PSOC™ 6 MCU

Programming specification

About this document

Scope and purpose

This document provides the information necessary to program the nonvolatile memory of the PSOC™ 6 MCU family. It describes the communication protocol required for access by an external programmer, explains the programming algorithm, and gives a basic description of the physical connection. The pin locations, electrical, and timing specifications of the physical connection are not part of this document. They can be found in the device datasheet. The programming algorithms described in the following sections are compatible with all PSOC™ 6 MCUs except PSOC™ 64 devices, which are described in [1].

Intended audience

This programming specification is intended for those developing programming solutions for the target devices. This includes third-party production programmers, as well as customers wanting to develop their programming systems. For more information, see Programming Solutions for Industrial and IoT Microcontrollers.

Table of contents

About this document **1**
Table of contents **2**
1 Introduction **4**
1.1 Programmer 4
1.2 PSOC™ 6 MCU family overview 5
2 Nonvolatile memory subsystem **6**
2.1 Application flash 7
2.2 Auxiliary flash (AUXflash) 7
2.3 Supervisory flash (Sflash) 7
2.4 Electronic fuses (eFuse) 8
2.5 Execute-in-place (XIP) 9
3 Hex file **10**
3.1 Organization of the hex file 10
4 Protocol stack **13**
4.1 Communication interface 14
4.2 Program and debug interface 14
4.2.1 DAP security 15
4.2.2 DAP power domain 15
4.2.3 SWD/JTAG selection 15
4.2.4 Hardware access commands 17
4.2.5 Pseudocode 18
4.3 Physical layer 19
5 Programming algorithm **24**
5.1 High-level programming flow 24
5.2 Constants and subroutines used in the programming flow 25
5.2.1 Constants 25
5.2.2 Subroutines 27
5.3 Step 1.A – Acquire PSOC™ 6 MCU 32
5.4 Step 1.B – Acquire PSOC™ 6 MCU (alternate method) 35
5.5 Step 2 – Identify silicon 37
5.6 Step 3 – Erase application flash 38
5.7 Step 4 – Verify blank checksum (optional) 39
5.7.1 Pseudocode – Step 4. Verify checksum 39
5.8 Step 5 – program application flash 40
5.9 Step 6 – Verify application flash 41
5.10 Step 7 – Verify checksum (optional) 43
5.10.1 Pseudocode – Step 7. Verify checksum 43
5.11 Step 8 – Program and verify AUXflash 43
5.12 Step 9 – Program and verify Sflash 46
5.13 Step 10 – Program eFuse 46
5.14 Step 11 – Verify eFuse (optional) 54
6 Appendix A: Intel hex file format **56**
7 Appendix B: eFuse data mapping in file **58**
8 Appendix C: Serial wire debug (SWD) Protocol **62**
9 Appendix D: Joint test action group (JTAG) protocol **65**
References **67**

Table of contents

Revision history....................68
Disclaimer....................74

1 Introduction

This document provides the information necessary to program the nonvolatile memory of the PSOC™ 6 MCU family. It describes the communication protocol required for access by an external programmer, explains the programming algorithm, and gives a basic description of the physical connection. The pin locations and the electrical and timing specifications of the physical connection are not a part of this document. They can be found in the device datasheet. The programming algorithms described in the following sections are compatible with all PSOC™ 6 MCUs.

1.1 Programmer

Figure 1 illustrates a high-level view of the programmer environment.

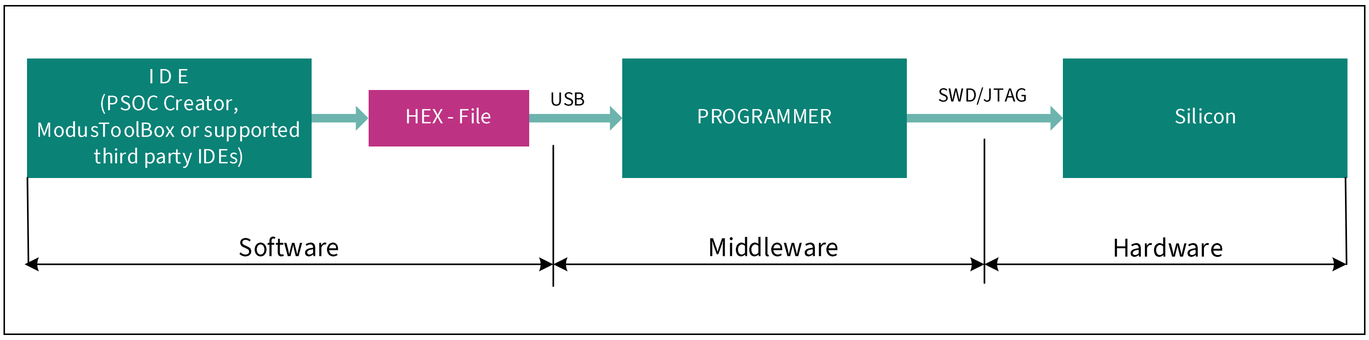


Figure 1 Programmer in the development environment

In the manufacturing environment, the integrated development environment (IDE) block is absent because its main purpose is to produce the binary file (hex, elf, and so on). Figure 1 shows, that the programmer performs the functions:

- Parses the binary file and extracts the necessary information
- Interfaces with the silicon as a serial wire debug (SWD) or JTAG master
- Implements the programming algorithm by translating the data from the binary file into SWD or JTAG signals

The structure of the programmer depends on its requirements. It can be software- or firmware-centric.

In a software-centric structure, the programmer's hardware is between the protocol (USB) and SWD. An external device (software) passes all the SWD commands to the hardware through the protocol. The bridge is not involved in parsing the binary file and programming algorithm; this is the task of the upper layer (software)—for example, MiniProg3 and Segger J-Link.

A firmware-centric structure is an independent hardware design in which all the functions of the programmer are implemented in one device, including storage for the binary file. Its main purpose is to act as a mass programmer in manufacturing.

This document does not discuss the specific implementation of the programmer. It focuses on data flow, algorithms, and physical interfacing.

1.2 PSOC™ 6 MCU family overview

The PSOC™ 6 MCU family is a dual-CPU solution, with both the Arm® Cortex®-M4 and Cortex®-M0+ processor cores. This MCU family supports the Arm® SWJ-DP Interface for programming and debugging operations, using SWD or JTAG protocols.

The nonvolatile subsystem of the silicon consists of a flash memory system. The flash memory system stores the user's program or data, as well as eFuses.

The part can be programmed after it is installed in the system by way of the SWD or JTAG interface (in-system programming).

See the device datasheet for the specifications on memory size and programming frequency range.

This document focuses on the specific programming operations without referencing the silicon architecture. Many important topics are detailed in the appendices. Other device-specific information can be found in the device's datasheet or technical reference manual.

This document includes the following appendices:

- Appendix A: Intel hex file format
- Appendix B: eFuse data mapping in file
- Appendix C: Serial wire debug (SWD) Protocol
- Appendix D: Joint test action group (JTAG) protocol

2 Nonvolatile memory subsystem

This chapter describes the nonvolatile memory subsystem of the PSOC™ 6 MCU silicon.

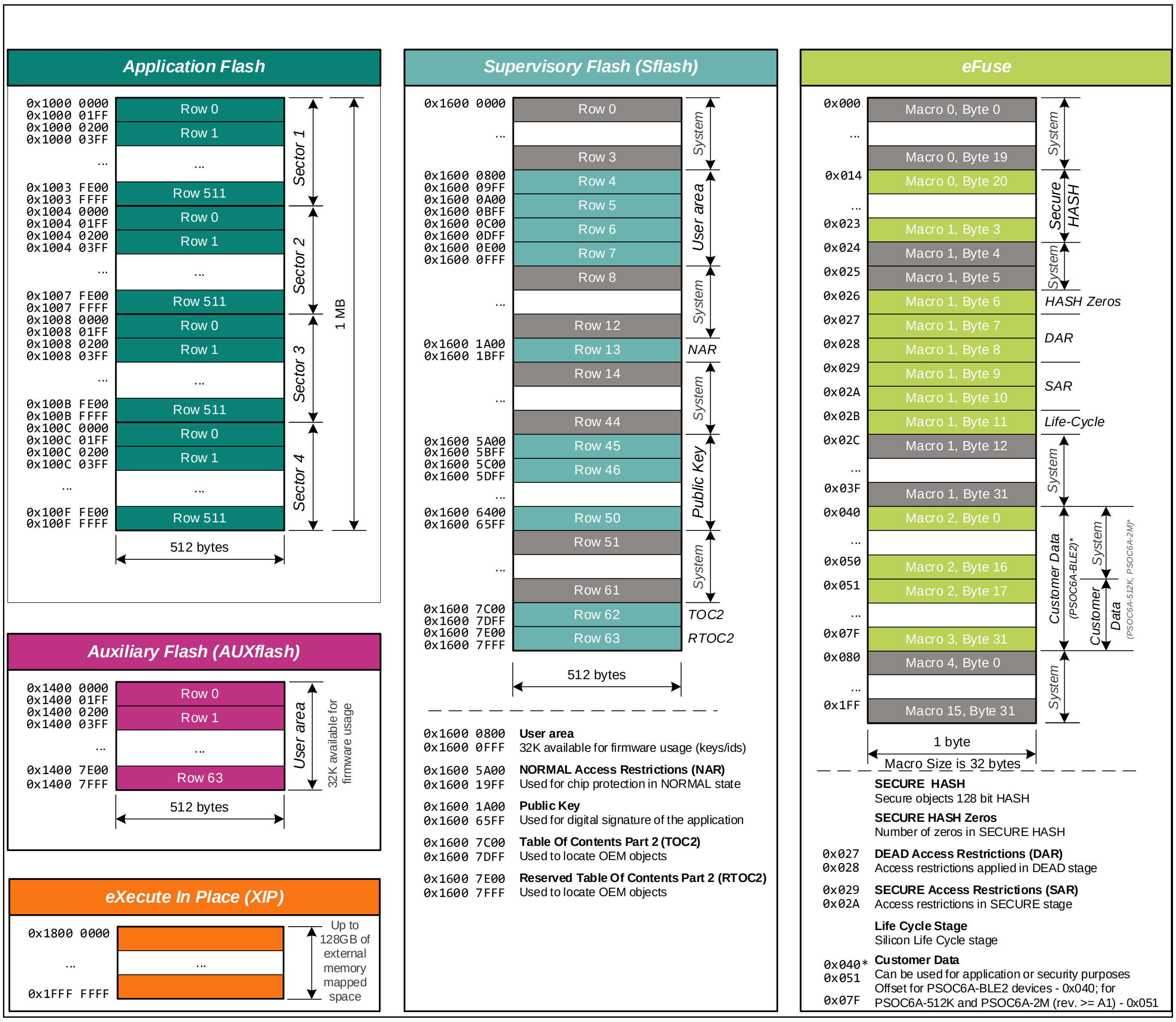

Application Flash
0x1000 0000 / 0x1000 01FF Row 0
0x1000 0200 / 0x1000 03FF Row 1
...
0x1003 FE00 / 0x1003 FFFF Row 511
Sector 1
0x1004 0000 / 0x1004 01FF Row 0
0x1004 0200 / 0x1004 03FF Row 1
...
0x1007 FE00 / 0x1007 FFFF Row 511
Sector 2
0x1008 0000 / 0x1008 01FF Row 0
0x1008 0200 / 0x1008 03FF Row 1
...
0x100B FE00 / 0x100B FFFF Row 511
Sector 3
0x100C 0000 / 0x100C 01FF Row 0
0x100C 0200 / 0x100C 03FF Row 1
...
0x100F FE00 / 0x100F FFFF Row 511
Sector 4
1 MB
512 bytes
Auxiliary Flash (AUXflash)
0x1400 0000 / 0x1400 01FF Row 0
0x1400 0200 / 0x1400 03FF Row 1
...
0x1400 7E00 / 0x1400 7FFF Row 63
User area
32K available for firmware usage
512 bytes
eXecute In Place (XIP)
0x1800 0000
...
0x1FFF FFFF
Up to 128GB of external memory mapped space
Supervisory Flash (Sflash)
0x1600 0000 Row 0
... Row 3
System
0x1600 0800 / 0x1600 09FF Row 4
0x1600 0A00 / 0x1600 0BFF Row 5
0x1600 0C00 / 0x1600 0DFF Row 6
0x1600 0E00 / 0x1600 0FFF Row 7
User area
Row 8 ... Row 12
System
0x1600 1A00 / 0x1600 1BFF Row 13
NAR
Row 14 ... Row 44
System
0x1600 5A00 / 0x1600 5BFF Row 45
0x1600 5C00 / 0x1600 5DFF Row 46
...
0x1600 6400 / 0x1600 65FF Row 50
Public Key
Row 51 ... Row 61
System
0x1600 7C00 / 0x1600 7DFF Row 62
TOC2
0x1600 7E00 / 0x1600 7FFF Row 63
RTOC2
512 bytes
0x1600 0800 / 0x1600 0FFF User area — 32K available for firmware usage (keys/ids)
0x1600 5A00 / 0x1600 19FF NORMAL Access Restrictions (NAR) — Used for chip protection in NORMAL state
0x1600 1A00 / 0x1600 65FF Public Key — Used for digital signature of the application
0x1600 7C00 / 0x1600 7DFF Table Of Contents Part 2 (TOC2) — Used to locate OEM objects
0x1600 7E00 / 0x1600 7FFF Reserved Table Of Contents Part 2 (RTOC2) — Used to locate OEM objects
eFuse
0x000 Macro 0, Byte 0
... Macro 0, Byte 19
System
0x014 Macro 0, Byte 20
...
0x023 Macro 1, Byte 3
Secure HASH
0x024 Macro 1, Byte 4
0x025 Macro 1, Byte 5
System
0x026 Macro 1, Byte 6
HASH Zeros
0x027 Macro 1, Byte 7
0x028 Macro 1, Byte 8
DAR
0x029 Macro 1, Byte 9
0x02A Macro 1, Byte 10
SAR
0x02B Macro 1, Byte 11
Life-Cycle
0x02C Macro 1, Byte 12
...
0x03F Macro 1, Byte 31
System
0x040 Macro 2, Byte 0
...
0x050 Macro 2, Byte 16
0x051 Macro 2, Byte 17
...
0x07F Macro 3, Byte 31
Customer Data (PSOC6A-BLE2)*
System
Customer Data (PSOC6A-512K, PSOC6A-2M)*
0x080 Macro 4, Byte 0
...
0x1FF Macro 15, Byte 31
System
1 byte
Macro Size is 32 bytes
SECURE HASH — Secure objects 128 bit HASH
SECURE HASH Zeros — Number of zeros in SECURE HASH
0x027 / 0x028 DEAD Access Restrictions (DAR) — Access restrictions applied in DEAD stage
0x029 / 0x02A SECURE Access Restrictions (SAR) — Access restrictions in SECURE stage
Life Cycle Stage — Silicon Life Cycle stage
0x040* / 0x051 / 0x07F Customer Data — Can be used for application or security purposes. Offset for PSOC6A-BLE2 devices - 0x040; for PSOC6A-512K and PSOC6A-2M (rev. >= A1) - 0x051

Figure 2 Nonvolatile subsystem

2.1 Application flash

The application flash is organized into sectors. Sector size can be either 128 KB for parts with 256 KB flash or 256 KB for all the rest. There are eight such sectors for parts with 2 MB flash on board, four and two sectors for parts with 1 MB and 256/512 KB flash, respectively. There are 256 or 512 rows in the sector, depending on the sector size (128 KB or 256 KB), each consisting of 512 bytes.

The programming granularity is one row at a time. The maximum number of rows during programming depends only on the part's flash size. The formulae are as follows:

$$L = 512 \quad \text{(L is the row size in bytes)}$$

$$N = \frac{FlashSize}{L} \quad \text{(N is the total number of rows)}$$

The flash memory is mapped directly to the CPU's address space starting from 0x10000000. Therefore, the firmware or external programmer can read its content directly from the given address.

2.2 Auxiliary flash (AUXflash)

In addition to the application flash, the flash macros may contain auxiliary flash. The AUXflash is typically used to store frequently updated data; for example, AUXflash can be used to emulate EEPROM memory.

The AUXflash is mapped directly to the CPU's address space starting from 0x14000000. Therefore, the firmware or external programmer can read its content directly from the given address.

Some devices may not have AUXflash, check its availability in the device datasheet.

2.3 Supervisory flash (Sflash)

In addition to the application flash and AUXflash regions, the flash macros contain supervisory flash (Sflash), which can store various application-specific data.

The Sflash memory is mapped directly to the CPU's address space; therefore, the firmware or external programmer can read its content directly from the given address.

These sub-regions are accessible in Sflash memory:

- **0x1600 0800 - 0x1600 0FFF** - User area. Up to 32 KB can be used by the application to store arbitrary data
- **0x1600 1A00 - 0x1600 1BFF** - NORMAL access restrictions (NAR)

Used for chip protection in the NORMAL lifecycle stage. Ensure that the NAR sub-region cannot be overwritten or erased if the new data is less restrictive than the existing data.

- **0x1600 5A00 - 0x1600 65FF** - Public Key. Used for digital signature of the application.
- **0x1600 7C00 - 0x1600 7DFF** - Table of contents part 2 (TOC2).

Used to locate various OEM objects such as application(s) start address(es) and format, address of SMIF configuration structure. Also used during the boot process to apply device's initial settings such as clock frequency configuration, duration of the Listen window, SWJ pins availability, and digital signature verification (authentication).

- **0x1600 7E00 - 0x1600 7FFF** - Reserved table of contents part 2 (RTOC2)

Writing to listed sub-regions is not possible when the chip is in the SECURE lifecycle stage. Writing to any Sflash address outside the specified sub-regions is not possible in any Life Cycle stage except VIRGIN, which is a factory-only stage.

2.4 Electronic fuses (eFuse)

PSOC™ 6 MCUs contain Electronic Fuses (eFuses), which include up to 16 nonvolatile memory macros of 256 bits each (4096 bits in total), with each bit being one-time programmable (OTP). These are implemented as a regular advanced high-performance Bus (AHB) peripheral with the following characteristics and assumptions:

- eFuses store the device lifecycle stage (NORMAL, SECURE, and SECURE_WITH_DEBUG), the protection settings, and up to 512 bits (64 bytes) of customer data
- eFuse memory can be programmed (eFuse bit value changed from ‘0’ to ‘1’) only once. If an eFuse bit is blown, it cannot be cleared
- Programming fuses require the associated I/O supply to be at a specific level: the device VDDIO0 (or VDDIO if only one VDDIO is present in the package) supply should be set to 2.5 V (±5%)
- The eFuse array can be read eight bits at a time using normal memory-mapped AHB register reads or corresponding system calls
- eFuses are programmed one bit at a time using a Command register

These eFuse bytes are accessible for production programming:

- **0x014 - 0x023** - SECURE HASH. 128-bit (16 bytes) HASH is used by boot code to authenticate objects in the table of contents part 2 (TOC2).
- **0x26** - SECURE HASH Zeros. The number of bits that are ‘0’ (fuses that are not blown) in the SECURE HASH above. This guarantees that once a HASH is programmed, it cannot be changed into another valid HASH value
- **0x027 - 0x028** - DEAD access restrictions (DAR). Chip access restrictions applied in the DEAD lifecycle stage:
- **0x029 - 0x02A** - SECURE access restrictions (SAR). Chip access restrictions applied in the SECURE lifecycle stage
- **0x02B** - Silicon lifecycle stage (NORMAL, SECURE_WITH_DEBUG, or SECURE)
- **0x040 (0x051) - 0x07F** - Customer data. It can be used by customers for application or security purposes. Offset for PSOC6A-BLE2 devices - 0x040; for PSOC6A-256K, PSOC6A-512K, and PSOC6A-2M (rev. >= A1) - 0x051

Because blowing an eFuse is an irreversible process, programming is recommended only in mass production programming under controlled factory conditions and not prototyping stages. See Appendix B for eFuse data mapping in the data file. For more details, see the “eFuse Memory” section of the reference manual.

2.5 Execute-in-place (XIP)

Unlike other memory regions, the XIP region is not associated with any physical memory in PSOC™ 6 MCUs. The purpose of the XIP region is to map the address space of the external memory devices, which are connected to PSOC™ 6 MCU silicon using the SMIF IP block. When the SMIF block is configured in XIP/Memory mode, it maps the AHB bus accesses to the external memory device addresses to make it behave like internal memory. This allows the CPU to execute code directly from external memory or use it as additional data storage.

Programming of the external flash memory devices via the SMIF IP block can be supported using a flash loader. A flash loader is an application compiled for a target CPU that implements programming algorithms and follows specific rules (framework) defined by a third-party IDE like Keil µVision, where CMSIS-based flash loaders are used. Such algorithms are loaded into target SRAM by programming software and executed from there for memory bank programming. Infineon supports such algorithms for third-party development tools like Keil µVision (MDK-ARM), IAR Embedded Workbench, and SEGGER J-Link Software and Documentation Pack. Flash loaders developed by Infineon include the system and SMIF driver middleware and have the following requirements:

- Configuration structures for the SMIF driver must be in the application flash. The Loader uses these structures to determine the design-specific settings of the external memory devices. It also uses these structures for the SMIF driver initialization for further read/write operations.
- The TOC2 must contain the pointer to the location in flash where the configuration structures for the SMIF driver are located. TOC2 is the predefined 512-byte-wide data structure, located at address 0x1600 6C00 in the Sflash region.
- An external flash memory device must be write-enabled and mapped to the XIP address space in PSOC™ MCUs (within the address range 0x1800 0000 - 0x1FFF FFFF).
- Data for the external memory device in the input binary file (hex, elf, and so on) must be allocated in the exact range of XIP address space, where the memory device is mapped.

These requirements necessitate a specific order of memory bank operations:

- For program operation, the application and supervisory flash banks must be programmed first, before the SMIF bank is programmed
- For the erase operation, the SMIF bank must be erased first, before the application and supervisory flash banks are erased

3 Hex file

This chapter describes the information that the programmer must extract from the hex file to program the PSOC™ 6 MCU.

3.1 Organization of the hex file

The hexadecimal (hex) file describes the nonvolatile configuration of the project. It is the data source for the programmer.

The hex file for the PSOC™ 6 MCUs follows the Intel Hex File format. Intel's specification is very generic and defines only some types of records that can make up the hex file. The specification allows customizing the format for any possible silicon architecture. The silicon vendor defines the functional meaning of the records, which typically varies for different chip families. See Appendix A for details of the Intel hex file format.

The PSOC™ 6 MCU defines these data sections in the hex file:

- User program (code) for the application flash region
- User data for the AUXflash region
- User / OEM data for the Sflash region
- User data or code for the external flash memory
- Checksum
- Metadata
- Chip-level protection data (eFuses)

See Figure 3 to determine the allocation of these sections in the address space of the Intel hex file.

The address space of the hex file does not map to the physical addresses of the CPU (other than the user flash). The programmer uses hex addresses (see Figure 3) to read sections from the hex file into its local buffer. Later, this data is programmed (translated) into the corresponding addresses of the silicon.

