-0] – for Error	Error Code	`CY_FB_SYSCALL_INVALID_ARGUMENT` `CY_FB_PD_PERM_NOT_ALLOWED`

Input Values (certificate not required)

Address	Value to be Written	Description
IPC_STRUCT.DATA		
Bits [31-24]	0x3A	Opcode for this system call
Bits [23-16]	0 = Disable 1 = Enable	Desired state of DAP access, close or open the DAP
Bits [15-8]	0 = CM0+ 1 = CM4 2 = System	Which port to control
Bits [7-0]	1 = certificate not required	

Return Values (certificate not required)

Address	Return Value	Description
IPC_STRUCT.DATA		
Bits [31-24]	0xA0 = Success 0xF0 = Error	Status code for this system call
Bits [23-0] – for Success	0x00	
Bits [23-0] – for Error	Error Code	`CY_FB_SYSCALL_INVALID_ARGUMENT` `CY_FB_PD_PERM_NOT_ALLOWED`

6.2 Status codes

At the end of every system call, a status code will be written over the arguments in the IPC_DATA register or in the SRAM location pointed by IPC_DATA. A success status is AXXXXXXX, where X indicates don't care values. A failure status is indicated by F<0/1/7>XXXXXX, where XXXXXX is the failure code.

"Secure Flashboot" also returns several status codes in DAP IPC_DATA at the end of the boot. Debug probes need to preserve them before invoking system calls.

Status	Value	Description
`STATUS_SUCCESS`	0xAXXXXXX	PASS, X - don't care
`STATUS_INVALID_PROTECTION`	0xF0000001	Reject the system call when CPUSS_PROTECTION is not NORMAL
`CY_FB_SYSCALL_ADDR_PROTECTED`	0xF0000008	Returned by all APIs when client doesn't have access to the region it is using for passing arguments
`CY_FB_SYSCALL_INVALID_OPCODE`	0xF000000B	Opcode not a valid API opcode
`CY_FB_SYSCALL_MASK`	0xFF000000	Status mask of the "Secure Flashboot" return value

Status	Value	Description
CY_FB_INVALID	0xF7000000	Fail status of the "Secure Flashboot" return value
CY_FB_INVALID_STATE_DEAD	0xF700DEAD	Device is in DEAD state
CY_FB_ALREADY_PROVISIONED	0xF7000001	Device is already provisioned; re-provisioning is not supported.
CY_FB_INVALID_FLASH_OPERATION	0xF7000002	Write to flash operation failed
CY_FB_INVALID_CY_JWT_SIGNATURE	0xF7000003	Signature verification of the JWT packet, which should be signed by Infineon private key, failed (cy_auth or image_cert)
CY_FB_INVALID_OEM_JWT_SIGNATURE	0xF7000004	Signature verification of the JWT packet, which should be signed by OEM private key, failed (rot_auth or prov_req)
CY_FB_INVALID_HSM_JWT_SIGNATURE	0xF7000005	Signature verification of the JWT packet, which should be signed by HSM private key, failed (prov_cmd)
CY_FB_INVALID_JWT_STUCTURE	0xF7000006	The cy_auth token in the prov_cmd packet doesn't contain a valid hsm_pub_key
CY_FB_INVALID_JWT_STUCTURE	0xF7000206	The cy_auth token in the prov_cmd packet doesn't contain a valid cy_pub_key
CY_FB_INVALID_JWT_STUCTURE	0xF7000306	The rot_auth token in the prov_cmd packet doesn't contain a valid oem_pub_key
CY_FB_INVALID_JWT_STUCTURE	0xF7000406	The rot_auth token in the prov_cmd packet doesn't contain a valid hsm_pub_key
CY_FB_INVALID_JWT_STUCTURE	0xF7000506	body part of the prov_cmd packet is absent
CY_FB_INVALID_JWT_STUCTURE	0xF7000606	base64_decode of the body part of the prov_cmd packet failed
CY_FB_INVALID_JWT_STUCTURE	0xF7000706	JSON parser failed with the decoded body part of the prov_cmd packet
CY_FB_INVALID_JWT_STUCTURE	0xF7000806	The "cy_auth" token is absent in the parsed body part of the prov_cmd packet
CY_FB_INVALID_JWT_STUCTURE	0xF7000C06	The "exp" token is absent in the parsed body part of the cy_auth packet
CY_FB_INVALID_JWT_STUCTURE	0xF7001706	The "oem_pub_key" token is absent in the parsed body part of the rot_auth packet
CY_FB_INVALID_JWT_STUCTURE	0xF7001806	The "hsm_pub_key" token is absent in the parsed body part of the rot_auth packet
CY_FB_INVALID_JWT_STUCTURE	0xF7001906	The "prod_id" token is absent in the parsed body part of the rot_auth packet
CY_FB_INVALID_INFINEON_KEY	0xF7000007	Import of Infineon public key from JSON to key storage or the validation of this key in cy_auth packet failed
CY_FB_INVALID_DEV_KEY_GEN	0xF7000009	Device key generation failed