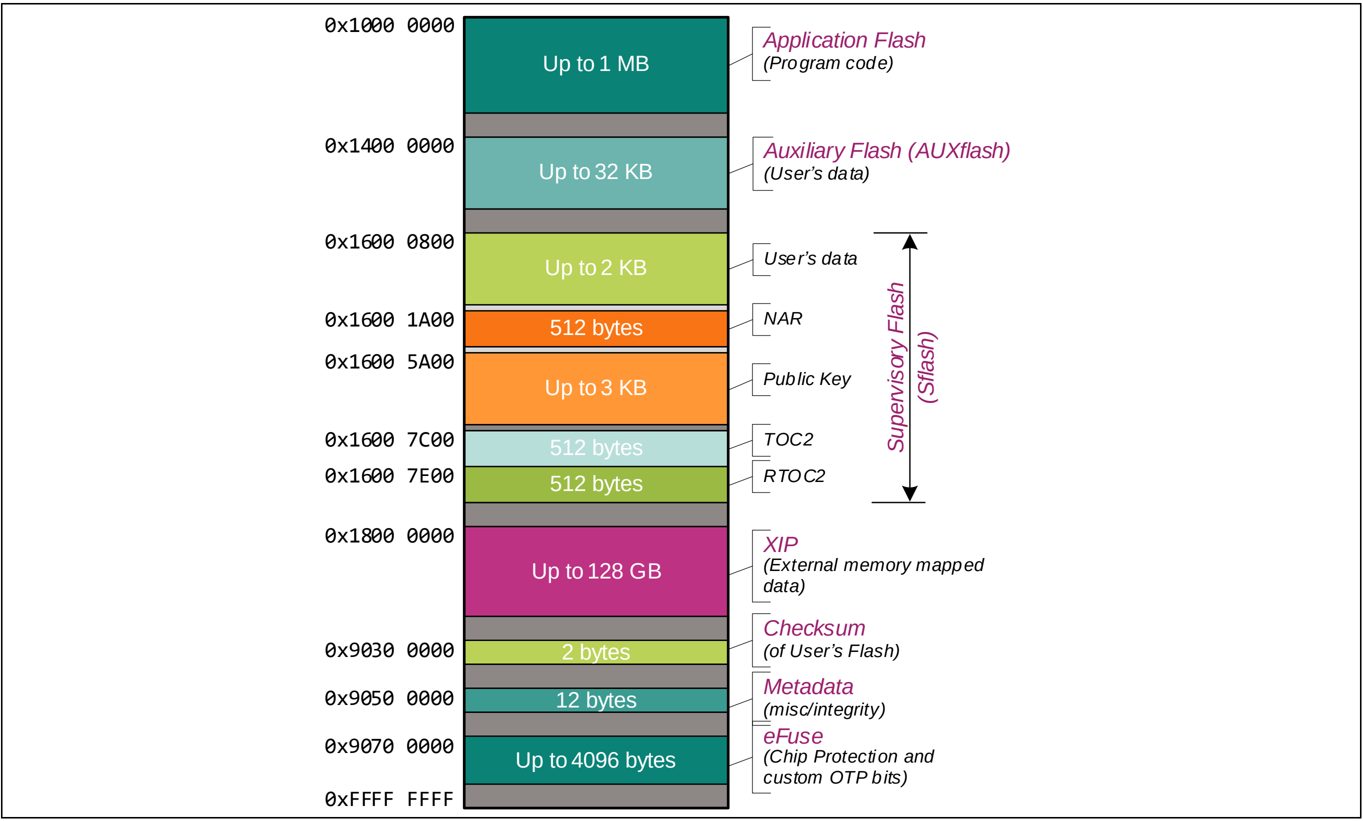


Figure 3 **Hex file organization for PSOC™ 6 MCUs**

- **0x1000 0000 – application flash**: This is the user code that must be programmed. The maximum size of this section must not exceed the flash size of the PSOC™ 6 MCU (up to 2 MB). The address space of this section in the hex file is directly mapped to the physical addresses of the CPU. See the section Application flash for the region description.
- **0x1400 0000 –AUXflash**: Can store up to 32 KB of application-specific information. The address space of the AUXflash section in the hex file is directly mapped to the physical addresses of the CPU. Availability of this section in hex file is optional and depends on linker scripts usage in user's project. See the section Auxiliary flash (AUXflash) for the region description.
- **0x1600 0000 –Sflash**: Can store application-specific information in five fragmented sub-sections. The address space of the Sflash section in the hex file is directly mapped to the physical addresses of the CPU. Availability of this section in hex file is optional and depends on linker scripts usage in user's project. See Supervisory flash (Sflash) for the region description.
- **0x1800 0000 –XIP**: Can be used to store up to 128 MB of the external memory-mapped data. The address space of the XIP section in the hex file is mapped to the physical addresses of the CPU only in case the user's program configures SMIF block in XIP/Memory mode. Availability of this section in hex file is optional and depends on linker scripts usage in user's project. Seethe section Execute-in-place (XIP) for the region description.
- **0x9030 0000 – Checksum** (two bytes): This is the checksum of the entire application flash section—the arithmetical sum of every byte in the user's flash. Only the two least significant bytes (LSB) of the result are saved in this section, in big-endian format (most significant byte, or MSB, first). This must be used by the programmer to check the integrity of the hex file and to verify the quality of the programming. In this context, "integrity" means that the checksum and user's flash sections must correspond in this file. At the end of programming, the checksum of flash (two LSBs) is compared to the checksum from the hex file.

- **0x9050 0000 – Metadata** (12 bytes): This section contains data that is not programmed into the PSOC™ 6 MCU. Instead, it is used to check the data integrity of the hex file and the silicon ID of the PSOC™ 6 MCU. Table 1 lists the fields in this section.

Table 1 Metadata in hex file

Offset	Data type	Length in bytes
0x00	Hex file version	2 (big-endian)
0x02	Silicon ID	4 (big-endian)
0x06	Reserved	1
0x07	Reserved	1
0x08	Internal use	4

  - **Hex file version**: This 2-byte field in the Infineon hex file defines its version (or type). The version for the PSOC™ 6 MCU is '5'. The programmer should use this field to ensure the file corresponds to the PSOC™ 6 MCU, or to select the appropriate parsing algorithm if the file supports several families.
  - **Silicon ID**: This 4-byte field (big-endian) represents the ID of the PSOC™ 6 MCU silicon:
    - byte[0] - Silicon ID Hi
    - byte[1] - Silicon ID Lo
    - byte[2] - Revision ID
    - byte[3] - Family ID

During programming, the ID of the acquired device is compared to the content of this field. To start programming, three of these fields must match. The Revision ID must be skipped because it is not essential for programming—there are many silicon revisions possible that do not change functionality. Infineon does not guarantee reliable programming (or data retention) if third-party programmers ignore this condition.

  - **Reserved**: Not used by the PSOC™ 6 MCU
  - **Internal Use**: This 4-byte field is used internally by the PSOC™ Programmer software. Because it is not related to actual programming, this field should be ignored by third-party vendors' tools.

- **0x9070 0000 – eFuse** (up to 4096 bytes): eFuse memory provides security functions with far more flexibility than exist in the Flash Protection, Write Once NVL, and Chip Protection options in prior devices. Each eFuse bit setting is stored in the hex file as a full byte. This is done for two reasons. First, it allows the programmer to distinguish bytes that are being set from bytes that we don't care about, or where we don't know the value. The second reason is that it more accurately reflects how these bits are programmed: the SROM functions sets one bit per call. The values are: 0x00 – Not blown; 0x01 – Blown; 0xFF – Ignore. Note: The programmer can only perform a "not blown" to "blown" operation. The programmer should read the corresponding eFuse bit from the device first and blow it only if the device value is '0' (not blown) and the hex value is 0x01 (blown). See the device datasheet for the number of eFuse bits available on the device. See Appendix B for eFuse data mapping in the hex file.

4 Protocol stack

This chapter explains the low-level details of the communication interface. Figure 4 illustrates the stack of protocols involved in the programming process. The programmer must implement both hardware and software components.

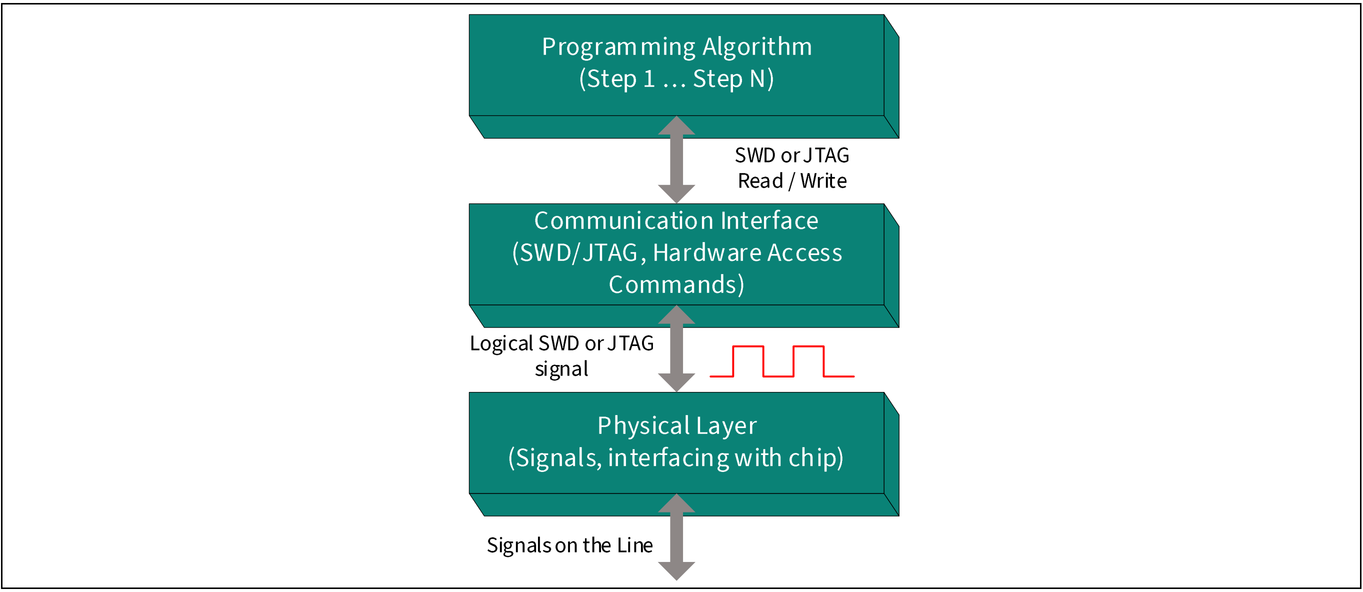


Figure 4 Programmer protocol stack

- **Programming algorithm** protocol, the topmost protocol, implements the whole programming flow in terms of logical and algorithmic steps. This protocol is implemented completely in software. Its smallest building block is the SWD or JTAG command. The whole programming algorithm is the meaningful flow of these blocks.
  - All programming algorithms are based on system API, stored in SROM (SROM API). During the programming of the flash row, the system code is executed from the SROM. It communicates with the Inter Processor Communication (IPC) module, which “knows” how to program flash. In contrast to a write operation, reading from flash is an immediate operation that is carried out directly from the necessary address (see Figure 2 for address space). Reading works on a word basis (4-byte); writing works on a row basis (512-byte).
  - The Programming algorithm protocol is the fundamental part of this specification. For more information on this algorithm, see Chapter 5: Programming Algorithm.
- **Communication interface** layer acts as a bridge between pure software and hardware implementations. SWJ interface implements a set of lower-level (protocol-dependent) commands. It also transforms the software representation of these commands into line signals (digital form). The SWJ interface helps to isolate the programming algorithm from hardware specifics, which makes the algorithm reusable.
- **Physical layer** is the complete hardware specification of the signals and interfacing pins and includes drive modes, voltage levels, resistance, and other components.

4.1 Communication interface

The external device (whether it is an Infineon-supplied programmer and debugger or a third-party device that supports programming and debugging) can access most internal resources through the Program and Debug Interface provided in PSOC™ 6 MCU silicon. The Serial Wire Debug (SWD) or the JTAG interface can be the communication protocol between the external device and the PSOC™ 6 MCU.

4.2 Program and debug interface

The main purpose of PSOC™ 6 MCU program and debug interface is to support programming and debugging through the JTAG or SWD interface and to provide read and write access to all memory and registers in the system while debugging, including the Cortex®-M4 and Cortex®-M0+ register banks when the core is running or halted.

The PSOC™ 6 MCU silicon implements a debug access port (DAP), which integrates Serial Wire/JTAG Debug Port (SWJ-DP) and complies with the Arm® specification. For more information, see [1].

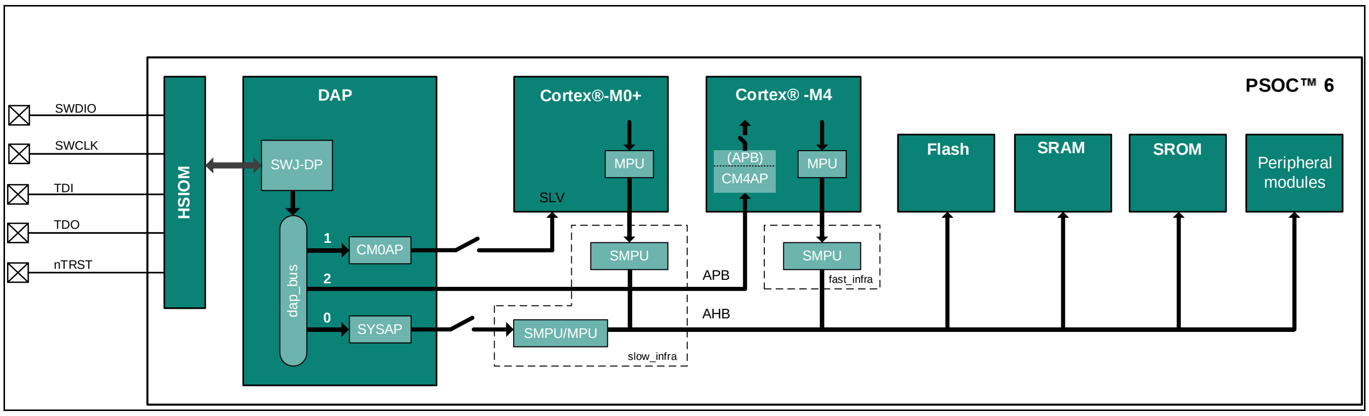


Figure 5 **Top-level silicon architecture**

The debug physical port pins communicate with the DAP through the high-speed I/O matrix (HSIOM). The DAP communicates with the Cortex®-M0+ CPU using the Arm®specified advanced high-performance bus (AHB) interface. AHB is the systems interconnect protocol used inside the device, which facilitates memory and peripheral register access by the AHB master. The PSOC™ 6 MCU has several AHB masters, including the Arm® CM4 CPU core, Arm® CM0+ CPU core, and DAP. The external host can effectively take control of the entire device through the DAP to perform programming and debugging operations.

The debug port (DP) connects to the DAP bus, which in turn connects to one of three access ports (AP), namely:

- **CM0-AP** connects directly to the AHB debug slave port (SLV) of the CM0+ and gives access to the CM0+ internal debug components. This also allows access to the rest of the system through the CM0+ AHB master interface. This provides the debug host the same view as an application running on the CM0+. This includes access to the MMIO registers of other debug components of the Cortex® M0+ subsystem. These debug components can also be accessed by the CM0+ CPU but cannot be reached through the other APs or by the CM4 core.
- **CM4-AP** is located inside the CM4 gives access to the CM4 internal debug components. The CM4-AP also allows access to the rest of the system through the CM4 AHB master interfaces. This provides the debug host the same view as an application running on the CM4 core. Additionally, the CM4-AP provides access to the debug components in the CM4 core through the external peripheral bus (EPB). These debug components can also be accessed by the CM4 CPU but cannot be reached through the other APs or by the CM0+ core.

- **System-AP** gives access to the rest of the system. This allows access to the System ROM table, which cannot be reached any other way. The System ROM table provides the MCU ID.

4.2.1 DAP security

For security reasons, all three APs can be independently disabled. Each AP disable is controlled by two MMIO bits. One bit, CPUSS_DP_CTL.xxx_ENABLE (where xxx can be CM0 or CM4, or SYS), is a regular read/write bit. This bit also resets to '0' and is set to '1' by either the ROM boot code or the flash boot code, depending on the lifecycle stage. This feature can be used to block debug access during normal operation but re-enable some debug access after successful authentication.

The second bit, CPUSS_DP_CTL.xxx_DISABLE, can be set during boot, before the debugger can connect, based on eFuse settings for SECURE or DEAD lifecycle stage or based on NAR settings in Supervisory flash for NORMAL lifecycle. After this bit is set, it cannot be cleared. This bit takes priority over the CPUSS_DP_CTL.xxx_ENABLE.

In addition, debug privileges are regulated by the platform protection mechanism using the memory protection units (MPUs), shared memory protection units (SMPUs), and peripheral protection units (PPUs).

Refer to the "Device Security" and "Protection Units" sections of the reference manual for more details on the security settings for the PSOC™ 6 MCU.

4.2.2 DAP power domain

Almost all the debug components are part of the Active power domain. The only exception is the SWD/JTAG-DP, which is part of the Deep Sleep power domain. This allows the debug host to connect during Deep Sleep mode, while the application is 'running' or powered down. This enables in field debugging for low-power applications in which the chip is mostly in Deep Sleep mode.

After the debugger is connected to the chip, it must bring the chip to the Active state before any operation. For this, the SWD/JTAG-DP has a register (DP_CTL_STAT) with two power request bits. The two bits are CDBGPWRUPREQ and CSYSPWRUPREQ, which request debug power and system power, respectively. These bits must remain set for the duration of the debug session.

Note that only the two SWD pins (SWCLKTCK and SWDIOTMS) are operational during the Deep Sleep mode – the JTAG pins are operational only in Active mode. The JTAG debug and JTAG boundary scan is not available when the system is in Deep Sleep mode.

4.2.3 SWD/JTAG selection

The JTAG and SWD are mutually exclusive because of Arm® SWJ-DP implementation and because they share pins. Therefore, an external programmer/debugger must be able to switch to the required protocol. The watcher circuit, implemented in SWJ-DP, detects a specific 16-bit select sequence on SWDIOTMS and decides if the JTAG or SWD interface is active. By default, JTAG operations are selected on powerup reset and therefore the JTAG protocol can be used from reset without sending a select sequence. The protocol switching can only occur when the selected interface is in its reset state (test-logic-reset for JTAG and line-reset for SWD).

To switch SWJ-DP from JTAG to SWD operation:

- Send at least 50 SWCLKTCK cycles with SWDIOTMS HIGH. This ensures that the current interface is in its reset state. The JTAG interface detects only the 16-bit JTAG-to-SWD sequence starting from the test-logic-reset state.

- Send the 16-bit JTAG-to-SWD select sequence on SWDIOTMS: 0b0111 1001 1110 0111, most significant bit (MSb) first. This can be represented as 0x79E7, transmitted MSB first or 0xE79E, transmitted least significant bit (LSb) first.
- Send at least 50 SWCLKTCK cycles with SWDIOTMS HIGH. This ensures that if SWJ-DP was already in SWD operation before sending the select sequence, the SWD interface enters line reset state.

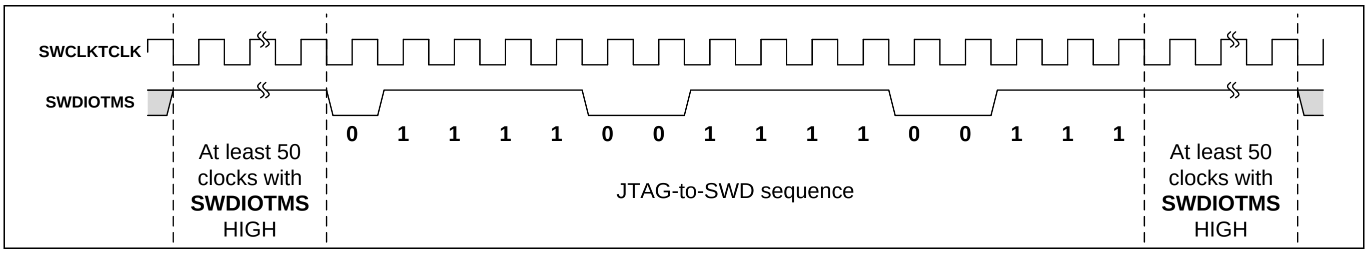

SWCLKTCLK
SWDIOTMS
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1
At least 50 clocks with SWDIOTMS HIGH
JTAG-to-SWD sequence
At least 50 clocks with SWDIOTMS HIGH

Figure 6 JTAG-to-SWD sequence timing

To switch SWJ-DP from SWD to JTAG operation:

- Send at least 50 SWCLKTCK cycles with SWDIOTMS HIGH. This ensures that the current interface is in its reset state. The SWD interface detects the 16-bit SWD-to-JTAG sequence only when it is in the reset state.
- Send the 16-bit SWD-to-JTAG select sequence on SWDIOTMS: 0b0011 1100 1110 0111, MSb first. This can be represented as 0x3CE7, transmitted MSb first, or 0xE73C, transmitted LSb first.
- Send at least five SWCLKTCK cycles with SWDIOTMS HIGH. This ensures that if SWJ-DP was already in JTAG operation before sending the select sequence, the JTAG TAP enters the test-logic-reset state.

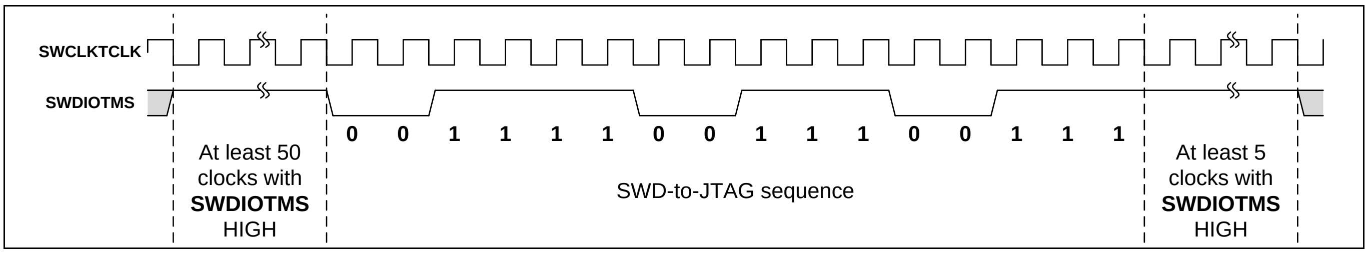

SWCLKTCLK
SWDIOTMS
0 0 1 1 1 1 0 0 1 1 1 0 0 1 1 1
At least 50 clocks with SWDIOTMS HIGH
SWD-to-JTAG sequence
At least 5 clocks with SWDIOTMS HIGH

Figure 7 SWD-to-JTAG sequence timing

For a more detailed description, see the “SWD and JTAG select mechanism” section of [1].

4.2.4 Hardware access commands

The SWJ-DP supports several types of transactions: Interface selection, Target Selection, Read, Write, and Port Reset. All are defined in the Arm® specification. The APIs must be implemented by the Communication Interface layer shown in Figure 4 . In addition, the upper protocol, Programming Algorithm, requires two extra commands to manipulate the hardware: Power(state) and ToggleReset(). Table 2 lists the hardware access commands used by the software layer.

Table 2 Hardware access commands

Command	Parameters	Description
DAP_JTAGtoSWD	–	Standard Arm® command to switch SWJ-DP from JTAG to SWD operations. This sequence synchronizes the programmer and chip; it is the first transaction in programming flow if SWD protocol is used. See SWD/JTAG selection for implementation details.
DAP_SWDtoJTAG	–	Standard Arm® command to switch SWJ-DP from SWD to JTAG operations. This sequence synchronizes the programmer and chip; it is the first transaction in the programming flow if the JTAG protocol is used. See SWD/JTAG selection for implementation details.
SWD_Write	IN APnDP, IN addr, IN data32, OUT ack	Sends 32-bit data to the specified register of the DAP using the SWD interface. The register is defined by the “APnDP” (1-bit) and “addr” (2-bit) parameters. The DAP returns a 3-bit status in “ack”.
SWD_Read	IN APnDP, IN addr, OUT data32, OUT ack, OUT parity	Reads 32-bit data from the specified register of the DAP using the SWD interface. The register is defined by the “APnDP” (1-bit) and “addr” (2-bit) parameters. DAP returns a 32-bit data, status, and parity (control) bit of the read 32-bit word.
JTAG_Write	IN APnDP, IN addr, IN data32, OUT ack	Sends 32-bit data to the specified register of the DAP using the JTAG interface. The register is defined by the “APnDP” (1-bit) and “addr” (2-bit) parameters. The DAP returns a 3-bit status in “ack”.
JTAG_Read	IN APnDP, IN addr, OUT data32, OUT ack	Reads 32-bit data from the specified register of the DAP using JTAG interface. The register is defined by the “APnDP” (1-bit) and “addr” (2-bit) parameters. DAP returns a 32-bit data and status.
ToggleReset	–	Generates the reset signal for PSOC™ 6 MCU. The programmer must have a dedicated pin connected to the XRES pin of the PSOC™ 6 MCU.
Power	IN state	If the programmer powers the PSOC™ 6 MCU, it must have this function to supply power to the device.

For information on the structure of the SWD read and write packets and their waveform on the bus, see Appendix C: Serial wire debug (SWD) Protocol. For information on the structure of the JTAG, see Appendix D: Joint test action group (JTAG) protocol.

The SWJ Read/Write commands allow accessing registers of the SWJ-DP module from Figure 6. The DAP functionally is split into two control units:

- Debug port (DP) – Is responsible for the physical connection to the programmer or debugger.
- Access port (AP) – Provides the interface between the DAP module and one or more debug components (such as the Cortex®-M0+ CPU).

The external programmer can access the registers of these access ports using the following bits in the SWJ packet:

- APnDP: Select access port (0 – DP, 1 - AP)
- ADDR: 2-bit field addressing a register in the selected access port

The SWJ Read/Write commands are used to access these registers. They are the smallest transactions that can appear on the SWJ bus. Table 3 shows the DAP registers that are used during programming.

Table 3 DAP registers (in Arm® notation)

Register	APnDP (1-bit)	Address (2-bit)	Access (R/W)	Full name
IDCODE	0	2'b00	R	Identification Code Register
ABORT	0	2'b00	W	AP ABORT Register
CTRL/STAT	0	2'b01	R/W	Control/Status Register
SELECT	0	2'b10	W	AP Select Register
RDBUFF	0	2'b11	R	Data buffer register
CSW	1	2'b00	R/W	Control Status/Word Register (CSW)
TAR	1	2'b01	R/W	Transfer Address Register
DRW	1	2'b11	R/W	Data Read/Write Register

For more information about these registers, see [1].

4.2.5 Pseudocode

This document uses an easy-to-read C-style pseudocode to show the programming algorithm. Pseudocode does not include low-level algorithmic details such as variable definitions or error handling. Ensure to implement error handling in the final code, which is typically to stop programming and return failure status if any programming step fails.

The following two commands are used for the programming script:

```
Write_DAP  (Register, data32)
Read_DAP   (Register, OUT data32)
```

Where the Register parameter is an AP/DP register defined by APnDP and address bits (see Table 3). The pseudo commands correspond to read or write SWJ transactions. Following are some examples:

```
Write_DAP (TAR, 0x08000000)
Write_DAP (DRW, 0x12345678)
Read_DAP (IDCODE, OUT swd_id)
```

The Register parameter technically can be represented as a structure in C:

```
struct DAP_Register {
    BYTE APnDP; // 1-bit field
    BYTE Addr;  // 2-bit field
};
```

Then, DAP registers will be defined as:

```
DAP_Register  TAR     = {1, 1},
              DRW     = {1, 3},
              IDCODE  = {0, 0};
```

The defined Write and Read pseudo commands are successful if both return the ACK status of the SWJ transaction. For the Read transaction, the parity bit must be considered (corresponds to the read data32 value). If the status of the transaction or the parity bit is (or both are) incorrect, the transaction has failed. In this case, depending on the programming context, programming must terminate, or the transaction must be tried again.

The implementation of Write and Read pseudo commands based on the hardware access commands SWJ Read and Write (Table 2) is as follows.

```
//- DAP Read/Write subroutines --------------------------------------------------
SWJ_Status Write_DAP (Register, data32) {
  if (Interface == SWD)
    SWD_Write (Register.APnDP, Register.Addr, data32, OUT ack);
  else // Interface == JTAG
    JTAG_Write (Register.APnDP, Register.Addr, data32, OUT ack);
  return ack;
}

SWJ_Status Read_DAP (Register, OUT data32) {
  ack = ACK_FAIL;
  if (Interface == SWD) {
    SWD_Read (Register.APnDP, Register.Addr, OUT data32, OUT ack, OUT parity);
   if (ack == 3'b001) { //ACK, then check the parity bit as well
      parityData32 = 0x00;
      for (i = 0; i < 32; i++)
        parityData32 ^= ((data32 >> i) & 0x01);
      if (parityData32 != parity)
        ack = 3'b111; //NACK
    }
  }
  else if (Interface == JTAG)
    JTAG_Read(Register.APnDP, Register.Addr, OUT data32, OUT ack);

  return ack;
}
```

The programming code in Programming algorithm is based mostly on the Write and Read pseudo commands and some commands in Table 2.

4.3 Physical layer

This section summarizes the hardware connection between the programmer and the PSOC™ 6 MCU for programming. Figure 8 shows the generic connection between the PSOC™ 6 MCU and the programmer. See Table 4 for pins/signals description.

See the device datasheet for the part's package pins location, electrical, and timing specifications.

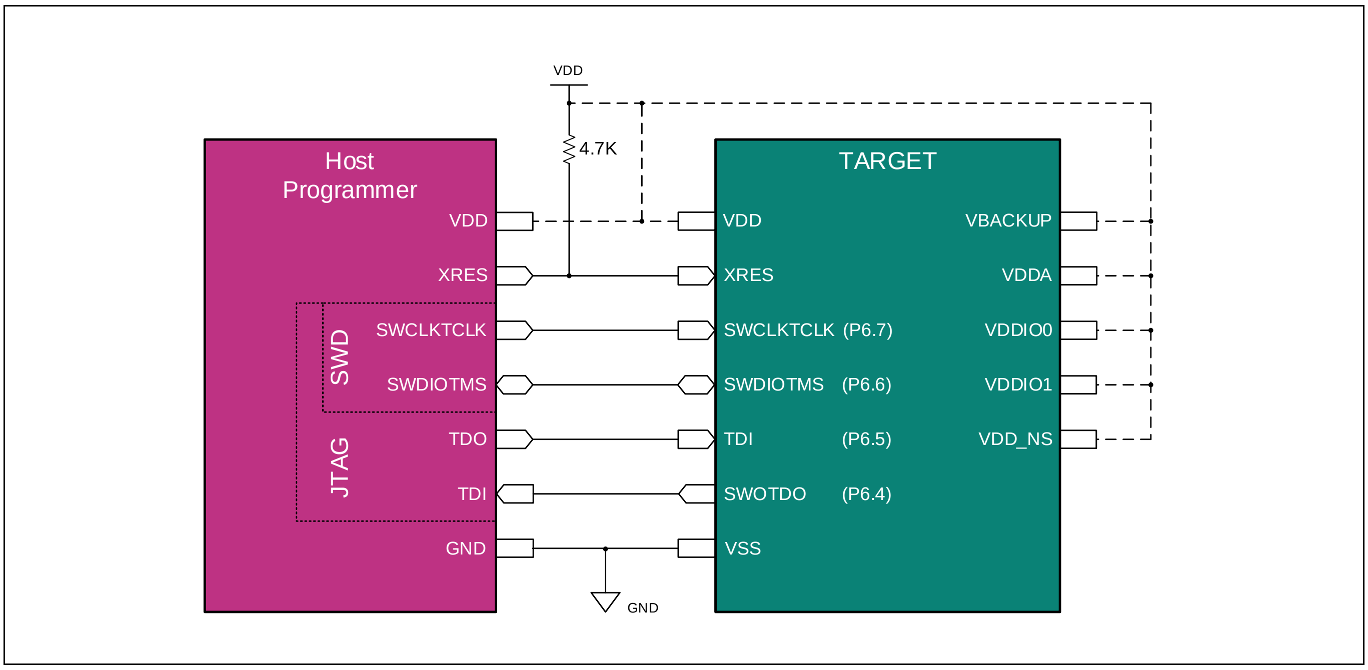


Figure 8 **Connection schematic of programmer**

Table 4 Pins/signals description

Pin	SWD		JTAG		Description
	Signal name	**Mandatory**	**Signal name**	**Mandatory**	
SWCLKTCLK	Serial wire clock (SWCLK)	Yes	Test clock (TCLK)	Yes	Data synchronization clock, driven by the host programmer/debugger. Although the Arm® specification does not define the minimum frequency of the SWD bus, the minimum for the PSOC™ 6 MCU family is 1.5 MHz. It is needed only on the first step to acquire the silicon during the boot window. After that, programming frequency can be as low as needed. For SWD, the host should perform all read or write operations on the SWDIO line on the falling edge of SWDCK. The PSOC™ 6 MCU performs read or write operations on SWDIO on the rising edge of SWDCK. For JTAG, the host writes to the TMS and TDI pins of the PSOC™ 6 MCU on the falling edge of TCK and the PSOC™ 6 MCU reads data on its TMS and TDI lines on the rising edge of TCK. PSOC™ 6 MCU writes to its TDO line on the falling edge of TCK and the host reads from the TDO line of the PSOC™ 6 MCU on the rising edge of TCK.
SWDIOTMS	Serial wire data input/ output (SWDIO)	Yes	Test mode select (TMS)	Yes	SWDIO is a bidirectional data input/output signal. TMS is the JTAG Test Mode Select signal, which is sampled at the rising edge of TCK to determine the next state.
SWOTDO	Serial wire output (SWO)	No	Test data out (TDO)	Yes	SWO signal (also known as TRACESWO) is required for Serial Wire Viewer (SWV) and not required for SWD programming. It provides real-time data trace information from the PSOC™ 6

Pin	SWD		JTAG		Description
	Signal name	Mandatory	Signal name	Mandatory	
					MCU, via the SWO pin, while the CPU continues to run at full speed. Data trace via SWV is not available using the JTAG interface. TDO signal represents the data shifted out of the device's test or programming logic and is valid on the falling edge of TCK when the internal state machine is in the correct state.
TDI	–	–	Test data in (TDI)	Yes	TDI signal represents the data shifted into the device's test or programming logic. It is sampled at the rising edge of TCK.
XRES	External reset (XRES)	NO [1]	Reset (XRES)	No [1]	External reset active LOW signal. The XRES is not related to the ARM standard. It is used to reset the part as a first step in a programming flow. *Note: XRES pin/signal is not TRST (Test Reset) signal for JTAG Interface, which is the optional pin that asynchronously resets only the JTAG test logic.*
GND	Ground (GND)	Yes	Ground (GND)	Yes	Negative supply voltage (ground)
VDD	Voltage drain drain (VDD)	No [2]	Voltage drain drain (VDD)	No [2]	Positive supply voltage. The PSOC™ 6 MCU can be powered by external power supply or by programmer.

1 XRES pin is mandatory for "Reset" PSOC™ 6 MCU Acquisition mode, but not used for "Power Cycle" mode.

2 VDD pin is mandatory for "Power Cycle" PSOC™ 6 MCU Acquisition mode, where programmer powers the PSOC™ 6 MCU and external power is not applied. For "Reset" Acquisition mode, the source of power supplier does not matter, so the pin is optional.

You can program a chip in either Reset (recommended) or Power Cycle mode. The mode affects only the first step - how to reset the part at the start of the programming flow. All other steps are the same.

- **Reset mode**: To start programming, the host toggles the XRES line and then sends SWD/JTAG commands (see Table 2). The power on the PSOC™ 6 MCU board can be supplied by the host or by an external power adapter (the VDD line can be optional).
- **Power Cycle mode**: To start programming, the host powers on the PSOC™ 6 MCU and then starts sending the SW/JTAG commands. The XRES line is not used.

The programmer should implement PSOC™ 6 MCU acquisition in the Reset mode. It is also the only way to acquire the PSOC™ 6 MCU if the board consumes too much current, which the programmer cannot supply. Power Cycle mode support is optional and should be used only the following:

- XRES pin is not available on the part's package, or
- The third-party programmer does not implement the XRES line but can supply power to the PSOC™ 6 MCU.

5 Programming algorithm

This chapter describes in detail the programming flow of the PSOC™ 6 MCU. It starts with a high-level description of the algorithm and then describes every step using pseudocode. All code is based on upper-level subroutines composed of atomic SWJ instructions (see Pseudocode). These subroutines are defined in the section Constants and subroutines used in the programming flow. The ToggleReset() and Power() commands are also used (see Table 2).

5.1 High-level programming flow

Figure 9 shows the sequence of steps that must be executed to program the PSOC™6 MCU. These steps are described in detail in the following sections. All the steps in this programming flow must be completed successfully for a successful programming operation. The programmer should stop the programming flow if any step fails. In addition, in pseudocode, it is assumed that the programmer checks the status of each SWJ transaction (Write_DAP, Read_DAP, WriteIO, ReadIO). This extra code is not shown in the programming script. If any of these transactions fail, then programming must be aborted.

The flash programming in the PSOC™ 6 MCU family is implemented using the SROM API. The external programmer puts the parameters into the SRAM (or registers) and requests system calls, which in turn perform flash updates.

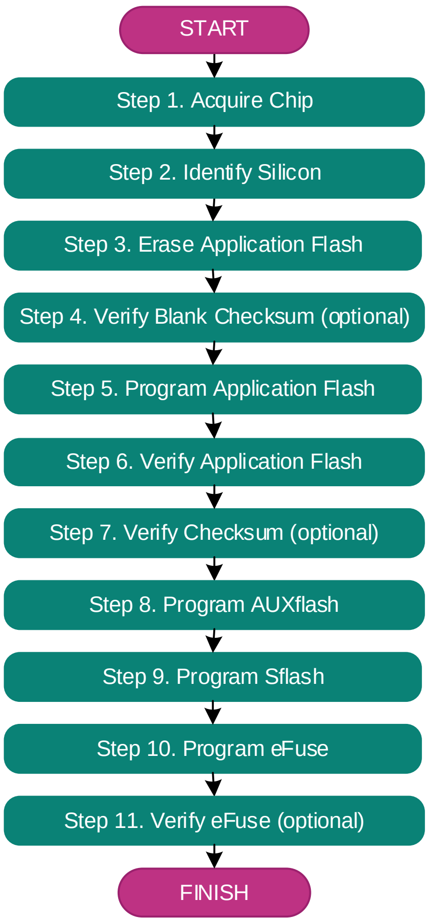

START
Step 1. Acquire Chip
Step 2. Identify Silicon
Step 3. Erase Application Flash
Step 4. Verify Blank Checksum (optional)
Step 5. Program Application Flash
Step 6. Verify Application Flash
Step 7. Verify Checksum (optional)
Step 8. Program AUXflash
Step 9. Program Sflash
Step 10. Program eFuse
Step 11. Verify eFuse (optional)
FINISH

Figure 9 High-level programming flow of PSOC™ 6 MCU

5.2 Constants and subroutines used in the programming flow

To make the pseudocode easier to understand, many registers and frequently used constants are named. The defined symbols are used in the pseudocode. Following is the complete list of constants used in the programming steps:

5.2.1 Constants

```
#if defined(PSOC™6ABLE2)
/* Specific for CY8C6xx6 and CY8C6xx7 devices ---------------------------------*/

#define MEM_SIZE_ROM                       0x00020000 // Size of System ROM
#define MEM_BASE_IPC                       0x40230000 // Base addr. of IPC structures
#define MEM_VTBASE                         0x402102B0 // CM0_VECTOR_TABLE_BASE
#define MEM_BASE_PPU4                      0x40014100 // PPU[4] base address
#define IPC_INTR_STRUCT                    0x40231000 // IPC_INTR structure address
#define IPC_STRUCT_LOCK_STATUS_OFFSET      0x10       // IPC lock status

#elif defined(PSOC6A2M) || defined(PSOC6A512K) || defined(PSOC6A256K)
/* Specific for CY8C6xx4, CY8C6xx5, CY8C6xx8, CY8C6xxA devices ----------------*/

#define MEM_SIZE_ROM                       0x00010000 // Size of System ROM
#define MEM_BASE_IPC                       0x40220000 // Base addr. Of IPC structures
#define MEM_VTBASE                         0x40201120 // CM0_VECTOR_TABLE_BASE
#define MEM_BASE_PPU4                      0x40010100 // PPU[4] base address
#define IPC_INTR_STRUCT                    0x40221000 // IPC_INTR structure address
#define IPC_STRUCT_LOCK_STATUS_OFFSET      0x1C       // IPC lock status
#endif /* End of target-specific constants definition -------------------------*/

#define MEM_BASE_ROM                       0x00000000 // Base of System ROM
#define MEM_BASE_SRAM                      0x08000000 // Base of SRAM
#define MEM_BASE_FLASH                     0x10000000 // Base of application flash
#define MEM_BASE_AUXFLASH                  0x14000000 // Base of auxiliary flash
#define MEM_SIZE_AUXFLASH                  0x00008000 // Size of auxiliary flash
#define MEM_BASE_SFLASH                    0x16000000 // Base of supervisory flash
#define MEM_SIZE_SFLASH                    0x00008000 // Size of supervisory flash

#define IPC_STRUCT_SIZE                    0x20
#define IPC_STRUCT0                        MEM_BASE_IPC // CM0+ IPC_STRUCT
#define IPC_STRUCT_ACQUIRE_OFFSET          0x00 // Used to acquire a lock
#define IPC_STRUCT_NOTIFY_OFFSET           0x08 // Used for Notification events
#define IPC_STRUCT_DATA_OFFSET             0x0C // 32-bit data element
#define IPC_STRUCT_LOCK_STATUS_ACQUIRED_MSK 0x80000000 // Is lock acquired
#define IPC_STRUCT_ACQUIRE_SUCCESS_MSK          0x80000000 // Is acquired
#define IPC_INTR_STRUCT_INTR_MASK_OFFSET 0x08 // Interrupt mask

#define SROMAPI_DATA_LOCATION_MSK 0x00000001 // [0]: 1 - arguments in IPC.DATA;
                                             //      0 - arguments in SRAM
#define SROMAPI_STATUS_MSK        0xF0000000
#define SROMAPI_STAT_SUCCESS      0xA0000000
#define SROMAPI_SILID_CODE        0x00000001 // SiliconId API code
#define SROMAPI_WRITEROW_CODE     0x05000100 // WriteRow API code
#define SROMAPI_PROGRAMROW_CODE   0x06000100 // ProgramRow API code
#define SROMAPI_ERASEALL_CODE     0x0A000001 // EraseAll API code
#define SROMAPI_ERASESECTOR_CODE  0x14000100 // EraseSector API code
#define SROMAPI_CHECKSUM_CODE     0x0B000001 // Checksum API code
#define SROMAPI_CHECKSUM_DATA_MSK 0x0FFFFFFF // Checksum mask
#define SROMAPI_BLOW_FUSE_CODE    0x01000001 // BlowFuse API code
#define SROMAPI_READ_FUSE_CODE    0x03000001 // ReadFuse API code
```

```
#define SROMAPI_GENERATE_HASH_CODE        0x1E000000 // GenerateHASH API code
#define SROMAPI_CHECK_FACTORY_HASH_CODE   0x27000001 // CheckFactoryHASH API code
#define SROMAPI_TRANSITION_TO_SECURE_CODE 0x2F000000 // TransitionToSecure API code

// 0x08003000: Address of SRAM where the API's parameters are stored by SW.
#define SRAM_SCRATCH_ADDR         MEM_BASE_SRAM + 0x00003000
#define ROW_SIZE                  512 // Flash Row Size
```

5.2.2 Subroutines

The programming flow includes some operations that are used in all steps. These are implemented as subroutines in the pseudocode.

Table 5 Subroutines used in programming flow

Subroutine	Description
bool WriteIO(addr32, data32)	Writes 32-bit data into the specified address of the CPU address space. Returns "true" if all SWJ transactions succeeded (ACKed).
bool ReadIO(addr32, OUT data 32)	Reads 32-bit data from the specified address of the CPU address space. Note that the actual size of the read data (8, 16, or 32 bits) depends on the setting in the CSW register of DAP (see Table 3). By default, all accesses are 32 bits long. Returns "true" if all SWDJ transactions succeeded (ACKed).
bool DAP_Handshake()	Performs device handshake after the reset. Uses DAP_JTAGtoSWD or DAP_SWDtoJTAG hardware access command to set the SWJ-DP in known state and reads the DP.IDCODE register. This sequence is repeated until read IDCODE register request is acknowledged or until timeout.
bool DAP_Init (apNum)	Initialize the Debug Port for programming operations. Accepts Access Port number as input parameter: 0 – System AP; 1 – CM0+ AP; 2 – CM4 AP.
bool DAP_ScanAP (OUT apNum)	Scans the Access Ports for the first available with CPU register access
bool Ipc_PollLockStatus (ipcId, isLockExpected)	Depending on isLockExpected parameter, waits until LOCK status bit of the IPC structure is released or acquired. ipcId input parameter determines the number of IPC structure (ipcId = 0 : CM0+ IPC_STRUCT; ipcId = 1 : CM4 IPC_STRUCT; ipcId = 2 : DAP IPC_STRUCT). Timeout is 1s. Returns "true" (success) if LOCK status bit corresponds to desired status; otherwise, returns "false".
bool Ipc_Acquire (ipcId)	Acquires IPC structure. The timeout is 1s. Returns "true" (success) if IPC structure is acquired; otherwise, returns "false".
bool PollSromApiStatus (addr32, OUT data32)	Waits until the SROM function is completed and then checks its status. addr32 is the address, where the SROM function status word is expected (IPC_STRUCT.DATA field or address in RAM if parameters for SROM function are passed in RAM). Timeout is 1s. Output parameter data32 is the status/result word, provided by the SROM function. Returns "true" (success) if the command is completed and its status is successful; otherwise, returns "false".
bool CallSromApi (callIdAndParams, OUT data32)	Executes SROM function. The input parameter is the API OpCode and parameters word. Output parameter data32 is the status/result word, provided by the SROM function. Returns "true" (success) if SROM function executed and returned success status; otherwise, returns "false".

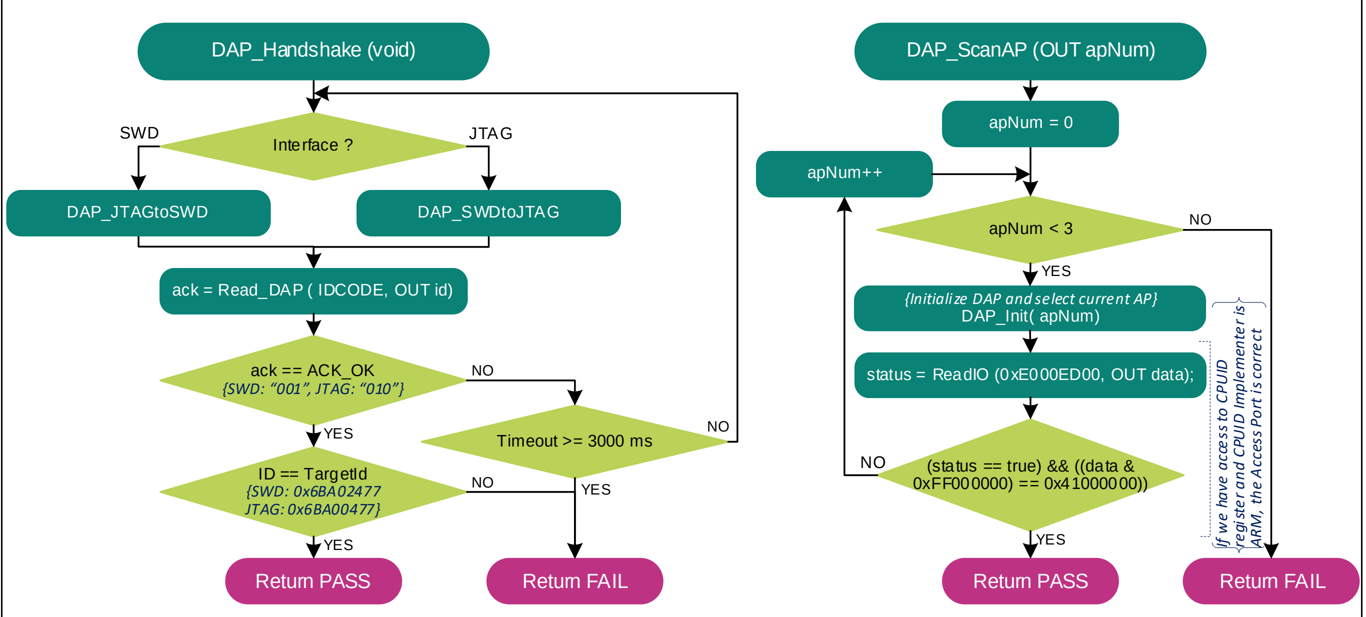
The implementation of these subroutines follows. It is based on the pseudocode and registers defined in the sections: Hardware access commands and Pseudocode. It uses the constants defined in this chapter.

```
//--- CPU/MMIO registers Read/Write subroutines
bool WriteIO (addr32, data32) {
  ackOK = (Interface == SWD)? 3b'001 /*SWD*/: 3b'010 /*JTAG*/;
  ack1 = Write_DAP (TAR, addr32);
  ack2 = Write_DAP (DRW, data32);
  return (ack1 == ackOK) && (ack2 == ackOK);
}
bool ReadIO (addr32, OUT data32) {
  ackOK = (Interface == SWD)? 3b'001 /*SWD*/: 3b'010 /*JTAG*/;
  ack1 = Write_DAP (TAR, addr32);
  ack2 = Read_DAP (DRW, OUT data32);
  ack3 = Read_DAP (RDBUFF, OUT data32);
  return (ack1 == ackOK) && (ack2 == ackOK) && (ack3 == ackOK);
}

//--- DAP initialization subroutines
bool DAP_Handshake () {
  // Timeout waiting for debug interface becomes enabled after device reset (tboot).
  // In worst case, when the boot code performs application HASH verification,
  // tboot is around 600ms and depends on CPU clock used by boot code.
  // For PowerCycle, timeout depends on the design schematic and must be longer.
  timeout  = 3000 ms;
  ackOK    = (Interface == SWD)? 3b'001      /*SWD*/: 3b'010      /*JTAG*/;
  targetID = (Interface == SWD)? 0x6BA02477 /*SWD*/: 0x6BA00477 /*JTAG*/;
  // Execute connection sequence - acquire port.
  // This is used as handshake between the debugger and target device.
  // Once the target device replied on request to read the IDCODE,
  // it means that the device is already booted after reset and ready to
communicate.
  do {
    if (Interface == SWD) DAP_JTAGtoSWD(); // SWD
    else DAP_SWDtoJTAG(); // JTAG
    ack = Read_DAP (IDCODE, OUT id);
  } while ((ack != ackOK) && (TimeElapsed < timeout));
  return (TimeElapsed <= timeout) && (id == targetID);
}

bool DAP_Init (apNum) {
  if (DAP_Handshake() == false) return false; // Handshake (e.g. after reset)
  if (Interface == JTAG) {
    // Power up debug port using the next bits in CTRL/STAT register:
    // [30]:CSYSPWRUPREQ and [28]:CDBGPWRUPREQ - power-up requests.
    // [5]:STICKYERR, [4]:STICKYCMP and [1]:STICKYORUN - sticky errors bits
    // Note: for JTAG, sticky error bits are read-write enabled and writing '1'
    // to these bits clears associated sticky errors.
    // For SWD, these bits are read-only and to clean the sticky errors,
    // you should write to appropriate bits of DP.ABORT register
    Write_DAP (CTRL/STAT, 0x50000032);
  }
  else { // SWD
    // Clear any Sticky Errors which could be left from previous sessions
    // Otherwise only power-down-up cycle helps to restore DAP.
    Write_DAP (ABORT, 0x0000001E);
    // Power up DAP
    Write_DAP (CTRL/STAT, 0x50000000);
  }
  // Select desired Access Port and set bank 0 in APACC space
  apSelect = apNum << 24;
  Write_DAP (SELECT, apSelect);
  // Set CSW (DbgSwEnable=0, Prot=0x23, SPIDEN=0, Mode=0x0, TrInProg=0,
  // DeviceEn=0, AddrInc=Auto-increment off, Size=Word (32 bits))
  // Note: Set Prot bits in DAP CSW register, because of no access to CPU
  // registers via M4 AP without these bits
  Write_DAP (CSW, 0x23000002);
  return true;
}

bool DAP_ScanAP (OUT apNum) {
  // Try all possible Access Ports
  // Scan only three APs [0]-[3], what is sufficient for MXS40 architecture.
  for (apNum = 0; apNum < 3; apNum++) {
    // Initializes DAP and selects Access Port with provided number
    if (DAP_Init (apNum) == false) continue;
    // Try to read the CPUID register. If the Implementer is ARM, the Access Port
    // is correct (we have access to the ARM registers)
    status = ReadIO (0xE000ED00, OUT data);
    if ((status == true) && ((data & 0xFF000000) == 0x41000000))
       return true;
  }
  return false;
}
```