Status	Value	Description
CY_FB_SYSCALL_PROTECTED	0xF700000A	Master with PC > 4 tried to use PSA syscall, but protection is enabled in the policy ((“protect_flags” & 2) != 0)
CY_FB_INVALID_SEC_HASH_CALC	0xF700000B	“Secure hash” calculation failed
CY_FB_INVALID_HSM_KEY	0xF700000C	Compare of the HSM public key in the cy_auth and rot_auth packets failed
CY_FB_INVALID_PROD_ID	0xF700000E	Validation of the prod_id in prov_req packet failed
CY_FB_INVALID_EFUSE_BLOW	0xF700000F	eFuse blow failed
CY_FB_INVALID_PC_CHANGE	0xF7000010	Protection Context change of the Crypto block failed
CY_FB_INVALID_JWT_POLICY	0xF7000011	Validation of provision policy by the schema failed
CY_FB_PD_PERM_NOT_ALLOWED	0xF7000012	Returned by TransitionToRMA or SyscallDPControl when requested action (enable DP or RMA) is not permitted by the provisioned policy
CY_FB_INVALID_CRYPTO_OPER	0xF7000013	Cryptographic operation failed
CY_FB_INVALID_JWT_ID_TOO_LONG	0xF7000014	The “prod_id” string is too long. The maximum allowed length is 64 bytes.
CY_FB_INVALID_IMG_HASH	0xF7000015	Calculated secured image hash doesn’t match with a hash received in the image_cert packet
CY_FB_INVALID_UDS_KEY_GEN	0xF7000017	Generation of a device UDS failed or it already exists
CY_FB_INVALID_GRP_KEY_IMPORT	0xF7000018	Import of the group key failed
CY_FB_INVALID_KEY_ENCRYPTION	0xF7000019	Encryption and storing of the device and group private keys failed
CY_FB_IDENTITY_CREATED	0xF700001A	Device private keys already exist
CY_FB_INVALID_CHAIN_LENGTH	0xF700001B	Length of the Chain of Trust string is too long. Maximum length is 5120 bytes.
CY_FB_INVALID_POLICY_LENGTH	0xF700001C	Length of the provisioning request string is too long. Maximum length is 10240 bytes.
CY_FB_INVALID_IMG_CERT_LENGTH	0xF700001D	Size of the image certificate is too big. Maximum size is 1024 bytes.
CY_FB_INVALID_IMG_JWT_SIGNATURE	0xF700001E	Image certificate signature validation failed
CY_FB_INVALID_IMG_JWT_STRUCTURE	0xF700001F	Structure of the image certificate is not correct
CY_FB_INVALID_DEV_ID	0xF7000020	Device ID in the “cy_auth” packet does not match with the provisioned device
CY_FB_INVALID_DIE_ID	0xF7000021	Die ID in the “cy_auth” packet does not match with the provisioned device
CY_FB_INVALID_REPROV_POLICY	0xF7000022	Re-provisioning settings in the policy does not match the previously provisioned ones
CY_FB_INVALID_REPROV_CERT	0xF7000023	Re-provisioning certificate signature validation failed
CY_FB_SYSCALL_INVALID_ARGUMENT	0xF7000024	Invalid arguments passed to a SysCall

Status	Value	Description
`CY_FB_SYSCALL_KEY_PROTECTED`	0xF7000025	Master with PC > 4 tried to use the `psa_asymmetric_sign` API with an internal key, but protection is enabled in the policy (`“protect_flags” != 0`)
`CY_FB_ADDRESS_OUT_OF_RANGE`	0xF7000026	The `EncryptedProgramming` SysCall tried to program regions of the flash memory that are not permitted according to the policy
`CY_FB_INVALID_KEY_LENGTH`	0xF7000027	Provisioning key length in JSON format is equal to 0 or is greater than 172 bytes
`CY_FB_INVALID_MEM_ALLOC`	0xF70000FF	Memory allocation failed

6.3 Firmware presence status

After provisioning and installing the firmware, verify that the firmware is in place.

Trigger a reset on the device by toggling the XRES pin.

“Secure Flashboot” checks the image certificate and default test firmware address (0x10000000) for the presence or absence of a valid firmware image in the device. It records the result of this check in the first word of “Secure Flashboot” public SRAM region that has Read only access. The public SRAM address is device-dependent. The addresses are provided in the table below:

Device	Public SRAM address
CYB06447BZI	0x08044800
CYS0644xxZI or CYB064xxZI	0x080FE000
CYB06445LQI	0x0803E000

If there is no firmware present, “Secure Flashboot” enters an idle loop after recording the result.

“Secure Flashboot” result values:

Value	Description
0xA1000100	User firmware exists and running on the CM4 core
0xA1000101	First immutable image firmware (bootloader) exists and running on the CM0+ core
0xFxxxxxxx	User firmware doesn't exist, or error occurred during firmware verification
0xF700001F	Invalid structure of the image certificate
0xF7000023	Invalid re-provisioning certificate
0xF7000106	FLL configuration failed
0xF7000107	Invalid application structure
0xF700010B	Invalid “SECURE_HASH3”
0xF7000110	Invalid policy
0xF7000111	Invalid “SECURE_HASH2”
0xF7001108	Invalid device private key
0xF700110E	Invalid protection settings
0xF7800000	Invalid ASSET_HASH calculated based on TOC1

Revision history

Date	Version	Description
2021-04-07	**	Initial release
2021-07-13	*A	Added a note in the **Provisioning command** section
2023-02-08	*B	Updated the **Provisioning command** section

Edition 2023-02-08
Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to www.infineon.com/support

Document reference
002-32406 Rev. *B