DAP_Handshake (void)
SWD
Interface ?
JTAG
DAP_JTAGtoSWD
DAP_SWDtoJTAG
ack = Read_DAP (IDCODE, OUT id)
ack == ACK_OK
{SWD: "001", JTAG: "010"}
NO
YES
Timeout >= 3000 ms
NO
ID == TargetId
{SWD: 0x6BA02477
JTAG: 0x6BA00477}
NO
YES
YES
Return PASS
Return FAIL
DAP_ScanAP (OUT apNum)
apNum = 0
apNum++
apNum < 3
NO
YES
{Initialize DAP and select current AP}
DAP_Init(apNum)
status = ReadIO (0xE000ED00, OUT data);
NO
(status == true) && ((data & 0xFF000000) == 0x41000000))
YES
If we have access to CPUID register and CPUID Implementer is ARM, the Access Port is correct
Return PASS
Return FAIL

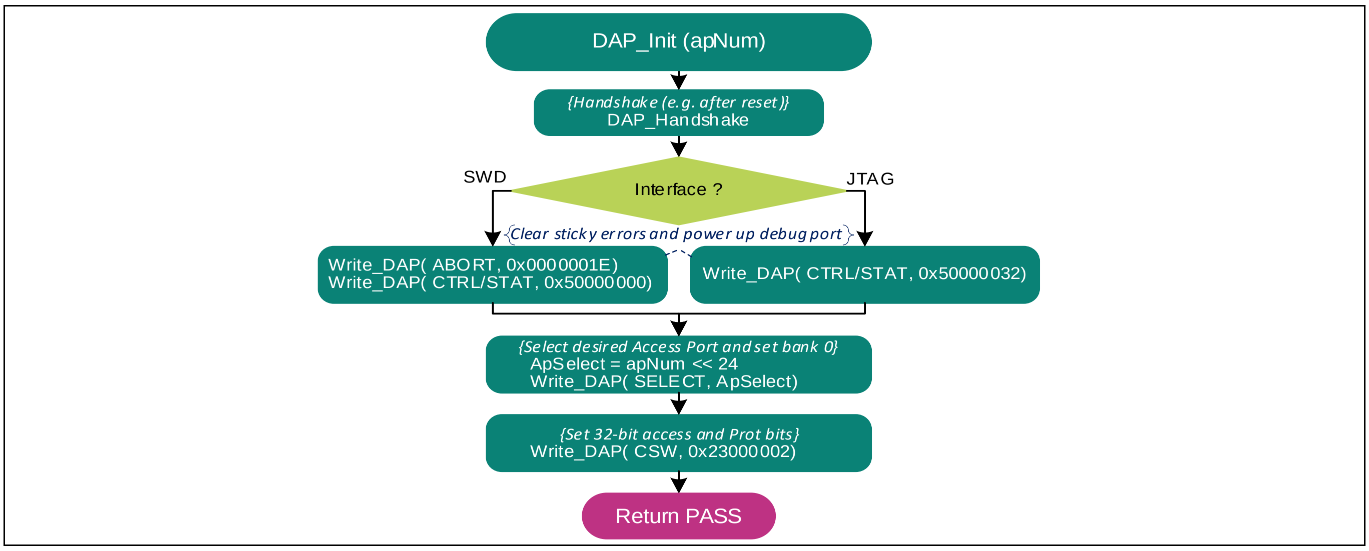

DAP_Init (apNum)
{Handshake (e.g. after reset)}
DAP_Handshake
SWD
Interface ?
JTAG
{Clear sticky errors and power up debug port}
Write_DAP(ABORT, 0x0000001E)
Write_DAP(CTRL/STAT, 0x50000000)
Write_DAP(CTRL/STAT, 0x50000032)
{Select desired Access Port and set bank 0}
ApSelect = apNum << 24
Write_DAP(SELECT, ApSelect)
{Set 32-bit access and Prot bits}
Write_DAP(CSW, 0x23000002)
Return PASS

Figure 10 **DAP initialization subroutines**

```
//--- SROM API usage subroutines

bool Ipc_PollLockStatus (ipcId, isLockExpected) {
  ipcAddr = IPC_STRUCT0 + IPC_STRUCT_SIZE * ipcId;
  do{
    ReadIO (ipcAddr + IPC_STRUCT_LOCK_STATUS_OFFSET, OUT status);
    isLocked = (status & IPC_STRUCT_LOCK_STATUS_ACQUIRED_MSK) != 0;
    isExpectedStatus = (isLockExpected && isLocked) || (!isLockExpected &&
!isLocked)
  }
  while ((!isExpectedStatus) && (TimeElapsed < 1 sec))
  return (TimeElapsed <= 1 sec)? true /* OK */: false /* timeout */;
}

bool Ipc_Acquire (ipcId) {
  ipcAddr = IPC_STRUCT0 + IPC_STRUCT_SIZE * ipcId;
  do {
    // Workaround for the SROM code issue. Write to IPC_ACQUIRE register may
    // fail because IPC structure is left protected by the PPU after previous
    // SROM API call.
    status = ReadIO (MEM_BASE_PPU4, OUT NULL); // Ignore output value
    if(status == false) DAP_Init(apNum);

    // To acquire the IPC[2] (IPC structure for DAP),
    // debugger must write any value to IPC[2].ACQUIRE register.
    // The write operation acquires the lock. The write value is irrelevant.
    // Note: This write is NOT required for flash loaders (running on CM0+ or CM4)
    status = WriteIO (ipcAddr + IPC_STRUCT_ACQUIRE_OFFSET,
                      IPC_STRUCT_ACQUIRE_SUCCESS_MSK);
    // To acquire the IPC[0] (CM0) or IPC[1] (CM4) (e.g. in flash loaders)
    // Master must read IPC[x].ACQUIRE register
    // If the SUCCESS field returns a '1', the read acquired the lock.
    // If the SUCCESS field returns a '0', the read did not acquire the lock.
    // Note that a single read access performs two functions:
    // - The attempt to acquire a lock.
    // - Return the result of the acquisition attempt (SUCCESS field).
    ReadIO (ipcAddr + IPC_STRUCT_ACQUIRE_OFFSET, OUT status);
    status &= IPC_STRUCT_ACQUIRE_SUCCESS_MSK;
  } while ((status == 0) && (TimeElapsed < 1 sec))
  return (TimeElapsed <= 1 sec)? true /* OK */: false /* timeout */;
}

bool PollSromApiStatus (addr32, OUT data32) {
  do {
    ReadIO (addr32, OUT data32);
    status = data32 & SROMAPI_STATUS_MSK;
  } while (status != SROMAPI_STAT_SUCCESS) && (TimeElapsed < 1 sec))
  return (TimeElapsed <= 1 sec)? true /* OK */: false /* timeout */;
}

bool CallSromApi (callIdAndParams, OUT data32) {
  // Use IPC for CM0+ (IpcId = 0) if using flash loader running on CM0+ core
  // Use IPC for CM4  (IpcId = 1) if using flash loader running on CM4 core
  // Use IPC for DAP  (IpcId = 2) if using external debugger
  ipcId = 2;
  ipcAddr = IPC_STRUCT0 + IPC_STRUCT_SIZE * ipcId;
  // Check where the arguments for SROM function are located
  // [0]: 1 - arguments are passed in IPC.DATA. 0 - arguments are passed in SRAM
  isDataInRam = (callIdAndParams & SROMAPI_DATA_LOCATION_MSK) == 0;

  // Acquire IPC_STRUCT[ipcId]
  if(Ipc_Acquire (ipcId) == false) return false;
  // Write one of these to IPC_STRUCT[ipcId].DATA:
  // a) SROM OpCode with Parameters (if all API parameters fit in one word)
  // b) Address in SRAM, where they are located
  if (isDataInRam) WriteIO (ipcAddr + IPC_STRUCT_DATA_OFFSET, SRAM_SCRATCH_ADDR);
  else             WriteIO (ipcAddr + IPC_STRUCT_DATA_OFFSET, callIdAndParams);

  // Set IPC_INTR_STRUCT[0(CM0+)].INTR_MASK to enable notification interrupt for
  // IPC_STRUCT[ipcId]. Read initial value first to restore other bits of
  // INTR_MASK field after system call
  intrMskDapEnabled = 1 << (16 + ipcId);
  ReadIO (IPC_INTR_STRUCT + IPC_INTR_STRUCT_INTR_MASK_OFFSET , &intrMskInitial);
  doWriteRestoreIntrMsk = intrMskInitial != intrMskDapEnabled;
  // Set just DAP enabled bit - do not OR it with initial mask, because of other
  // enabled interrupts notifications may hurt programming if the user application
  // running by 2nd core invokes system calls.
  if (doWriteRestoreIntrMsk) WriteIO (IPC_INTR_STRUCT +
IPC_INTR_STRUCT_INTR_MASK_OFFSET, intrMskDapEnabled);

  // Notify to IPC_INTR_STRUCT[0]. IPC_STRUCT[IpcId].MASK <- Notify
  // This starts SROM function execution
  WriteIO (ipcAddr + IPC_STRUCT_NOTIFY_OFFSET, 1 << 0 /*IPC_INTR_STRUCT0*/);
  // Poll lock status for released state and poll Data word
  if (Ipc_PollLockStatus(ipcId, false) == false) return false;
  if (isDataInRam) status = PollSromApiStatus (SRAM_SCRATCH_ADDR, data32);
  else status = PollSromApiStatus (ipcAddr + IPC_STRUCT_DATA_OFFSET, data32);
  // Restore IPC_INTR_STRUCT[0(CM0+)].INTR_MASK if was modified
  if (doWriteRestoreIntrMsk)
    WriteIO (IPC_INTR_STRUCT + IPC_INTR_STRUCT_INTR_MASK_OFFSET, intrMskInitial);
  return status;
}
```

5.3 Step 1.A – Acquire PSOC™ 6 MCU

The first step in programming the PSOC™ 6 MCU is to put it into Test mode (or Programming mode). This is a special mode in which the CPU is controlled by the external programmer, which can also access other system resources such as SRAM and registers. The main purpose of this step is to prevent the execution of the user code from the application flash region. After the user's code starts, it can repurpose the SWJ pins [3] (use them as GPIO), so the external debugger will not be able to communicate with the device. Additionally, if there is corrupted user code in the application flash region, the Cortex®-M0+ core may enter a lockup state. This step has strict timing requirements that the host must meet to enter Test mode successfully. Figure 11 shows the timing diagram for entering Test mode.

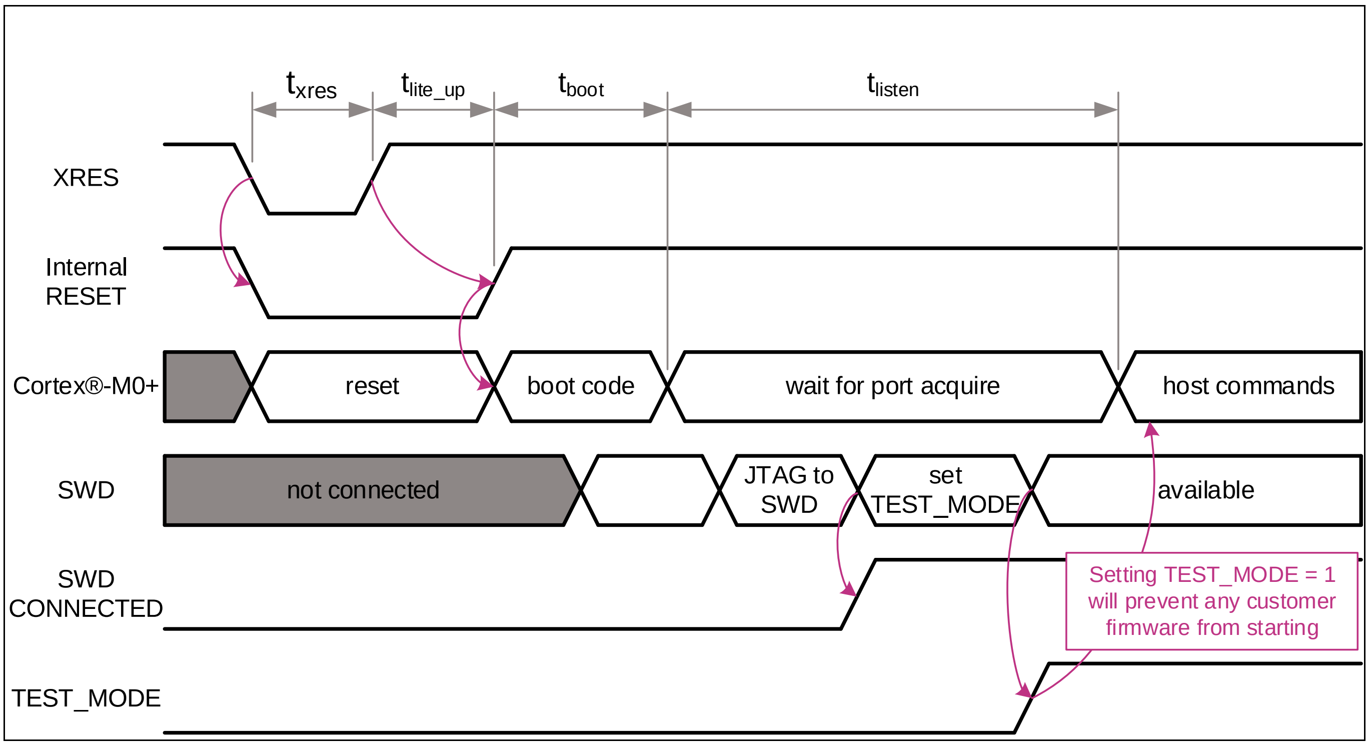


Figure 11 Timing diagram for entering test mode

[3] Application firmware is expected to follow this procedure for SWJ pin configuration:

a) Do not touch the configuration of the SWJ pins for parts that have a permanent SWD interface. They will be properly configured and may have already connected to the SWD probe when the firmware starts.

b) For parts that repurpose their SWD pins:

- If the SWD interface is presently active (CPUSS_DP_STATUS.SWJ_CONNECTED bit is '1'), leave the pins in their current state; a probe has connected during the acquire window and the pins should not be repurposed.
- If the SWD interface is not active, you may configure the pins and enable the alternate purpose.

This diagram details the chip's internal signals while entering Test mode. Everything starts from toggling the XRES line (or applying power), so the chip enters Internal Reset mode for t_{lite_up}period. After that, the system boot code starts execution. When completed, the CPU waits during a t_{listen} period for a special connection sequence on the SWJ port. If, during this time, the host sends the correct sequence of SWJ commands, the CPU enters Test mode. Otherwise, it starts the execution of the user's code from the application flash region. Timing parameters may vary depending on the boot code execution flow (see Table 6). Therefore, the best way to enter Test mode is to start sending an acquire sequence immediately after XRES is toggled (or power is supplied in Power Cycle mode). This sequence is sent iteratively until it succeeds (all SWJ transactions are ACKed and all conditions are met).

Table 6 Boot timing parameters

Parameter	Description	Min	Max	Unit
t_{lite_up}	Time from Reset release until CPU begins executing boot code	–	250	µs
t_{boot}	Time from when boot code started execution until it opens SWJ lines and starts waiting for TEST_MODE sequence. This time varies depending on CPU clock, device lifecycle stage, and amount of data for HASH verification by boot code for SECURE application.	0.7	600	ms
t_{listen}	Amount of time boot code waits and listens for the SWJ port initialization sequence before starting the application firmware execution. Note that the default duration of listen window (t_{listen}) is 20 ms, but it may vary from 0 ms to 100 ms. This can be configured by specific bits in the table of contents part 2 (TOC2).	0	100	ms

Figure 12 shows the Acquire Chip procedure. It is detailed in terms of the SWD transaction. Note that the recommended minimum frequency of the programmer is 1.5 MHz, which meets the timing requirement of this step.

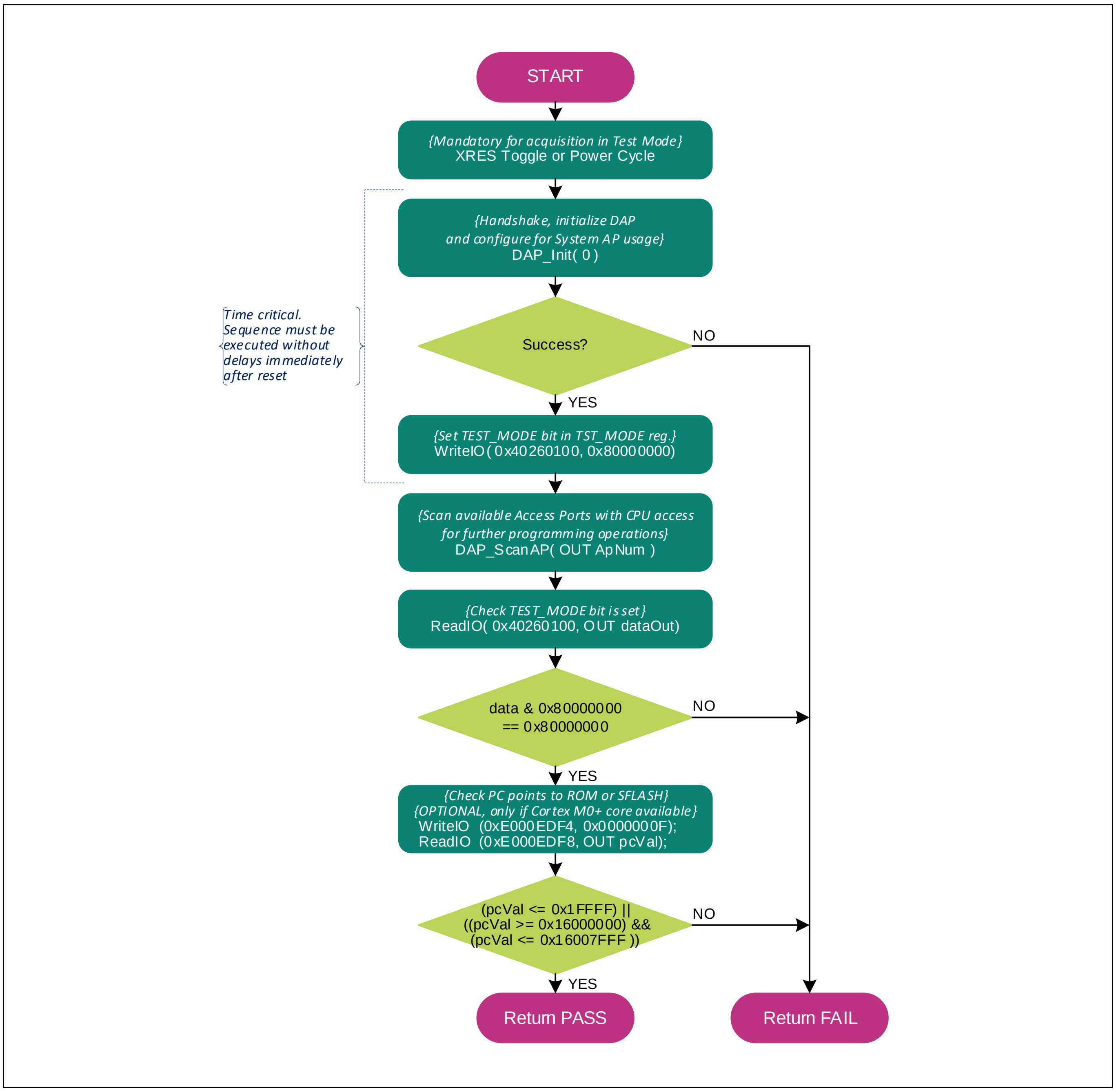

START
{Mandatory for acquisition in Test Mode}
XRES Toggle or Power Cycle
{Handshake, initialize DAP and configure for System AP usage}
DAP_Init(0)
Time critical. Sequence must be executed without delays immediately after reset
Success?
NO
YES
{Set TEST_MODE bit in TST_MODE reg.}
WriteIO(0x40260100, 0x80000000)
{Scan available Access Ports with CPU access for further programming operations}
DAP_ScanAP(OUT ApNum)
{Check TEST_MODE bit is set}
ReadIO(0x40260100, OUT dataOut)
data & 0x80000000 == 0x80000000
NO
YES
{Check PC points to ROM or SFLASH}
{OPTIONAL, only if Cortex M0+ core available}
WriteIO (0xE000EDF4, 0x0000000F);
ReadIO (0xE000EDF8, OUT pcVal);
(pcVal <= 0x1FFFF) || ((pcVal >= 0x16000000) && (pcVal <= 0x16007FFF))
NO
YES
Return PASS
Return FAIL

Figure 12 **Flowchart of the acquire chip step**

Code Listing 1 **Pseudocode – Step 1.A. Acquire chip**

```
// Reset Target depending on acquire mode - Reset or Power Cycle
if (AcquireMode == RESET) ToggleXRES(); // Toggle XRES pin, target must be powered
else if (AcquireMode == POWER_CYCLE) PowerOn();// Supply power to target

// Do handshake between the debugger and target device after reset,
// Initialize the Debug Port and select System Access Port (AP[0])
DAP_Init (0);

// Enter CPU into Test Mode
// Set TEST_MODE bit in TST_MODE SRSS register
WriteIO (SRSS_TST_MODE, SRSS_TST_MODE_TEST_MODE_MSK);
```

```
// The steps above are time critical and must be executed without delays after
// reset. No hurry for further steps - target already acquired in Test Mode

// Scan for and initialize Access Port with CPU access for further operations
DAP_ScanAP (OUT ApNum);

// Check TEST_MODE bit is set
ReadIO (SRSS_TST_MODE, OUT dataOut);
if ((dataOut & SRSS_TST_MODE_TEST_MODE_MSK) == 0) return FAIL;

// OPTIONAL - ONLY for devices with Cortex M0+ core available
// Check PC - in Test Mode, it should point to address in ROM or in Sflash
WriteIO (0xE000EDF4, 0x0000000F);
ReadIO  (0xE000EDF8, OUT pcVal);
if (((pcVal >= MEM_BASE_ROM) && (pcVal < (MEM_BASE_ROM + MEM_SIZE_ROM))
 || ((pcVal >= MEM_BASE_SFLASH) && (pcVal < (MEM_BASE_SFLASH + MEM_SIZE_SFLASH))))
 return PASS; else return FAIL;

return PASS
```

5.4 Step 1.B – Acquire PSOC™ 6 MCU (alternate method)

The "Acquire Chip" sequence in the previous section is based on entering the PSOC™ 6 MCU test mode by triggering a hard-reset condition and then sending the acquire sequence within the specified time window. The hard-reset condition is generated by toggling either the XRES pin or the power supply to the device. Programming by entering test mode using XRES or power cycling is the recommended method for third-party production programmers or any other general-purpose programmer.

There might be cases where the host programmer hardware or software constraints might prevent the programming of the device in test mode. These constraints can include:

- Host programmer hardware might be IO pin constrained and cannot spare an extra IO for toggling the XRES pin or the power supply to the PSOC™ 6 MCU. Only the SWJ protocol pins are available for programming.
- The host programmer software application is unable to meet the timing requirements to enter PSOC™ 6 MCU test mode after triggering a hard-reset condition. In such a scenario, the MCU enters the user code execution mode after the test mode timing window elapses.

For a host programmer with any of the above constraints, the modified acquire-chip sequence provided in this section does not require XRES/power supply toggling, and it does not have the test-mode timing requirements. Only the SWJ protocol pins are used for programming. This modified sequence works only under the following conditions:

- The SWJ pins on the PSOC™ 6 MCU have not been repurposed for any other application-firmware-specific use. If the SWJ pins are repurposed as part of the existing firmware image in flash memory, the SWJ pins are not available for communication with the host SWJ interface to update the existing firmware image.
- The Access Restriction Properties allow SWJ access to the Access Debug Ports (Normal Access Restriction properties are applicable if the device is in the Normal Protection state, Secure and Dead Access Restriction properties are applicable if the device is in the Secure and Dead Protection state respectively).

Developers wanting to program devices using the modified sequence should be aware of these limitations. Devices coming from the factory satisfy both the above-listed conditions and hence can be programmed using the modified acquire sequence. But if firmware not meeting any of the above conditions is programmed to the PSOC™ 6 MCU, then subsequent re-programming of the device is not possible using the modified acquire sequence. Due to this limitation, this method is not recommended for third-party programmers or general-purpose programmers because they would generally be required to support programming under all possible operating conditions.

Figure 13 shows the Acquire chip (alternate method) procedure.

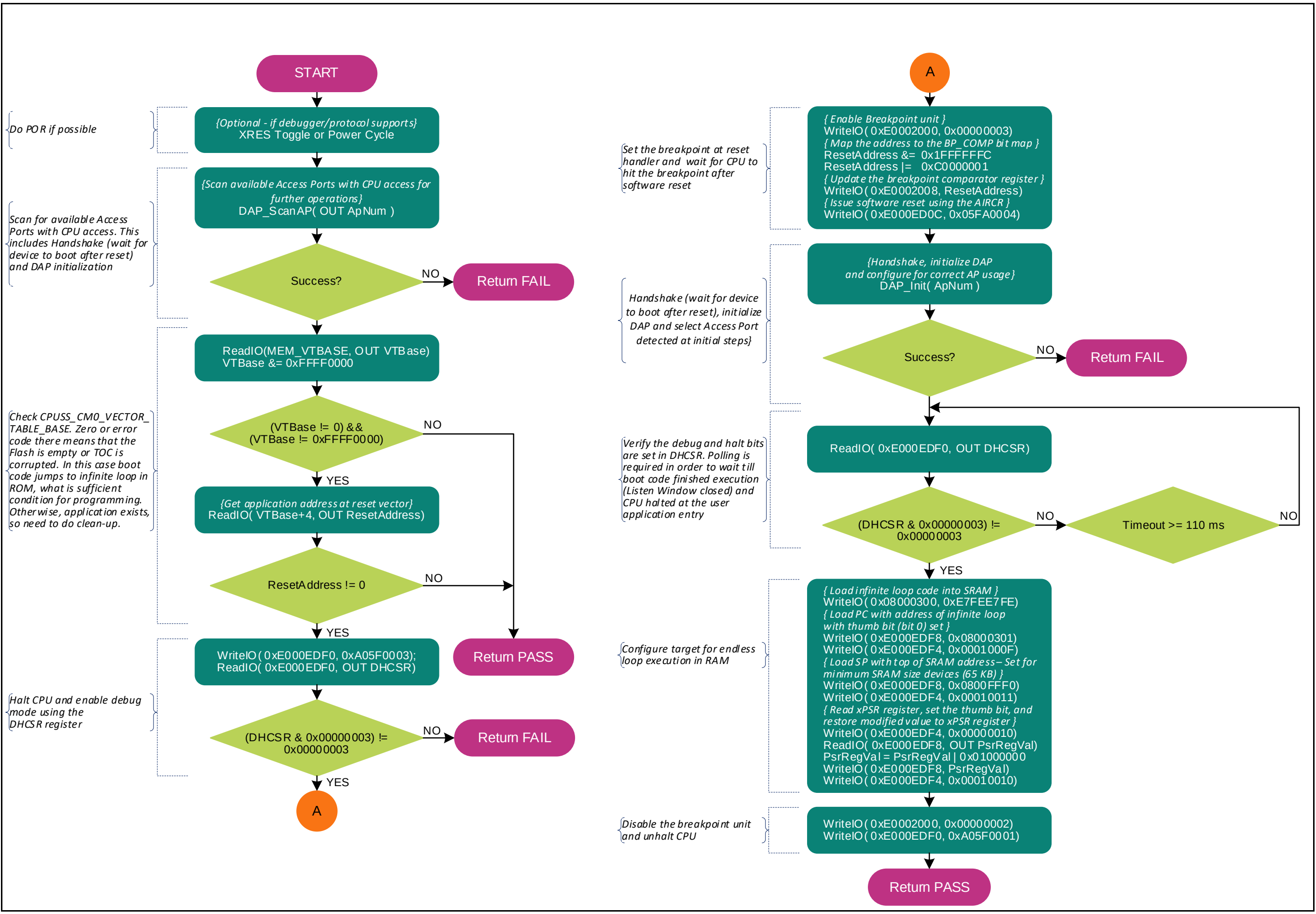

START
Do POR if possible
{Optional - if debugger/protocol supports}
XRES Toggle or Power Cycle
Scan for available Access Ports with CPU access. This includes Handshake (wait for device to boot after reset) and DAP initialization
{Scan available Access Ports with CPU access for further operations}
DAP_ScanAP(OUT ApNum)
Success?
NO
Return FAIL
Check CPUSS_CM0_VECTOR_TABLE_BASE. Zero or error code there means that the Flash is empty or TOC is corrupted. In this case boot code jumps to infinite loop in ROM, what is sufficient condition for programming. Otherwise, application exists, so need to do clean-up.
ReadIO(MEM_VTBASE, OUT VTBase)
VTBase &= 0xFFFF0000
(VTBase != 0) &&
(VTBase != 0xFFFF0000)
NO
YES
{Get application address at reset vector}
ReadIO(VTBase+4, OUT ResetAddress)
ResetAddress != 0
NO
YES
Return PASS
Halt CPU and enable debug mode using the DHCSR register
WriteIO(0xE000EDF0, 0xA05F0003);
ReadIO(0xE000EDF0, OUT DHCSR)
(DHCSR & 0x00000003) != 0x00000003
NO
Return FAIL
YES
A
A
Set the breakpoint at reset handler and wait for CPU to hit the breakpoint after software reset
{ Enable Breakpoint unit }
WriteIO(0xE0002000, 0x00000003)
{ Map the address to the BP_COMP bit map }
ResetAddress &= 0x1FFFFFFC
ResetAddress |= 0xC0000001
{ Update the breakpoint comparator register }
WriteIO(0xE0002008, ResetAddress)
{ Issue software reset using the AIRCR }
WriteIO(0xE000ED0C, 0x05FA0004)
Handshake (wait for device to boot after reset), initialize DAP and select Access Port detected at initial steps}
{Handshake, initialize DAP and configure for correct AP usage}
DAP_Init(ApNum)
Success?
NO
Return FAIL
Verify the debug and halt bits are set in DHCSR. Polling is required in order to wait till boot code finished execution (Listen Window closed) and CPU halted at the user application entry
ReadIO(0xE000EDF0, OUT DHCSR)
(DHCSR & 0x00000003) != 0x00000003
NO
Timeout >= 110 ms
NO
YES
Configure target for endless loop execution in RAM
{ Load infinite loop code into SRAM }
WriteIO(0x08000300, 0xE7FEE7FE)
{ Load PC with address of infinite loop with thumb bit (bit 0) set }
WriteIO(0xE000EDF8, 0x08000301)
WriteIO(0xE000EDF4, 0x0001000F)
{ Load SP with top of SRAM address – Set for minimum SRAM size devices (65 KB) }
WriteIO(0xE000EDF8, 0x0800FFF0)
WriteIO(0xE000EDF4, 0x00010011)
{ Read xPSR register, set the thumb bit, and restore modified value to xPSR register }
WriteIO(0xE000EDF4, 0x00000010)
ReadIO(0xE000EDF8, OUT PsrRegVal)
PsrRegVal = PsrRegVal | 0x01000000
WriteIO(0xE000EDF8, PsrRegVal)
WriteIO(0xE000EDF4, 0x00010010)
Disable the breakpoint unit and unhalt CPU
WriteIO(0xE0002000, 0x00000002)
WriteIO(0xE000EDF0, 0xA05F0001)
Return PASS

Figure 13 Flowchart of the acquire chip step (alternate method)

Code Listing 2 Pseudocode – Step 1.B Acquire chip (alternate method)

```
// Reset Target depending on acquire mode - Reset or Power Cycle
// This step is optional - just to ensure silicon did not stuck in some error state
if (AcquireMode == RESET) ToggleXRES(); // Toggle XRES pin, target must be powered
else if (AcquireMode == POWER_CYCLE) PowerOn();// Supply power to target

// Scan for available Access Ports with CPU access. This includes Handshake
// (wait for device to boot after reset and DAP initialization
success = DAP_ScanAp (OUT ApNum );
if (!success) return FAIL;

// Check CPUSS_CM0_VECTOR_TABLE_BASE. Zero or error code means that the Flash is
// empty or TOC is corrupted. In this case boot code jumps to infinite loop in ROM
// or executes ‘dummy’ application, what is sufficient condition for programming.
// Otherwise, user application exists, so need to do clean-up.
ReadIO (MEM_VTBASE, OUT vtBase);
vtBase &= 0xFFFF0000;
if ((vtBase == 0) || (vtBase == 0xFFFF0000)) return PASS;
// Get application address at reset vector
ReadIO (vtBase + 4, OUT resetAddress);
if (resetAddress == 0) return PASS;

// Enable/verify debug, and halt the CPU using the DHCSR register
WriteIO (0xE000EDF0, 0xA05F0003);
ReadIO (0xE000EDF0, OUT DHCSR);
if ((DHCSR & 0x00000003) != 0x00000003) return FAIL;

// Enable Breakpoint unit using the BP_CTRL (Breakpoint Control Register)
WriteIO (0xE0002000, 0x00000003); // Set bits [0]: ENABLE =1, [1]: KEY=1
// Update the breakpoint compare register: map the address bits to the breakpoint
// compare register bit map, set the enable breakpoint bit, and the match bits
resetAddress = (resetAddress & 0x1FFFFFFC) | 0xC0000001;
WriteIO (0xE0002008, resetAddress);

// Issue software reset using the AIRCR (Application Interrupt and Reset
// Control Register). Note: do not check for ACK_OK of this operation
WriteIO (0xE000ED0C, 0x05FA0004);
// Handshake (wait for device to boot after reset), initialize DAP
// and select Access Port detected at initial steps}
success = DAP_Init (ApNum);

// Verify the debug and halt bits are set in DHCSR.
// Polling is required in order to wait till boot code finished execution
// (Listen Window closed) and CPU halted at the user application entry
do {
  ReadIO (0xE000EDF0, OUT DHCSR);
} while (((DHCSR & 0x00000003) != 0x00000003)) && (TimeElapsed < 110 ms));
if (TimeElapsed >= 110 ms) return FAIL;

// Load infinite loop code in SRAM address 0x08000300
// And set PC with this address + thumb bit (bit 0) set
WriteIO (0x08000300, 0xE7FEE7FE);
WriteIO (0xE000EDF8, 0x08000301);
WriteIO (0xE000EDF4, 0x0001000F);

// Load SP with top of SRAM address – Set for minimum SRAM size devices (65 KB)
WriteIO (0xE000EDF8, 0x0800FFF0);
WriteIO (0xE000EDF4, 0x00010011);

// Set the thumb bit in xPSR register
WriteIO (0xE000EDF4, 0x00000010);
ReadIO  (0xE000EDF8, OUT psrRegVal);
psrRegVal = psrRegVal | 0x01000000;
WriteIO (0xE000EDF8, psrRegVal);
WriteIO (0xE000EDF4, 0x00010010);

// Disable Breakpoint unit and unhalt CPU
WriteIO (0xE0002000, 0x00000002);
WriteIO (0xE000EDF0, 0xA05F0001);
return PASS;
```

5.5 Step 2 – Identify silicon

This step is required to identify the acquired PSOC™ 6 MCU and verify that it corresponds to the input data file. It reads the ID from the metadata section in the file and compares it with the ID obtained from the PSOC™ 6 MCU, using the Silicon ID SROM function.

Code Listing 3 Pseudocode – Step 2. Check silicon ID

```
// Read "Silicon ID" from the target using SROM request
// Type 0: Get Family ID & Revision ID
opCode = SROMAPI_SILID_CODE + (0x0000FF00 & (0 << 8));
status = CallSromApi (opCode, OUT dataOut0);
if(!status) return FAIL;

// Type 1: Get Silicon ID and protection state
opCode = SROMAPI_SILID_CODE + (0x0000FF00 & (1 << 8));
status = CallSromApi(opCode, OUT dataOut1);
if (!status) return FAIL;

FamilyIdHi     = (dataOut0 & 0x0000FF00) >>  8; // Family ID High
FamilyIdLo     = (dataOut0 & 0x000000FF) >>  0; // Family ID Low
RevisionId     = (dataOut0 & 0x00FF0000) >> 16; // Rev ID Major & Rev ID Minor
SiliconIdHi    = (dataOut1 & 0x0000FF00) >>  8; // Silicon ID High
SiliconIdLo    = (dataOut1 & 0x000000FF) >>  0; // Silicon ID Low
ProtectState   = (dataOut1 & 0x000F0000) >> 16; // Protection state:
                           // 0- UNKNOWN, 1- VIRGIN, 2- NORMAL, 3- SECURE, 4- DEAD
LifeCycleStage = (dataOut1 & 0x00F00000) >> 20; // Life cycle stage:
                           // 0 - VIRGIN, 1 - NORMAL, 2- SEC_W_DBG, 3 -SECURE
// Identify Device Family:
// Device Family                    | Family ID | Si ID Range
// PSOC6A-BLE2 (CY8C6xx6, CY8C6xx7) | 0x100     | E200-E2FF
// PSOC6A-2M   (CY8C6xx8, CY8C6xxA) | 0x102     | E400-E4FF
// PSOC6A-512K (CY8C6xx5)           | 0x105     | E700-E7FF
// PSOC6A-256K (CY8C6xx4)           | 0x10E     | EAC0-EAFF
DeviceFamily = UNKNOWN;
if (FamilyIdHi == 0x01) {
 if ((FamilyIdLo == 0x00) && (SiliconIdHi == 0xE2))      DeviceFamily = PSOC6ABLE2;
 else if ((FamilyIdLo == 0x02) && (SiliconIdHi == 0xE4)) DeviceFamily = PSOC6A2M;
 else if ((FamilyIdLo == 0x05) && (SiliconIdHi == 0xE7)) DeviceFamily = PSOC6A512K;
 else if ((FamilyIdLo == 0x0E) && (SiliconIdHi == 0xEA)) DeviceFamily = PSOC6A256K;
}
if (DeviceFamily == UNKNOWN) return FAIL;

// Read Silicon ID from binary file, 4 bytes from address 0x9050 0002 (big endian):
// fileID[0] - Silicon ID Hi ; fileID[1] - Silicon ID Lo
// fileID[2] - Revision ID   ; fileID[3] - Family ID
// Compare the Family ID and Silicon ID in device vs. data file
fileID = FILE_ReadSiliconID(); // API must be implemented.
if ((FamilyIdHi  != fileID.FamilyIdHi)  || (FamilyIdLo  != fileID.FamilyIdLo) ||
    (SiliconIdHi != fileID.SiliconIdHi) || (SiliconIdLo != fileID.SiliconIdLo)
    return FAIL;

return PASS;
```

5.6 Step 3 – Erase application flash

The flash must be erased before programming. This step erases all rows in application flash calling the EraseAll or EraseSector SROM functions.

Code Listing 4 Pseudocode – Step 3. Erase application flash

```
// Read protection state from target device
// (0: UNKNOWN, 1: VIRGIN,  2: NORMAL, 3: SECURE, 4: DEAD)
```

```
opCode = SROMAPI_SILID_CODE + (0x0000FF00 & (1 << 8));
CallSromApi (opCode, OUT dataOut);
protectState = (dataOut & 0x000F0000) >> 16;
// If silicon protection state is VIRGIN or NORMAL, use 'EraseAll' SROM function
if ((chipProt == 1) || (chipProt == 2)) {
  status = CallSromApi (SROMAPI_ERASEALL_CODE, OUT dataOut);
}
else {
  // Otherwise (SECURE, DEAD...) use 'EraseSector' SROM function to erase
  // entire application flash by:
  // - 256 KB sectors (0x10000000, 0x10040000, 0x10080000 and 0x100C0000 for
  //   target devices with 1MB of application flash memory
  // - 128 KB sectors (0x10000000 and 0x10020000 for target devices with 256KB of
  //   application flash memory
  totalSectors = 4; // Use 2 for devices with 256/512 KB of application flash
  sectorSize = 0x40000; // Use 0x20000 for devices with 256 KB of application flash
  for (long i = 0; i < totalSectors; i++) {
    // SRAM_SCRATCH: EraseSector SROM function OpCode
    WriteIO (SRAM_SCRATCH_ADDR, SROMAPI_ERASESECTOR_CODE);
    // SRAM_SCRATCH + 0x04: Flash address to be erased
    sectAddr = MEM_BASE_FLASH + i * sectorSize;
    WriteIO (SRAM_SCRATCH_ADDR + 0x04, sectAddr);
    status = CallSromApi (SROMAPI_ERASESECTOR_CODE, OUT dataOut);
    if (!status) break;
  }
}
return status;
```

5.7 Step 4 – Verify blank checksum (optional)

This step uses the Checksum SROM function to verify that the checksum of the erased application flash is 0x00. This step is considered as optional because the EraseAll and EraseSector SROM functions (used in the Erase Application Flash step) always perform the checksum after the erase and return error status if a non-zero checksum is encountered.

5.7.1 Pseudocode – Step 4. Verify checksum

```
// Bytes 1 and 2 of the parameters select whether the checksum
// is performed on the whole flash, or a row of flash.
// The row of flash is determined by the Row Id Lo and Row Id Hi parameters.
opCode = SROMAPI_CHECKSUM_CODE +
   (0 << 22) +          // Flash region: 0 – application, 1 - AUXflash, 2 - Sflash
   (1 << 21) +          // Whole flash: 0 – page, 1 – whole flash
   ((0 & 0x1FFF) << 8); // Row id Bits[20:8]
CallSromApi(opCode, OUT dataOut);
// Get checksum bits (skip SROM function status)
// After the user’s flash is erased, its checksum must be 0x00
checksum = dataOut & SROMAPI_CHECKSUM_DATA_MSK;
return (checksum == 0)? PASS : FAIL;
```

5.8 Step 5 – program application flash

Flash memory is programmed in rows. Each row is 512 bytes long. The programmer must serially program each row individually. The source data is extracted from the binary file starting from address 0x1000 0000 (see Figure 3). The flash size and the row size are input parameters of this step. Note that the flash size of the acquired silicon must be equal to the size of the user's code in the binary file, as verified in Step 2 by comparing the silicon IDs of the file and the PSOC 6 MCU. This step uses the ProgramRow SROM function.

Figure 14 illustrates this programming algorithm.

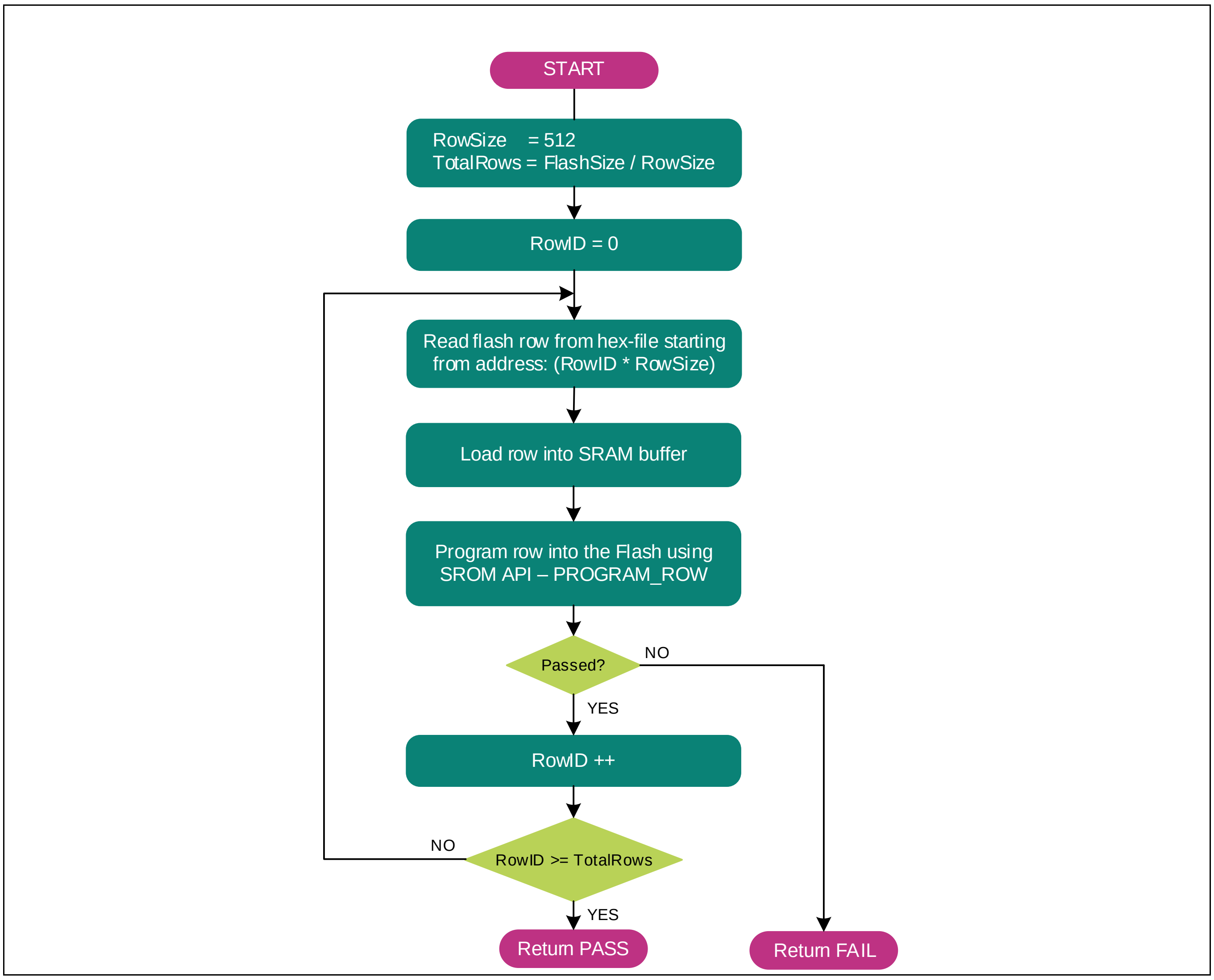

START
RowSize = 512
TotalRows = FlashSize / RowSize
RowID = 0
Read flash row from hex-file starting from address: (RowID * RowSize)
Load row into SRAM buffer
Program row into the Flash using SROM API – PROGRAM_ROW
Passed?
NO
YES
RowID ++
NO
RowID >= TotalRows
YES
Return PASS
Return FAIL

Figure 14 **Flowchart of the program application flash step**

Code Listing 5 Pseudocode – Step 5. Program application flash

```
// Program all rows of the application flash. Flash Size must be provided.
totalRows = FlashSize / ROW_SIZE;
for (rowID = 0; rowID < totalRows; rowID++) {
  flashStartAddr = MEM_BASE_FLASH + rowID * ROW_SIZE;
  // 1. Extract 512-byte row from the data file from address "flashStartAddr" and
  // put into buffer. FILE_ReadData API must be implemented by Programmer.
  data[] = FILE_ReadData(flashStartAddr, ROW_SIZE);

  // 2. Prepare ProgramRow SROM funtion
  // SRAM_SCRATCH        : OpCode
  // SRAM_SCRATCH + 0x04: Data location/size and Integrity check
  // SRAM_SCRATCH + 0x08: Flash address to be programmed
  // SRAM_SCRATCH + 0x0C: Pointer to the first data byte location
  WriteIO (SRAM_SCRATCH_ADDR, SROMAPI_PROGRAMROW_CODE);
  params =
      (6 <<  0) +   // Data size: 6 - 512b
      (1 <<  8) +   // Data location: 1 - SRAM
      (0 << 16) +   // Verify row: 0 - Data integrity check is not performed
      (0 << 24);    // Not used
  WriteIO (SRAM_SCRATCH_ADDR + 0x04, params);
  WriteIO (SRAM_SCRATCH_ADDR + 0x08, flashStartAddr);
  dataRamAddr = SRAM_SCRATCH_ADDR + 0x10;
  WriteIO (SRAM_SCRATCH_ADDR + 0x0C, dataRamAddr);
  // Load row bytes into SRAM buffer
  for (i = 0; i < ROW_SIZE|| i < sizeof(data); i += 4) {
    dataWord = (data[i + 3] << 24) +
               (data[i + 2] << 16) +
               (data[i + 1] << 8)  +
               (data[i] << 0);
    WriteIO (dataRamAddr + i, dataWord);
  }

  // 3. Call ProgramRow SROM function
  status = CallSromApi(SROMAPI_PROGRAMROW_CODE, OUT DataOut);
  if (!status) return FAIL;
}
return PASS;
```

5.9 Step 6 – Verify application flash

Because the checksum is verified eventually and cannot completely guarantee that the content is written without errors, this step should be kept in the programming flow for higher reliability.

During verification, the programmer reads a row from flash and the corresponding data from the binary file and compares them. If any difference is found, the programmer must stop and return a failure. Each row must be considered.

Reading from the flash is achieved by direct access to the memory space of the CPU. No SROM functions are required; simply read the word (32 bits) from the address range 0x10000000 to 0x10000000 + FlashSize – 4. For example, Figure 15 illustrates the verification algorithm.

```
ReadIO (0x10000000, OUT flashWord);
ReadIO (0x10000004, OUT flashWord);
…
ReadIO (0x100FFFFC, OUT flashWord);
```

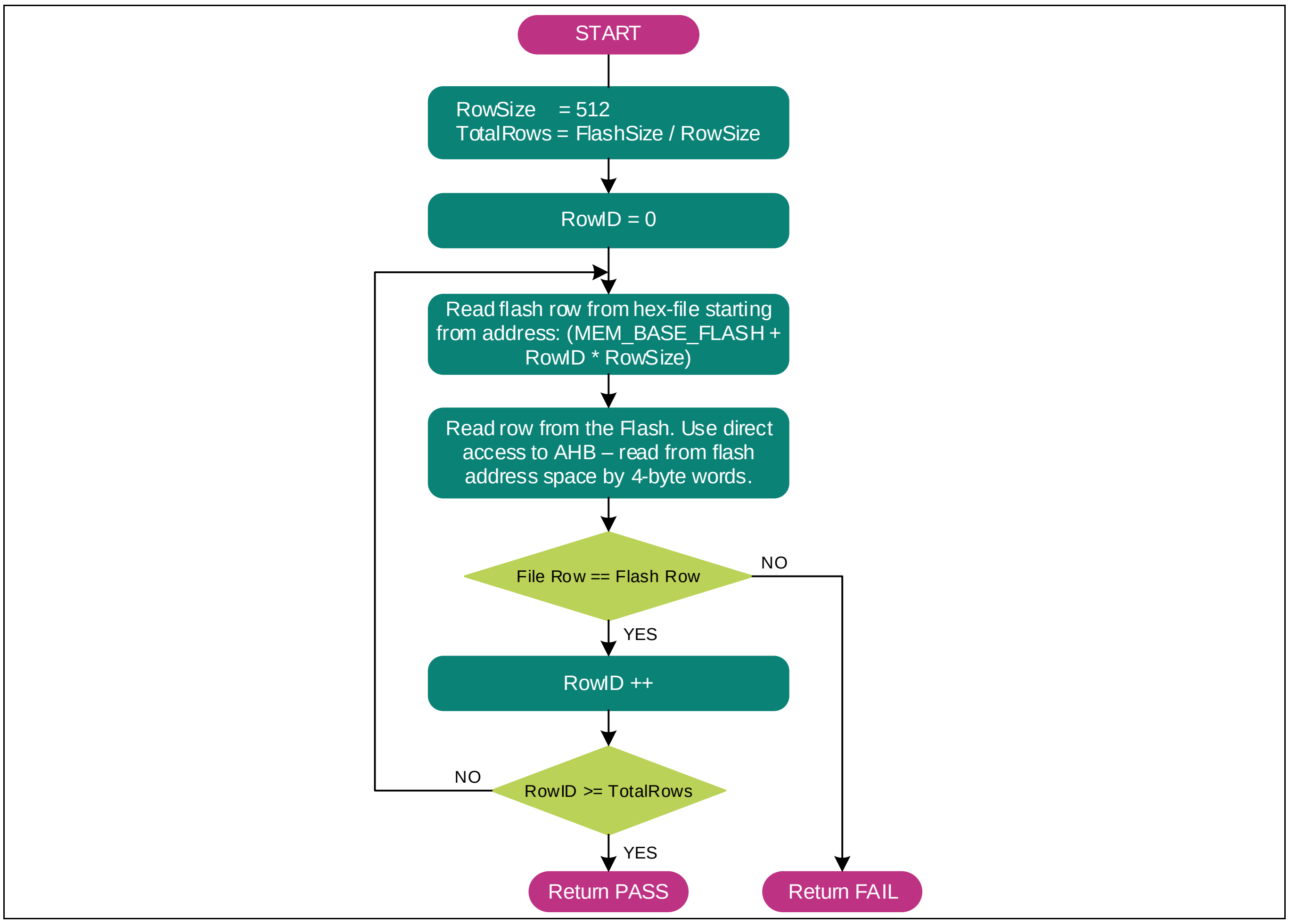

START
RowSize = 512
TotalRows = FlashSize / RowSize
RowID = 0
Read flash row from hex-file starting from address: (MEM_BASE_FLASH + RowID * RowSize)
Read row from the Flash. Use direct access to AHB – read from flash address space by 4-byte words.
File Row == Flash Row
NO
YES
RowID ++
RowID >= TotalRows
NO
YES
Return PASS
Return FAIL

Figure 15 **Flowchart of the verify application flash step**

Code Listing 6 **Pseudocode – Step 6. Verify application flash**

```
// Read and Verify Flash rows. Flash Size must be provided
totalRows = FlashSize / ROW_SIZE;
for (int rowID = 0; rowID < totalRows; rowID++) {
  //1. Read and Extract 512-byte row from the binary file into buffer.
  //   FILE_ReadData API must be implemented by Programmer.
  rowAddress = MEM_BASE_FLASH + rowID * ROW_SIZE; // line address of row in flash
  fileData[] = FILE_ReadData (rowAddress, ROW_SIZE);

  //2. Read row from chip via AHB-interface
  for (i = 0; i < ROW_SIZE; i += 4) {
    ReadIO (rowAddress + i, OUT dataOut);
    chipData[i + 0] = (dataOut >>  0) & 0xFF;
    chipData[i + 1] = (dataOut >>  8) & 0xFF;
    chipData[i + 2] = (dataOut >> 16) & 0xFF;
    chipData[i + 3] = (dataOut >> 24) & 0xFF;
  }
  //3. Compare them
  for (i = 0; i < ROW_SIZE; i++)
    if (chipData[i] != fileData[i]) return FAIL;
}
return PASS;
```

5.10 Step 7 – Verify checksum (optional)

This step validates the result of the flash programming process. It calculates the checksum of the user rows written in Step 5 (Checksum SROM function is used) and compares this value with the 2-byte checksum from the metadata section (if it exists) in the input data file. The checksum operation cannot completely guarantee that the data is written correctly. For this reason, the verify flash step is also recommended.

5.10.1 Pseudocode – Step 7. Verify checksum

```
// Use Checksum SROM function to get the checksum of whole application flash
// Bytes 1 and 2 of the parameters select whether the checksum
// is performed on the whole flash, or a row of flash.
// The row of flash is determined by the Row Id Lo and Row Id Hi parameters.
opCode = SROMAPI_CHECKSUM_CODE +
    (0 << 22) +         // Flash region: 0 – application, 1 - AUXflash, 2 - Sflash
    (1 << 21) +         // Whole flash: 0 – page, 1 – whole flash
    ((0 & 0x1FFF) << 8);// Row id Bits[20:8]
status = CallSromApi (opCode, OUT dataOut);
if(!status) return FAIL;
checksum = dataOut & SROMAPI_CHECKSUM_DATA_MSK; // skip SROM function status

// Read 2-byte checksum of user code from input data file and compare
fileChecksum = FILE_ReadChecksum(); // API must be implemented by Programmer.
return ((checksum & 0xFFFF) == fileChecksum)? PASS : FAIL;
```

5.11 Step 8 – Program and verify AUXflash

The Auxiliary flash (AUXflash) can store application-specific information and can be programmed using the same SROM functions as the application flash. However, the ProgramRow SROM function, used for the application flash programming, requires the flash row to be in the erased state, which is satisfied by performing the Erase Application Flash step. However, that step erases only the application flash region.

The WriteRow SROM function is used for AUXflash programming. This API includes an erase row step in its execution flow, so does not require the flash row to be erased in advance.

AUXflash rows are not stored in the binary file by default, so if your workflow requires that the AUXflash data be in the binary file, the linker scripts should contain an appropriate section (starting at 0x1400 0000), and the data for this section should be properly defined. Alternatively, the user application can update the AUXflash region whenever needed (CPU access via SROM API) - for example, to store calibration data, non-volatile parameters, and so on.

The AUXflash rows are the same size as the rows in the application flash region (512 bytes) and are mapped to the CPU's address space in the range 0x1400 0000 - 0x1400 7FFF. Therefore, the user application can read these rows directly from these addresses.

Figure 16 illustrates the programming algorithm for the AUXflash and Sflash regions.

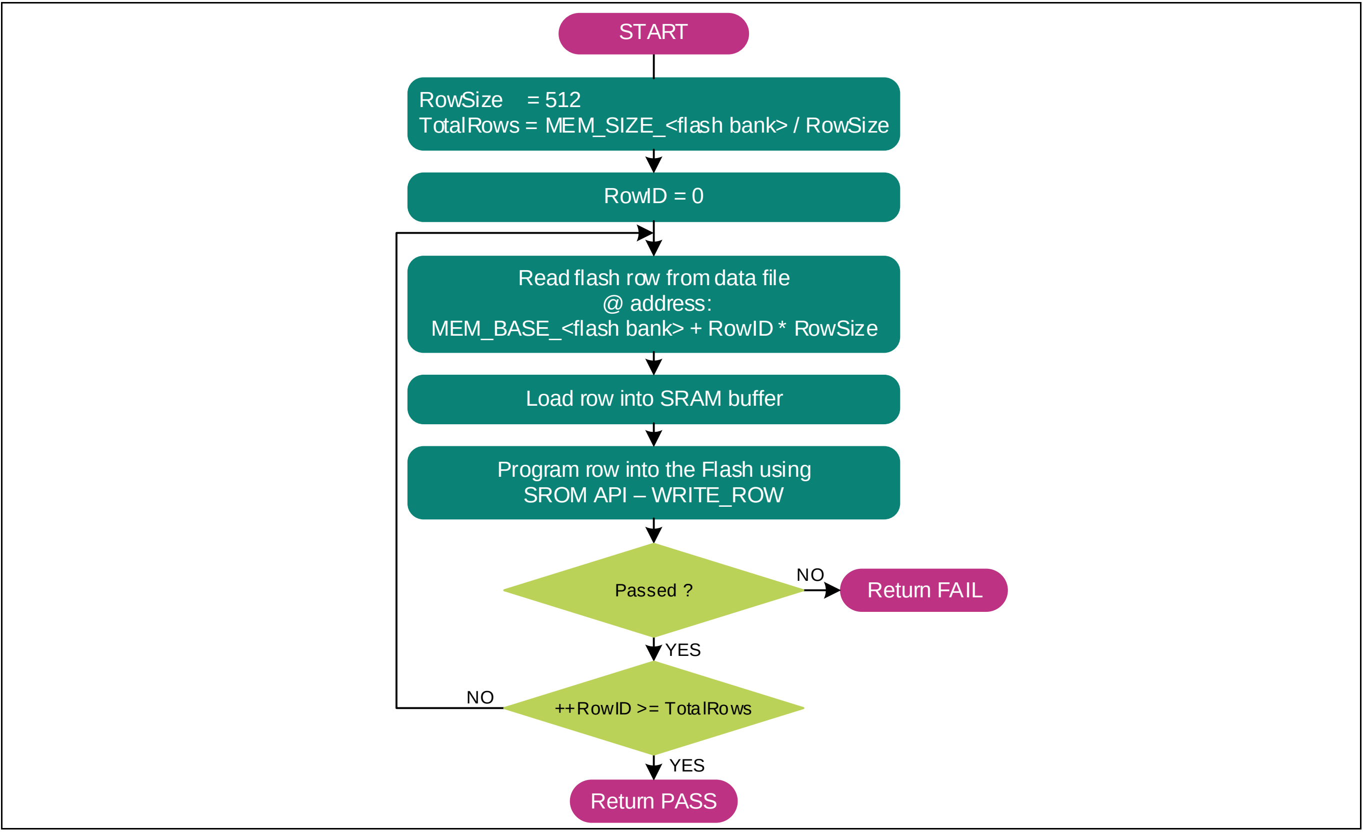

START
RowSize = 512
TotalRows = MEM_SIZE_<flash bank> / RowSize
RowID = 0
Read flash row from data file @ address: MEM_BASE_<flash bank> + RowID * RowSize
Load row into SRAM buffer
Program row into the Flash using SROM API – WRITE_ROW
Passed ?
NO
Return FAIL
YES
++RowID >= TotalRows
NO
YES
Return PASS

Figure 16 Flowchart of "program and verify AUXflash" step

Code Listing 7 Pseudocode. Step 8 – Program and verify AUXflash

```
// This is an entry point of the “Program and Verify AUXflash” step, which programs
// and verifies “User area” AUXflash region (sixty-four rows starting @ 0x14000000)

totalRows = MEM_SIZE_AUXFLASH / ROW_SIZE;
for (int towID = 0; rowID < totalRows; rowID++) {
  address = MEM_BASE_AUXFLASH + rowID * ROW_SIZE
  status = WriteFlashRowFromFile (address);
  if (!status) return FAIL;
  status = CompareFlashRowWithFile (address);
  if (!status) return FAIL;
}

//--- AUXflash and Sflash programming and verification subroutines

WriteFlashRowFromFile (address) {
  // WriteFlashRowFromFile subroutine extracts 512 bytes of data (one flash row)
  // from the input binary file and programs it to the flash bank at provided
  // address, using the WriteRow SROM function.

  // Read flash data from binary file. Function must be implemented and return
  // requested number of bytes starting at provided address. Missed data must be
  // padded to the size of flash row (512 bytes) with the flash erase value (‘0’).
  data[] = FILE_ReadData(address, ROW_SIZE);
  if (sizeof(data) < ROW_SIZE) return PASS; // No data in file at given address

  // Prepare WriteRow SROM function
  // SRAM_SCRATCH: OpCode
  // SRAM_SCRATCH + 0x04: Data location/size and Integrity check
  // SRAM_SCRATCH + 0x08: Flash address to be programmed
  // SRAM_SCRATCH + 0x0C: Pointer to the first data byte location
  WriteIO (SRAM_SCRATCH_ADDR, SROMAPI_WRITEROW_CODE);
  params = (6 <<  0) + // Data size: 6 - 512b
           (1 <<  8) + // Data location: 1 - SRAM
           (0 << 16) + // Verify row: 0 - Data integrity check is not performed
           (0 << 24);  // Not used
  WriteIO (SRAM_SCRATCH_ADDR + 0x04, params);
  WriteIO (SRAM_SCRATCH_ADDR + 0x08, address);
  dataRamAddr = SRAM_SCRATCH_ADDR + 0x10;
  WriteIO (SRAM_SCRATCH_ADDR + 0x0C, dataRamAddr);
  // Load Row bytes into SRAM buffer
  for (i = 0; i < ROW_SIZE || i < sizeof(data); i += 4) {
    dataWord = (data[i + 3] << 24) +
               (data[i + 2] << 16) +
               (data[i + 1] <<  8) +
               (data[i]);
    WriteIO (dataRamAddr + i, dataWord);
  }

  // Call WriteRow SROM function
  status = CallSromApi(SROMAPI_WRITEROW_CODE, OUT dataOut);
  if (!status) return FAIL;
  return PASS;
}

CompareFlashRowWithFile (address) {
  // CompareFlashRowWithFile subroutine extracts 512 bytes (row) of data from the
  // input binary file and compares it to the data read from target’s flash

  // Read flash data from binary file. Function must be implemented and return
  // requested number of bytes starting at provided address. Missed data must be
  // padded to the size of flash row (512 bytes) with the flash erase value (‘0’).
  fileData[] = FILE_ReadData(address, ROW_SIZE);
  if (sizeof(fileData) < ROW_SIZE) return PASS; // No data in file at given address

  // Read row of data (512 bytes) from target device via AHB-interface
  for (i = 0; i < ROW_SIZE; i += 4)  {
    ReadIO (address + i, OUT dataOut);
    chipData[i + 0] = (dataOut >>  0) & 0xFF;
    chipData[i + 1] = (dataOut >>  8) & 0xFF;
    chipData[i + 2] = (dataOut >> 16) & 0xFF;
    chipData[i + 3] = (dataOut >> 24) & 0xFF;
  }

  // Compare data byte in device with the data byte in file
  for (i = 0; i < ROW_SIZE; i++)
    if (chipData[i] != fileData[i]) return FAIL;
}
```

5.12 Step 9 – Program and verify Sflash

The "User area" in the supervisory flash region (Sflash) of the PSOC™ 6 MCU can store various application-specific data such as calibration data, non-volatile parameters, and so on. The Sflash also includes several programmable sub-regions (such as NAR, Public Key, and TOC2) defined for usage by the boot process and user application for device initialization, protection and data integrity validation purposes. Refer to section 2.3 for the address ranges in the Sflash region that can be programmed.

Sflash can be programmed using an approach like Auxiliary flash (AUXflash). The Sflash rows are the same size as the rows in the application flash region (512 bytes) and are mapped to the CPU's address space. Therefore, a user application can read these rows directly from these addresses.

Sflash rows are not stored in the binary file by default, so if your workflow requires that the Sflash data be in the binary file, the linker scripts must contain appropriate sections and the data for these sections must be properly defined. Alternatively, the user application can update the Sflash region whenever needed (CPU access via SROM API). During mass production, a vendor should define the programming process—where to get Sflash data and at which row/address to store it.

Code Listing 8 Pseudocode. Step 9 - Program and verify Sflash

```
// This is an entry point of the "Program and Verify Sflash" step.
// Program and verify "User area" sub-region (four rows starting @ 0x16000800)
for (int towID = 0; rowID < 4; rowID++) {
  address = 0x16000800 + rowID * ROW_SIZE
  status = WriteFlashRowFromFile   (address); if (!status) return FAIL;
  status = CompareFlashRowWithFile (address); if (!status) return FAIL;
}

// Program and verify "NAR" sub-region (one row starting @ 0x16001A00)
address = 0x16001A00 + rowID * ROW_SIZE
status = WriteFlashRowFromFile   (address); if (!status) return FAIL;
status = CompareFlashRowWithFile (address); if (!status) return FAIL;

// Program and verify "Public Key" sub-region (six rows starting @ 0x16005A00)
for (int towID = 0; rowID < 6; rowID++) {
  address = 0x16005A00 + rowID * ROW_SIZE
  status = WriteFlashRowFromFile   (address); if (!status) return FAIL;
  status = CompareFlashRowWithFile (address); if (!status) return FAIL;
}

// Program and verify "TOC2"/"RTOC2" sub-regions (two rows starting @ 0x16007C00)
for (int towID = 0; rowID < 2; rowID++) {
  address = 0x16005A00 + rowID * ROW_SIZE
  status = WriteFlashRowFromFile   (address); if (!status) return FAIL;
  status = CompareFlashRowWithFile (address); if (!status) return FAIL;
}
```

5.13 Step 10 – Program eFuse

At this point, the programmer writes protection and user-defined data into eFuse memory: Life Cycle stage, Secure HASH and HASH Zeros, Secure, and Dead Access restrictions, and Customer Data. A warning message should be provided when eFuses are being programmed because after they are programmed, they cannot be changed.

eFuse memory can be logically divided into two parts: Protection settings and Customer Data. The difference is that the Protection settings are intended to govern the protection of the chip, and Customer Data is dedicated to customer usage, for example, to store cryptographic keys or part identification information. Customer Data programming must be done before protection settings since eFuse programming is locked once the chip is moved to the Secure state.

The eFuse memory starts in address 32’h9070 0000 in the binary file and can contain the next fields: Life Cycle stage, Secure and Dead Access Restriction properties, and Customer Data. The Secure HASH and HASH Zeros are calculated run-time right before the Lifecycle stage transition.

The eFuse programming must be the last step in the programming flow, the Table of Contents and all data included in the HASH verification must be programmed before this step. If this data is changed when the Secure bit is blown, the PSOC™ 6 MCU will go to the Dead state.

Figure 17 shows the sequence to program the protection settings in eFuse.

See the “Device Security” and “Protection Units” sections of the reference manual for more details on the security settings for the PSOC™ 6 MCU.

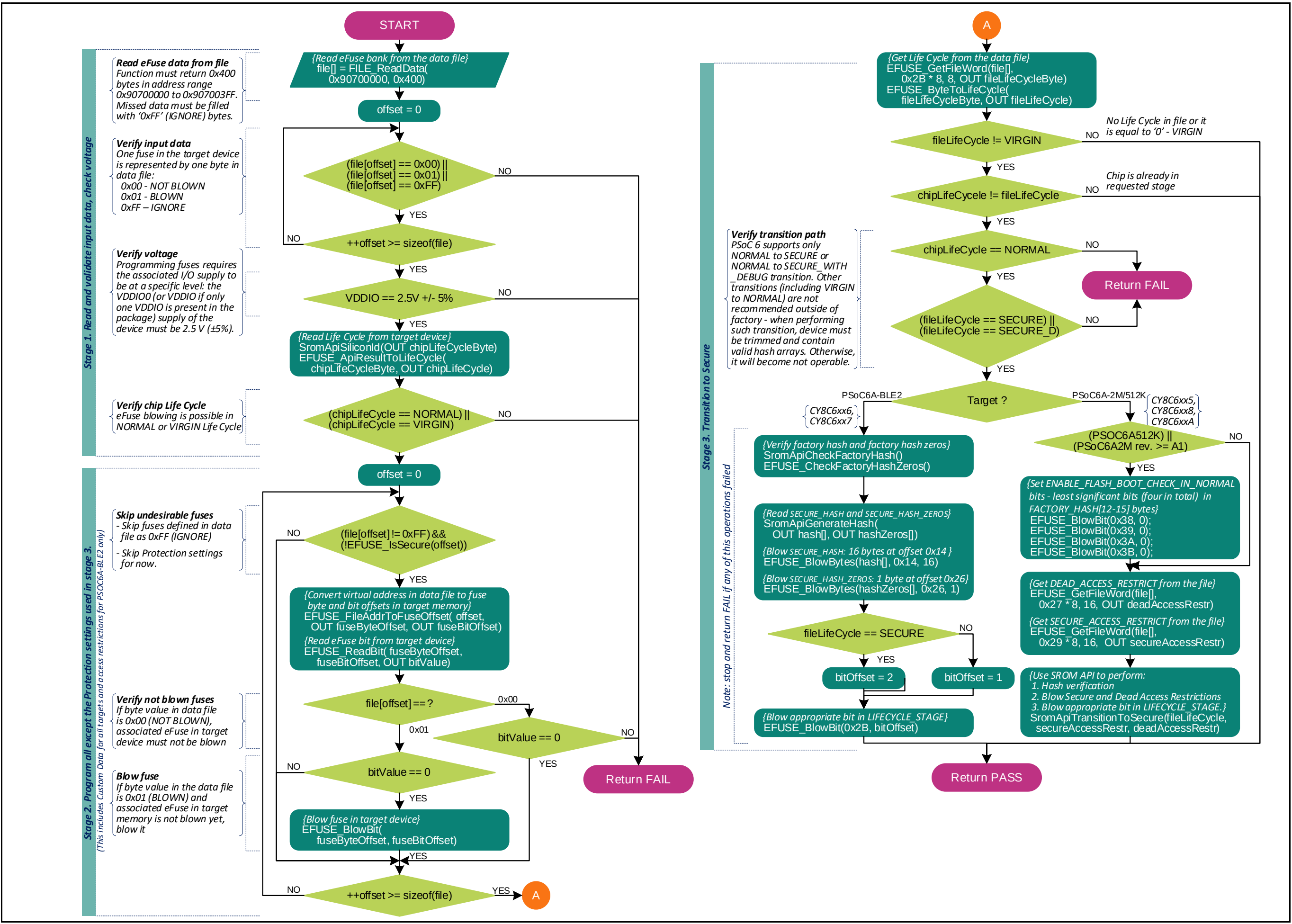

START
Stage 1. Read and validate input data, check voltage
Read eFuse data from file
Function must return 0x400 bytes in address range 0x90700000 to 0x907003FF. Missed data must be filled with ‘0xFF’ (IGNORE) bytes.
{Read eFuse bank from the data file}
file[] = FILE_ReadData(
0x90700000, 0x400)
offset = 0
Verify input data
One fuse in the target device is represented by one byte in data file:
0x00 - NOT BLOWN
0x01 - BLOWN
0xFF – IGNORE
(file[offset] == 0x00) ||
(file[offset] == 0x01) ||
(file[offset] == 0xFF)
NO
YES
++offset >= sizeof(file)
NO
YES
Verify voltage
Programming fuses requires the associated I/O supply to be at a specific level: the VDDIO0 (or VDDIO if only one VDDIO is present in the package) supply of the device must be 2.5 V (±5%).
VDDIO == 2.5V +/- 5%
NO
YES
{Read Life Cycle from target device}
SromApiSiliconId(OUT chipLifeCycleByte)
EFUSE_ApiResultToLifeCycle(
chipLifeCycleByte, OUT chipLifeCycle)
Verify chip Life Cycle
eFuse blowing is possible in NORMAL or VIRGIN Life Cycle
(chipLifeCycle == NORMAL) ||
(chipLifeCycle == VIRGIN)
NO
offset = 0
Stage 2. Program all except the Protection settings used in stage 3.
(This includes Custom Data for all targets and access restrictions for PSOC6A-BLE2 only)
Skip undesirable fuses
- Skip fuses defined in data file as 0xFF (IGNORE)
- Skip Protection settings for now.
(file[offset] != 0xFF) &&
(!EFUSE_IsSecure(offset))
NO
YES
{Convert virtual address in data file to fuse byte and bit offsets in target memory}
EFUSE_FileAddrToFuseOffset(offset,
OUT fuseByteOffset, OUT fuseBitOffset)
{Read eFuse bit from target device}
EFUSE_ReadBit(fuseByteOffset,
fuseBitOffset, OUT bitValue)
Verify not blown fuses
If byte value in data file is 0x00 (NOT BLOWN), associated eFuse in target device must not be blown
file[offset] == ?
0x00
0x01
bitValue == 0
NO
YES
Return FAIL
Blow fuse
If byte value in the data file is 0x01 (BLOWN) and associated eFuse in target memory is not blown yet, blow it
bitValue == 0
NO
YES
{Blow fuse in target device}
EFUSE_BlowBit(
fuseByteOffset, fuseBitOffset)
YES
++offset >= sizeof(file)
NO
YES
A
A
Stage 3. Transition to Secure
{Get Life Cycle from the data file}
EFUSE_GetFileWord(file[],
0x2B * 8, 8, OUT fileLifeCycleByte)
EFUSE_ByteToLifeCycle(
fileLifeCycleByte, OUT fileLifeCycle)
fileLifeCycle != VIRGIN
NO
No Life Cycle in file or it is equal to ‘0’ - VIRGIN
YES
chipLifeCycele != fileLifeCycle
NO
Chip is already in requested stage
YES
Verify transition path
PSoC 6 supports only NORMAL to SECURE or NORMAL to SECURE_WITH _DEBUG transition. Other transitions (including VIRGIN to NORMAL) are not recommended outside of factory - when performing such transition, device must be trimmed and contain valid hash arrays. Otherwise, it will become not operable.
chipLifeCycle == NORMAL
NO
YES
Return FAIL
(fileLifeCycle == SECURE) ||
(fileLifeCycle == SECURE_D)
NO
YES
Target ?
PSoC6A-BLE2
CY8C6xx6,
CY8C6xx7
PSoC6A-2M/512K
CY8C6xx5,
CY8C6xx8,
CY8C6xxA
Note: stop and return FAIL if any of this operations failed
{Verify factory hash and factory hash zeros}
SromApiCheckFactoryHash()
EFUSE_CheckFactoryHashZeros()
{Read SECURE_HASH and SECURE_HASH_ZEROS}
SromApiGenerateHash(
OUT hash[], OUT hashZeros[])
{Blow SECURE_HASH: 16 bytes at offset 0x14 }
EFUSE_BlowBytes(hash[], 0x14, 16)
{Blow SECURE_HASH_ZEROS: 1 byte at offset 0x26}
EFUSE_BlowBytes(hashZeros[], 0x26, 1)
fileLifeCycle == SECURE
NO
YES
bitOffset = 2
bitOffset = 1
{Blow appropriate bit in LIFECYCLE_STAGE}
EFUSE_BlowBit(0x2B, bitOffset)
(PSOC6A512K) ||
(PSoC6A2M rev. >= A1)
NO
YES
{Set ENABLE_FLASH_BOOT_CHECK_IN_NORMAL bits - least significant bits (four in total) in FACTORY_HASH[12-15] bytes}
EFUSE_BlowBit(0x38, 0);
EFUSE_BlowBit(0x39, 0);
EFUSE_BlowBit(0x3A, 0);
EFUSE_BlowBit(0x3B, 0);
{Get DEAD_ACCESS_RESTRICT from the file}
EFUSE_GetFileWord(file[],
0x27 * 8, 16, OUT deadAccessRestr)
{Get SECURE_ACCESS_RESTRICT from the file}
EFUSE_GetFileWord(file[],
0x29 * 8, 16, OUT secureAccessRestr)
{Use SROM API to perform:
1. Hash verification
2. Blow Secure and Dead Access Restrictions
3. Blow appropriate bit in LIFECYCLE_STAGE.}
SromApiTransitionToSecure(fileLifeCycle,
secureAccessRestr, deadAccessRestr)
Return PASS

Figure 17 Flowchart of the “program eFuse” step

Code Listing 9 Pseudocode – Step 10. Program eFuse

```
// This is an entry point of the "Program eFuse" step.
// Pseudo-code is simplified for better readability - make sure to stop execution
// and return FAIL status in case any operation failed.
// --- Stage 1. Read and validate input data, check voltage and Life Cycle stage

// Read eFuse data from binary file. Function must be implemented and return
// 0x400 bytes in address range 0x90700000 to 0x907003FF.
// Missed data must be filled with 0xFF (IGNORE).
file[] = FILE_ReadData (0x90700000, 0x400);
// Verify input data. One fuse in the target device is represented by one byte
// in file: 0x00 - NOT BLOWN, 0x01 – BLOWN, 0xFF – IGNORE
for (int offset = 0; offset < sizeof(file); offset ++)
  if ((file[offset] != 0x00) && (file[offset] != 0x01) && (file[offset] != 0xFF))
    return FAIL;

// Verify voltage. Programming fuses requires the associated I/O supply to be at
// a specific level: the VDDIO0 (or VDDIO if only one VDDIO is present in the
// package) supply of the device must be 2.5 V (±5%). Function must be implemented.
VerifyVddio();

// Verify chip Life Cycle. eFuse blowing is possible in NORMAL or VIRGIN Life Cycle
CallSromApi (SROMAPI_SILID_CODE + (0x0000FF00 & (1 << 8)), OUT dataOut);
chipLifeCycleByte = (dataOut & 0x00F00000) >> 20; // Life cycle stage
EFUSE_ApiResultToLifeCycle (chipLifeCycleByte, OUT chipLifeCycle);
if ((chipLifeCycle != VIRGIN) && (chipLifeCycle != NORMAL)) return FAIL;

// --- Stage 2. Program all except the Protection settings used in stage 3

for (int offset = 0; offset < sizeof(file); offset ++) {
  // Skip fuses defined in file as 0xFF (IGNORE)
  // and skip (for now) Protection settings.
  if ((file[offset] == 0xFF) || EFUSE_IsSecure(offset)) continue;

  // Convert virtual address in file to fuse byte and bit offsets in target
  // memory and read associated fuse from target device
  EFUSE_FileAddrToFuseOffset (offset, OUT fuseByteOffset, OUT fuseBitOffset);
  EFUSE_ReadBit (fuseByteOffset, fuseBitOffset, OUT bitValue);

  if (file[offset] == 0x00 /* NOT BLOWN */) {
    // If byte value in file is 0, associated fuse in device must not be blown
    if (bitValue != 0x00) return FAIL;
  }
  else if (bitValue == 0x00) {
    // Blow fuse in device if associated byte value in the file is 0x01 (BLOWN)
    // and the fuse is not blown yet
    EFUSE_BlowBit (fuseByteOffset, fuseBitOffset);
  }
}

// --- Stage 3. Transition to Secure ---

// Get Life Cycle from the file
EFUSE_GetFileWord (file[], 0x2B * 8, 8, OUT fileLifeCycleByte);
EFUSE_LifeCycleByteToEnum (fileLifeCycleByte, OUT fileLifeCycle);
// Skip Protection settings programming if no Life Cycle in data file or it is
// equal to 0x00 (VIRGIN) or if chip is already in requested stage
if ((fileLifeCycle == VIRGIN)||(fileLifeCycle == chipLifeCycle)) return PASS;

// Verify transition path. PSOC 6 supports only NORMAL to SECURE or NORMAL to
// SECURE_WITH_DEBUG transition. Other transitions (including VIRGIN to NORMAL)
// are not recommended outside of factory and may result in not operable device.
if (chipLifeCycle != NORMAL) return FAIL;
if ((fileLifeCycle != SECURE_WITH_DEBUG)&&(fileLifeCycle != SECURE)) return FAIL;
if (DeviceFamily == PSOC6ABLE2) { // PSOC6A-BLE2
  // Verify FACTORY_HASH and FACTORY_HASH_ZEROS.
  // APIs returns FAIL if verification failed.
  CallSromApi (SROMAPI_CHECK_FACTORY_HASH_CODE, OUT dataOut);
  EFUSE_CheckFactoryHashZeros();

  // Generate SECURE_HASH and SECURE_HASH_ZEROS using the SROM API
  WriteIO (SRAM_SCRATCH_ADDR, SROMAPI_GENERATE_HASH_CODE);
  CallSromApi (SROMAPI_GENERATE_HASH_CODE, OUT dataOut);
  for (int i = 0; i < 4; i++) {
    ReadIO (SRAM_SCRATCH_ADDR + 4*(i+1), OUT dataOut);
    hash[4*i + 0] = (dataOut >>  0) & 0xFF;
    hash[4*i + 1] = (dataOut >>  8) & 0xFF;
    hash[4*i + 2] = (dataOut >> 16) & 0xFF;
    hash[4*i + 3] = (dataOut >> 24) & 0xFF;
  }
  ReadIO (SRAM_SCRATCH_ADDR + 20, OUT dataOut);
  hashZeros = dataOut & 0xFF;

  // Blow SECURE_HASH and Blow SECURE_HASH_ZEROS
  EFUSE_BlowBytes (hash[], 0x14, 16);      // 16 bytes at offset 0x14
  EFUSE_BlowBytes (hashZeros[], 0x26, 1); // 1 byte at offset 0x26

  // Blow appropriate bit in LIFECYCLE_STAGE
  bitOffset = (fileLifeCycle == SECURE)? 2 : 1 /*SECURE_WITH_DEBUG*/;
  EFUSE_BlowBit (0x2B, bitOffset);
}
else { // PSOC6A-2M/512K/256K (CY8C6xx4, CY8C6xx5, CY8C6xx8 and CY8C6xxA)
  // Applicable for PSOC6A-256K, PSOC6A-512K, PSOC6A-2M starting at rev. A1(*A):
  // Set ENABLE_FLASH_BOOT_CHECK_IN_NORMAL bits - least significant bits (four in
  // total) in FACTORY_HASH[12-15] bytes just before TransitionToSecure API call.
  if ((DeviceFamily == PSOC6A256K)||( DeviceFamily == PSOC6A512K)||
        ((DeviceFamily == PSOC6A2M)&&(RevisionId >= 0x12)))
    for (int i = 0; i < 4; i++)
      EFUSE_BlowBit(0x3B - i; /* FACTORY_HASH15 - i */, 0);

  // Get restriction bits from the binary file:
  // DEAD_ACCESS_RESTRICT:   16 bytes @ offset 0x27
  // SECURE_ACCESS_RESTRICT: 16 bytes @ offset 0x29
  EFUSE_GetFileWord(file[], 0x27 * 8, 16, OUT deadAccessRestr);
  EFUSE_GetFileWord(file[], 0x29 * 8, 16, OUT secureAccessRestr);

  // TransitionToSecure SROM function validates the FACTORY_HASH and programs
  // SECURE_HASH, secure access restrictions and dead access restrictions into
  // eFuse. Then, it programs  secure or secure with debug fuse for corresponding
  // LifeCycle stage transition.
  opCode = SROMAPI_TRANSITION_TO_SECURE_CODE;
  if (fileLifeCycle == SECURE_WITH_DEBUG) opCode |= 1 << 8;
  WriteIO (SRAM_SCRATCH_ADDR, opCode);
  WriteIO (SRAM_SCRATCH_ADDR + 0x04, secureAccessRestr);
  WriteIO (SRAM_SCRATCH_ADDR + 0x08, deadAccessRestr);
  CallSromApi (opCode, OUT dataOut);
}
```

Code Listing 10 Pseudocode – Step 10. Program eFuse – subroutines

```
//--- eFuse verification subroutines

EFUSE_CheckFactoryHashZeros (void) {
  // Checks FACTORY_HASH_ZEROS for PSOC6A-BLE2

  // Read FACTORY_HASH_ZEROS from device (one byte at offset = 0x3C)
  EFUSE_ReadByte (0x3C, OUT hashZerosChip);
  if (hashZerosChip == 0) return PASS; // Earlier silicon revisions

  // Use ReadFuseByte SROM function to read 16 bytes of FACTORY_HASH from device.
  hashZerosCalculated = 0;
  for (BYTE byteOffset = 0; byteOffset < 0x10; byteOffset++) {
    EFUSE_ReadByte (0x2C + byteOffset, OUT byteVal); // base offset = 0x2C
    // Calculate number of 0's in current byte
    onesInByte = 0;
    while (byteVal > 0) {
      if ((byteVal & 1) == 1) onesInByte++; // Check lower bit
      byteVal >>= 1;                        // Shift removing lower bit
    }
    zerosInByte = 8 - onesInByte;
    hashZerosCalculated += zerosInByte;
  }
  return (hashZerosChip == hashZerosCalculated)? PASS : FAIL;
}

EFUSE_IsSecure (offset) {
  // Defines if eFuse at given offset is related to Protection settings
  // and should not be blown at first pass

  // Skip LIFECYCLE_STAGE for PSOC6A-BLE2, PSOC6A-2M, PSOC6A-512K, and PSOC6A-256K
  if (offset == 0x2B) return TRUE; // LIFECYCLE_STAGE
  // Skip DEAD and SECURE Access Restrictions for PSOC6A-2M/512K/256K devices
  if ((DeviceFamily == PSOC6A2M)||( DeviceFamily == PSOC6A512K)||
        (DeviceFamily == PSOC6A256K))
    if ((offset == 0x27)||(offset == 0x28)|| // DEAD_ACCESS_RESTRICT
        (offset == 0x29)||(offset == 0x2A))  // SECURE_ACCESS_RESTRICT
      return TRUE;

  return FALSE;
}
```

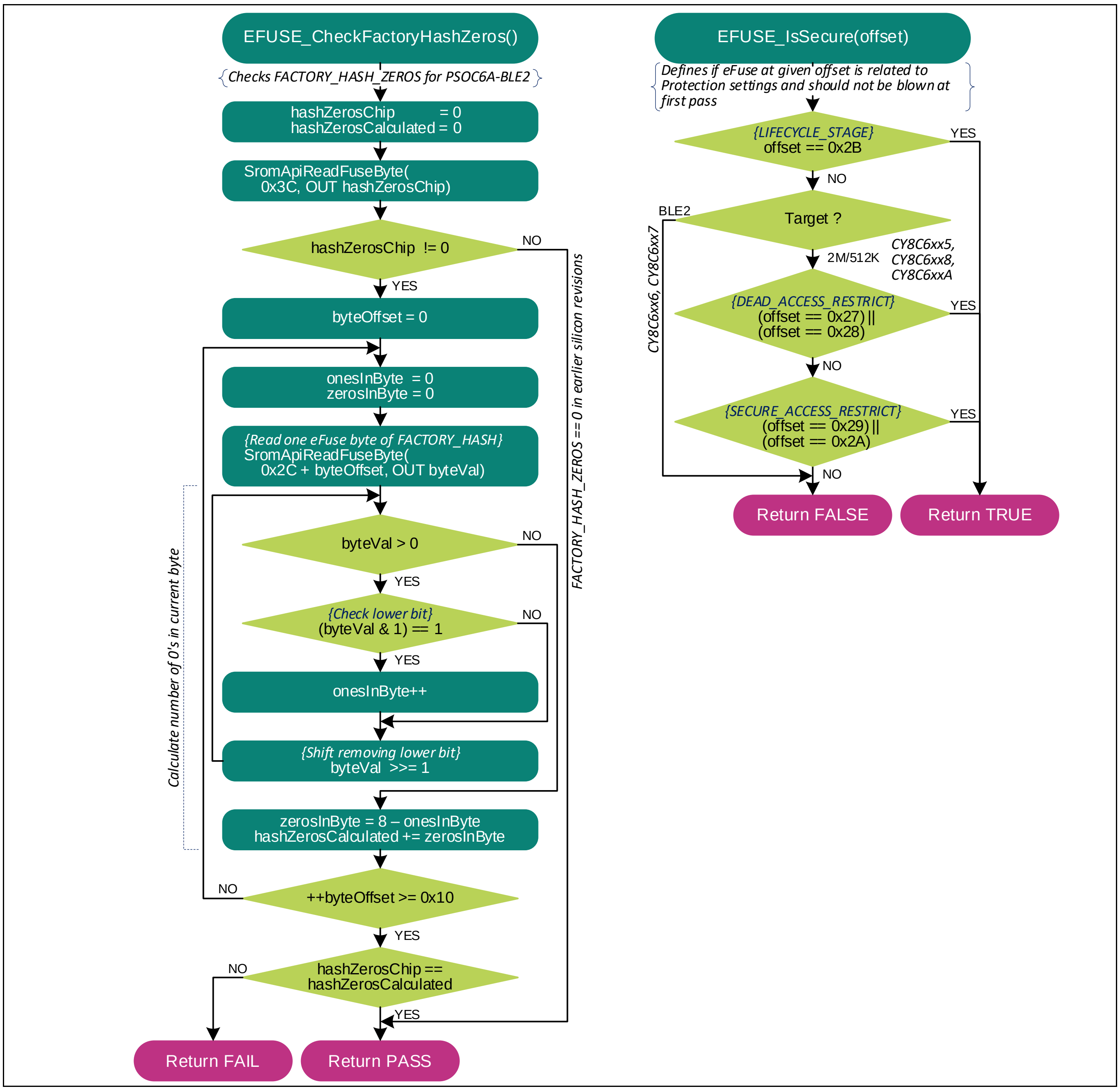

EFUSE_CheckFactoryHashZeros()
{Checks FACTORY_HASH_ZEROS for PSOC6A-BLE2}
hashZerosChip = 0
hashZerosCalculated = 0
SromApiReadFuseByte(0x3C, OUT hashZerosChip)
hashZerosChip != 0
NO
YES
byteOffset = 0
onesInByte = 0
zerosInByte = 0
{Read one eFuse byte of FACTORY_HASH}
SromApiReadFuseByte(0x2C + byteOffset, OUT byteVal)
byteVal > 0
{Check lower bit}
(byteVal & 1) == 1
onesInByte++
{Shift removing lower bit}
byteVal >>= 1
zerosInByte = 8 – onesInByte
hashZerosCalculated += zerosInByte
++byteOffset >= 0x10
hashZerosChip == hashZerosCalculated
Return FAIL
Return PASS
Calculate number of 0's in current byte
FACTORY_HASH_ZEROS == 0 in earlier silicon revisions
EFUSE_IsSecure(offset)
Defines if eFuse at given offset is related to Protection settings and should not be blown at first pass
{LIFECYCLE_STAGE}
offset == 0x2B
Target ?
BLE2
2M/512K
CY8C6xx5, CY8C6xx8, CY8C6xxA
CY8C6xx6, CY8C6xx7
{DEAD_ACCESS_RESTRICT}
(offset == 0x27) ||
(offset == 0x28)
{SECURE_ACCESS_RESTRICT}
(offset == 0x29) ||
(offset == 0x2A)
Return FALSE
Return TRUE

Figure 18 eFuse verification subroutines

```
//--- Life Cycle parsing subroutines

EFUSE_ByteToLifeCycle (lifeCycleByte, OUT lifeCycleEnum) {
  // Converts LIFECYCLE_STAGE bit-field value read directly from eFuse or from the
  // data file to enumerable value.
  // Note: if higher bit is set, all lower bits are ignored
  // LIFECYCLE_STAGE [7:3] - Reserved or not used for production
  // LIFECYCLE_STAGE [2]   - SECURE
  // LIFECYCLE_STAGE [1]   - SECURE_WITH_DEBUG
  // LIFECYCLE_STAGE [0]   - NORMAL
  if (lifeCycleByte == 0) lifeCycleEnum = VIRGIN;
  else if ((lifeCycleByte & 0xF8) != 0) lifeCycleEnum = RESERVED;
  else if ((lifeCycleByte & 0x04) != 0) lifeCycleEnum = SECURE;
  else if ((lifeCycleByte & 0x02) != 0) lifeCycleEnum = SECURE_WITH_DEBUG;
  else lifeCycleEnum = NORMAL;
}

EFUSE_ApiResultToLifeCycle (lifeCycleByte, OUT lifeCycleEnum) {
  // Converts the result of SROM function such as Silicon Id to enumerable value
  if (lifeCycleByte == 0) lifeCycleEnum = VIRGIN;
  else if (lifeCycleByte == 1) lifeCycleEnum = NORMAL;
  else if (lifeCycleByte == 2) lifeCycleEnum = SECURE_WITH_DEBUG;
  else if (lifeCycleByte == 3) lifeCycleEnum = SECURE;
  else lifeCycleEnum = RESERVED; // Reserved or not used for production
}
```

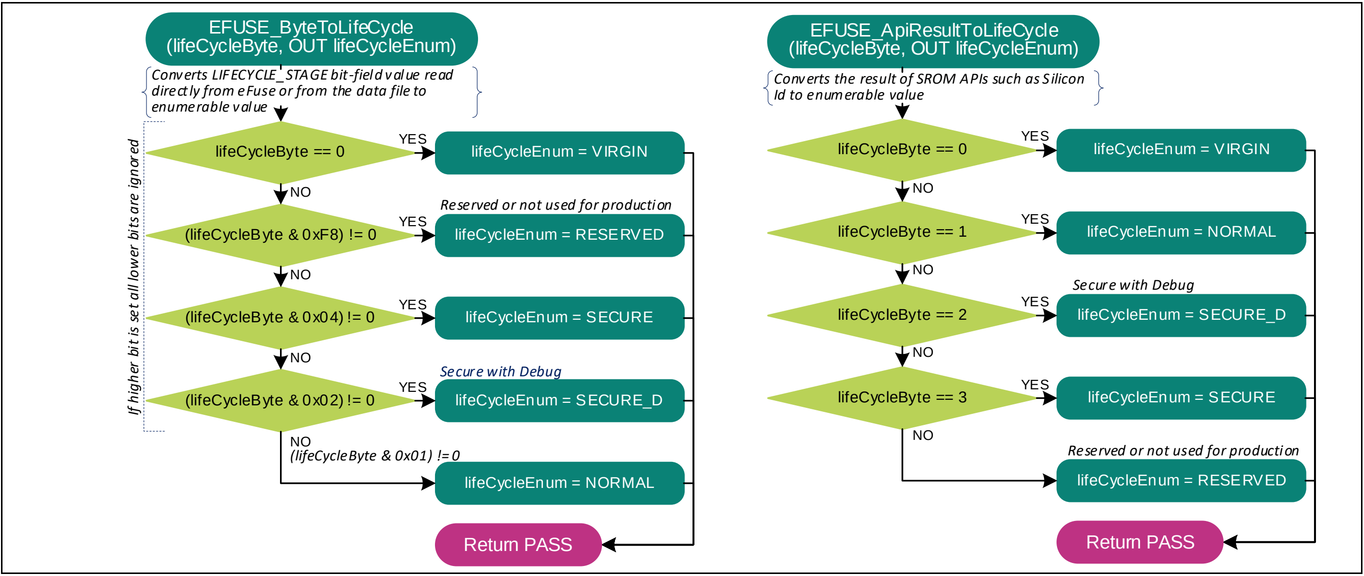

EFUSE_ByteToLifeCycle (lifeCycleByte, OUT lifeCycleEnum)
Converts LIFECYCLE_STAGE bit-field value read directly from eFuse or from the data file to enumerable value
If higher bit is set all lower bits are ignored
lifeCycleByte == 0
YES
lifeCycleEnum = VIRGIN
NO
Reserved or not used for production
(lifeCycleByte & 0xF8) != 0
lifeCycleEnum = RESERVED
(lifeCycleByte & 0x04) != 0
lifeCycleEnum = SECURE
Secure with Debug
(lifeCycleByte & 0x02) != 0
lifeCycleEnum = SECURE_D
NO (lifeCycleByte & 0x01) != 0
lifeCycleEnum = NORMAL
Return PASS
EFUSE_ApiResultToLifeCycle (lifeCycleByte, OUT lifeCycleEnum)
Converts the result of SROM APIs such as Silicon Id to enumerable value
lifeCycleByte == 0
lifeCycleEnum = VIRGIN
lifeCycleByte == 1
lifeCycleEnum = NORMAL
Secure with Debug
lifeCycleByte == 2
lifeCycleEnum = SECURE_D
lifeCycleByte == 3
lifeCycleEnum = SECURE
Reserved or not used for production
lifeCycleEnum = RESERVED
Return PASS

Figure 19 **Life cycle parsing subroutines**

```
//--- eFuse file parsing subroutines

EFUSE_GetFileWord (file[], offset, bitsCount, OUT eFuseWord) {
  // Gets eFuse word from the data file. One fuse in the target device is
  // represented by one byte in file: 0x00 - NOT BLOWN, 0x01 - BLOWN, 0xFF - IGNORE
  if (offset >  0x907003FF) return FAIL;
  if (offset >= 0x90700000) offset -= 0x90700000; // Convert address to offset
  eFuseWord = bitOffset = 0;
  while ( (offset < sizeof(file)) && (bitOffset < bitsCount) ) {
    if (file[offset] = 0x01 /*BLOWN*/) eFuseWord |= 1 << bitOffset;
    offset++;
    bitOffset++;
  }
  return PASS;
}

EFUSE_FileAddrToFuseOffset (addr, OUT fuseByteOffset, OUT fuseBitOffset) {
  // Convert virtual address in data file to fuse byte and bit offsets in target
  if (addr >  0x907003FF) return FAIL;
  if (addr >= 0x90700000) addr -= 0x90700000; // Convert address to offset
  fuseByteOffset = addr / 8;
  fuseBitOffset  = addr % 8;
}
```

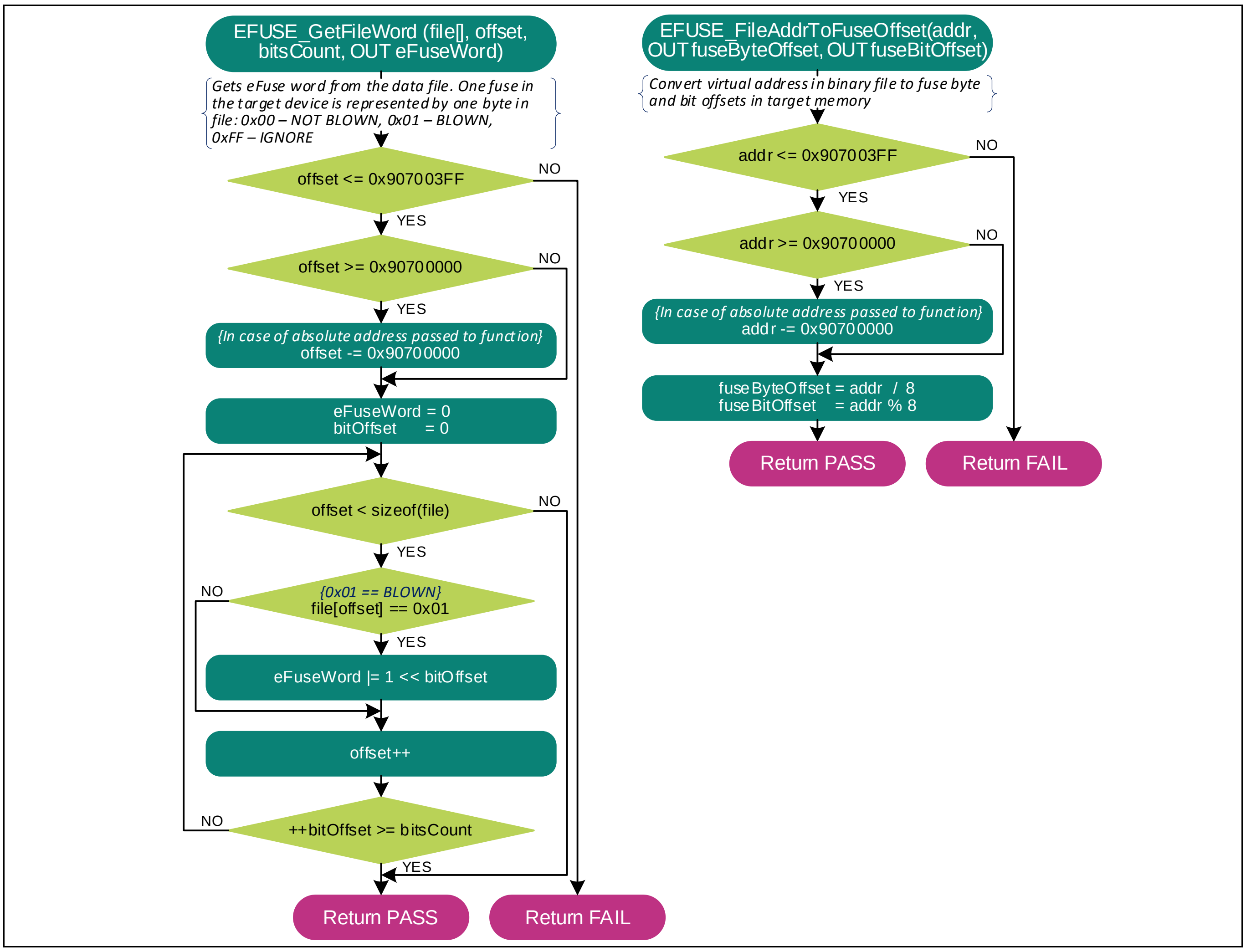

EFUSE_GetFileWord (file[], offset, bitsCount, OUT eFuseWord)
{Gets eFuse word from the data file. One fuse in the target device is represented by one byte in file: 0x00 – NOT BLOWN, 0x01 – BLOWN, 0xFF – IGNORE
offset <= 0x907003FF
NO
YES
offset >= 0x90700000
NO
YES
{In case of absolute address passed to function} offset -= 0x90700000
eFuseWord = 0
bitOffset = 0
offset < sizeof(file)
NO
YES
{0x01 == BLOWN}
file[offset] == 0x01
NO
YES
eFuseWord |= 1 << bitOffset
offset++
++bitOffset >= bitsCount
NO
YES
Return PASS
Return FAIL
EFUSE_FileAddrToFuseOffset(addr, OUT fuseByteOffset, OUT fuseBitOffset)
{Convert virtual address in binary file to fuse byte and bit offsets in target memory
addr <= 0x907003FF
NO
YES
addr >= 0x90700000
NO
YES
{In case of absolute address passed to function} addr -= 0x90700000
fuseByteOffset = addr / 8
fuseBitOffset = addr % 8
Return PASS
Return FAIL

Figure 20 **eFuse file parsing subroutines**

```
//--- eFuse Read/Write subroutines

EFUSE_BlowBytes (data[], eFuseByteOffset, bytesCount) {
  // Blows array of eFuse bytes and checks if the eFuses were blown correctly
  for (dataOffset = 0; dataOffset < bytesCount); dataOffset++)
    for (bitOffset = 0; bitOffset < 8); bitOffset++)
      if ((data[dataOffset] >> bitOffset) & 1 == 1)
        EFUSE_BlowBit (eFuseByteOffset + dataOffset, bitOffset);
}

EFUSE_BlowBit (byteOffset, bitOffset) {
  // Blows the addressed eFuse bit and checks if eFuse was blown correctly

  // Check initial value first and skip blowing if fuse has already been blown
  EFUSE_ReadBit(byteOffset, bitOffset, OUT bitValue);
  if (bitValue == 0) {
  // Calculate macro address and update byte address appropriately
  macroOffset = 0;
  while (byteOffset >= 32 /* eFuse macro size */) {
    byteOffset -= 32;
    macroOffset++;
  }
```

```
   // Call BlowFuseBit SROM function to blow the fuse
   opCode = SROMAPI_BLOW_FUSE_CODE +
     (0x00FF0000 & (byteOffset  << 16)) + // Byte offset in macro
     (0x0000F000 & (macroOffset << 12)) + // Macro Address
     (0x00000700 & (bitOffset   << 8));   // Bit position in Byte
   CallSromApi (opCode, OUT dataOut);

   // Verify the fuse is successfully blown
   EFUSE_ReadBit(byteOffset, bitOffset, OUT bitValue);
   return (bitValue == 1)? PASS : FAIL;
 }

EFUSE_ReadByte (byteOffset, OUT byteValue) {
   // Uses ReadFuseByte SROM function to read eFuse byte (eight fuses) from device
   opCode = SROMAPI_READ_FUSE_CODE + (byteOffset << 8);
   CallSromApi (opCode, OUT byteValue);
 }

EFUSE_ReadBit (byteOffset, bitOffset, OUT bitValue) {
   // Reads fuse from target device
   EFUSE_ReadByte (byteOffset, OUT byteValue);
   bitValue = (byteValue >> bitOffset) & 0x01;
 }
```

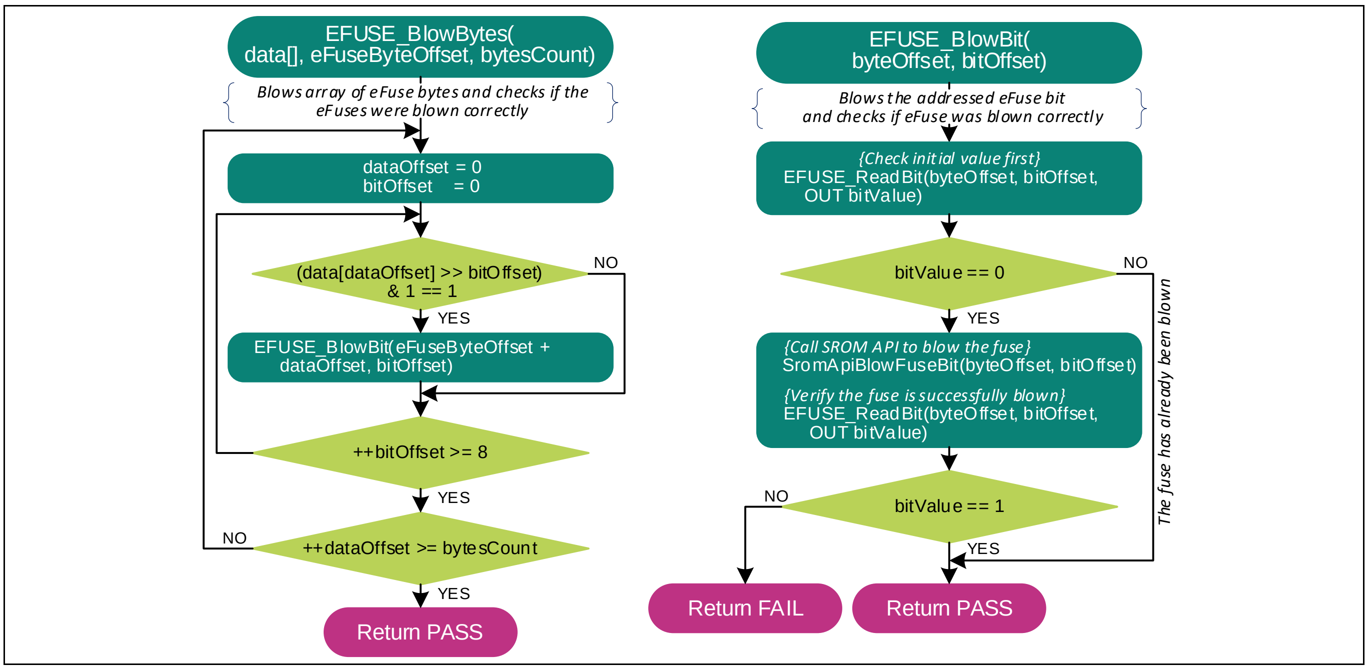

EFUSE_BlowBytes(data[], eFuseByteOffset, bytesCount)
Blows array of eFuse bytes and checks if the eFuses were blown correctly
dataOffset = 0
bitOffset = 0
(data[dataOffset] >> bitOffset) & 1 == 1
NO
YES
EFUSE_BlowBit(eFuseByteOffset + dataOffset, bitOffset)
++bitOffset >= 8
YES
NO
++dataOffset >= bytesCount
YES
Return PASS
EFUSE_BlowBit(byteOffset, bitOffset)
Blows the addressed eFuse bit and checks if eFuse was blown correctly
{Check initial value first}
EFUSE_ReadBit(byteOffset, bitOffset, OUT bitValue)
bitValue == 0
NO
YES
{Call SROM API to blow the fuse}
SromApiBlowFuseBit(byteOffset, bitOffset)
{Verify the fuse is successfully blown}
EFUSE_ReadBit(byteOffset, bitOffset, OUT bitValue)
The fuse has already been blown
NO
bitValue == 1
YES
Return FAIL
Return PASS

Figure 21 **eFuse Read/Write subroutines**

5.14 Step 11 – Verify eFuse (optional)

This step compares the eFuse data in input binary file with the eFuse values in target device and returns FAIL status in case of mismatch. It is optional to perform eFuse verification step just after programming – eFuse programming step includes all required validation. Verification step may be useful before programming if initial eFuse state in target device is unknown.

Note that eFuse memory is not accessible in the Secure or Dead Protection states. This means that the Verify eFuse step can be performed only when the target device is in NORMAL or VIRGIN state.

Code Listing 11 Pseudocode – Step 11. Verify eFuse

```
// This is an entry point of the "Verify eFuse" step.

// Read eFuse data from binary file. Function must be implemented and return 0x400
// bytes in address range 0x90700000 to 0x907003FF.
// Missed data must be filled with 0xFF (IGNORE).
file[] = FILE_ReadData(0x90700000, 0x400);

for (int offset = 0; offset < sizeof(file); offset ++) {
  // Skip fuses defined in file as 0xFF (IGNORE)
  if (file[offset] == 0xFF) continue;

  // Convert virtual address in file to fuse byte and bit offsets in memory and
  // read associated fuse from target device
  EFUSE_FileAddrToFuseOffset (offset, OUT fuseByteOffset, OUT fuseBitOffset);
  EFUSE_ReadBit (fuseByteOffset, fuseBitOffset, OUT bitValue);

  if (file[offset] != bitValue) return FAIL;
}
return PASS;
```

6 Appendix A: Intel hex file format

Intel hex file records are a text representation of hexadecimal-coded binary data. Only ASCII characters are used, so the format is portable across most computer platforms. Each line (record) of Intel hex file consists of six parts, as shown in Figure 22.

Start Code (Colon Character)	Byte Count (1 byte)	Address (2 bytes)	Record Type (1 byte)	Data (N bytes)	Checksum (1 byte)

Figure 22 Hex file record structure

Start code, one character - an ASCII colon (:)

- **Byte count**, two hex digits (1 byte) - specifies the number of bytes in the data field.
- **Address**, four hex digits (2 bytes) - a 16-bit address of the beginning of the memory position for the data.
- **Record type**, two hex digits (00 to 05) - defines the type of the data field. The record types used in the hex file generated by Infineon are as follows.
- 00 - Data record, which contains data and 16-bit address.
- 01 - End of file record, which is a file termination record and has no data. This must be the last line of the file; only one is allowed for every file.
- 04 - Extended linear address record, which allows full 32-bit addressing. The address field is 0000, the byte count is 02. The two data bytes represent the upper 16 bits of the 32-bit address, when combined with the lower 16-bit address of the 00-type record.
- **Data**, a sequence of ‘n’ bytes of the data, represented by 2n hex digits.
- **Checksum**, two hex digits (1 byte), which is the least significant byte of the two's complement of the sum of the values of all fields except fields 1 and 6 (start code ‘:’ byte and two hex digits of the checksum).

Examples for the different record types used in the hex file generated for the PSOC™ 6 MCU are as follows.

Consider that these three records are placed in consecutive lines of the hex file (chip-level protection and end of hex file).

```
:0200000490600A
:0100000002FD
:00000001ff
```

For the sake of readability, “record type” is highlighted in red and the 32-bit address of the chip-level protection is in blue.

The first record (:0200000490600A) is an extended linear address record as indicated by the value in the Record Type field (04). The address field is 0000, the byte count is 02. This means that there are two data bytes in this record. These data bytes (0x9060) specify the upper 16 bits of the 32-bit address of data bytes. In this case, all the data records that follow this record are assumed to have their upper 16-bit address as 0x9060 (in other words, the base address is 0x90600000). 0A is the checksum byte for this record:

```
0x0A = 0x100 – (0x02+0x00+0x00+0x04+0x90+0x60).
```

The next record (:0100000002FD) is a data record, as indicated by the value in the Record Type field (00). The byte count is 01, meaning there is only one data byte in this record (02). The 32-bit starting address for these data bytes is at address 0x90600000. The upper 16-bit address (0x9060) is derived from the extended linear address record in the first line; the lower 16-bit address is specified in the address field of this record as 0000. FD is the checksum byte for this record.

The last record (:00000001FF) is the end-of-file record, as indicated by the value in the Record Type field (01). This is the last record of the hex file.

Note: *The data records of the following multibyte region in the hex file are in big-endian format (MSB in lower address): checksum data at address 0x9030 0000, metadata at address 0x9050 0000. The data records of the rest of the multibyte regions in the hex file are all in little-endian format (LSB in lower address).*

7 Appendix B: eFuse data mapping in file

eFuse bits are stored in the data file using the address range 0x90700000 to 907003FF. This range is virtual and cannot be used for direct AHB Read or Write operations. Refer to Table 7 and Table 8 for eFuse data mapping in file.

Table 7 eFuse data mapping in binary file

eFuse byte address in memory			**eFuse byte address in file**		**PSOC6A-BLE2, PSOC6A-2M rev. < A1**	**PSOC6A-256K, PSOC6A-512K, PSOC6A-2M rev. >= A1**	
Offset	**Macro #**	**Byte in macro**	**[bit 0]**	**[bit 7]**			
0x000	0	0	0x90700000	0x90700007			
…	…	…	…	…	System or Reserved	System or Reserved	
0x013	0	19	0x90700098	0x9070009F			
0x014	0	20	0x907000A0	0x907000A7	SECURE_HASH0	SECURE_HASH0	
…	…	…	…	…	…	…	
0x023	1	3	0x90700118	0x9070011F	SECURE_HASH15	SECURE_HASH15	
0x024	1	4	0x90700120	0x90700127	System or Reserved	System or Reserved	
0x025	1	5	0x90700128	0x9070012F			
0x026	1	6	0x90700130	0x90700137	SECURE_HASH_ZEROS	SECURE_HASH_ZEROS	
0x027	1	7	0x90700138	0x9070013F	DEAD_ACCESS_RESTRICT0	DEAD_ACCESS_RESTRICT0	
0x028	1	8	0x90700140	0x90700147	DEAD_ACCESS_RESTRICT1	DEAD_ACCESS_RESTRICT1	
0x029	1	9	0x90700148	0x9070014F	SECURE_ACCESS_RESTRICT0	SECURE_ACCESS_RESTRICT0	
0x02A	1	10	0x90700150	0x90700157	SECURE_ACCESS_RESTRICT1	SECURE_ACCESS_RESTRICT1	
0x02B	1	11	0x90700158	0x9070015F	LIFECYCLE_STAGE	LIFECYCLE_STAGE	
0x02C	1	12	0x90700160	0x90700167	FACTORY_HASH0	FACTORY_HASH0	
…	…	…	…	…	…	…	
0x037	1	23	0x907001B8	0x907001BF	FACTORY_HASH11	FACTORY_HASH11	
0x038	1	24	0x907001C0	0x907001C7	FACTORY_HASH12	FACTORY_HASH12	ENABLE_FLASH_BOOT_CHECK_IN_NORMAL
0x039	1	25	0x907001C8	0x907001CF	FACTORY_HASH13	FACTORY_HASH13	
0x03A	1	26	0x907001D0	0x907001D7	FACTORY_HASH14	FACTORY_HASH14	
0x03B	1	27	0x907001D8	0x907001DF	FACTORY_HASH15	FACTORY_HASH15	
0x03C	1	28	0x907001E0	0x907001E7	FACTORY_HASH_ZEROS	FACTORY_HASH_ZEROS	
0x03D	1	29	0x907001E8	0x907001EF			
…	…	…	…	…	System or Reserved	System or Reserved	
0x03F	1	31	0x907001F8	0x907001FF			
0x040	2	0	0x90700200	0x90700207		CY_ASSET_HASH0	
…	…	…	…	…	CUSTOMER_DATA	…	
0x04F	2	15	0x90700278	0x9070027F		CY_ASSET_HASH15	

eFuse byte address in memory			eFuse byte address in file		PSOC6A-BLE2, PSOC6A-2M rev. < A1	PSOC6A-256K, PSOC6A-512K, PSOC6A-2M rev. >= A1
Offset	Macro #	Byte in macro	[bit 0]	[bit 7]		
0x050	2	16	0x90700280	0x90700287	🟪	CY_ASSET_HASH_ZEROS
0x051	2	17	0x90700288	0x9070028F	🟪	CUSTOMER_DATA
…	…	…	…	…	🟪	CUSTOMER_DATA
0x07F	3	31	0x907003F8	0x907003FF	🟪	CUSTOMER_DATA

▨	- Used by the system or reserved. Put “ignore” (0xFF) to the data file or skip this range.
🟨	- User-programmable security data (blown during lifecycle transition from NORMAL).
🟩	- Security data blown at the factory. Cannot be programmed by the customer. Validated during the lifecycle transition from NORMAL. Put “ignore” (0xFF) to the data file or skip this range.
🟪	- Customer data

where,

eFuse Area name	Purpose
SECURE_HASH	128-bit (16 bytes) HASH used by boot code to authenticate objects in the Table of Contents, Part 2 (TOC2).
SECURE_HASH_ZEROS	The number of bits that are “0” (fuses that are not blown) in the SECURE_HASH above. This guarantees that once a HASH is programmed, it cannot be changed into another valid HASH value.
DEAD_ACCESS_RESTRICT	Chip access restrictions applied at the DEAD lifecycle stage.
SECURE_ACCESS_RESTRICT	Chip access restrictions applied at the SECURE lifecycle stage.
LIFECYCLE_STAGE	Silicon lifecycle stage. **Note** Only NORMAL to SECURE or NORMAL to SECURE_WITH_DEBUG lifecycle transitions are supported by Infineon tools. If you were supplied with the silicon at a lifecycle stage other than NORMAL, please contact Infineon.
FACTORY_HASH	128-bit (16 bytes) HASH (VIRGIN Factory objects).
FACTORY_HASH_ZEROS	The number of bits that are “0” (fuses that are not blown) in the FACTORY_HASH above.
ENABLE_FLASH_BOOT_CHECK_IN_NORMAL	The field is stored in the least significant bits of last four FACTORY_HASH bytes for PSOC6A-256K, PSOC6A-512K, and PSOC6A-2M (revision id >= A1) devices. If all four bits are blown, boot code validates CY_ASSET_HASH and FACTORY_HASH in NORMAL lifecycle stage. All four bits must be blown before lifecycle transition to SECURE. All the four bits must be in same state (blown or not blown), otherwise the device will go to DEAD state.
CY_ASSET_HASH	128-bit (16 bytes) HASH of Infineon objects such as Flash Boot, TOC1, RTOC1, etc. Introduced for PSOC6A-256K, PSOC6A-512K and PSOC6A-2M (revision id >= A1) devices.

eFuse Area name	Purpose
CY_ASSET_HASH_ZEROS	The number of bits that are '0' (fuses that are not blown) in the CY_ASSET_HASH above
CUSTOMER_DATA	Can be used by the customer for application or security purposes

Table 8 Security fuses bit mapping in the data file

eFuse byte offset in memory, area name	eFuse bit[#]	Associated byte in file	Bit field name
0x27 DEAD_ACCESS_RESTRICT0	0	0x90700138	CM0_DISABLE
	1	0x90700139	CM4_DISABLE
	2	0x9070013A	SYS_DISABLE
	3	0x9070013B	SYS_AP_MPU_ENABLE
	4	0x9070013C	SFLASH_ALLOWED
	5	0x9070013D	
	6	0x9070013E	MMIO_ALLOWED
	7	0x9070013F	
0x28 DEAD_ACCESS_RESTRICT1	0	0x90700140	FLASH_ALLOWED
	1	0x90700141	
	2	0x90700142	
	3	0x90700143	SRAM_ALLOWED
	4	0x90700144	
	5	0x90700145	
	6	0x90700146	UNUSED. Put "ignore" (0xFF) to the data file.
	7	0x90700147	DIRECT_EXECUTE_DISABLE
0x029 SECURE_ACCESS_RESTRICT0	0	0x90700148	CM0_DISABLE
	1	0x90700149	CM4_DISABLE
	2	0x9070014A	SYS_DISABLE
	3	0x9070014B	SYS_AP_MPU_ENABLE
	4	0x9070014C	SFLASH_ALLOWED
	5	0x9070014D	
	6	0x9070014E	MMIO_ALLOWED
	7	0x9070014F	
0x02A SECURE_ACCESS_RESTRICT1	0	0x90700150	FLASH_ALLOWED
	1	0x90700151	
	2	0x90700152	
	3	0x90700153	SRAM_ALLOWED
	4	0x90700154	
	5	0x90700155	
	6	0x90700156	SMIF_XIP_ALLOWED
	7	0x90700157	DIRECT_EXECUTE_DISABLE
	0	0x90700158	NORMAL

eFuse byte offset in memory, area name	eFuse bit[#]	Associated byte in file	Bit field name
0x2B LIFECYCLE_STAGE	1	0x90700159	SECURE_WITH_DEBUG
	2	0x9070015A	SECURE
	3	0x9070015B	Infineon use only. Put “ignore” (0xFF) to the data file.
	4	0x9070015C	
	5	0x9070015D	
	6	0x9070015E	
	7	0x9070015F	

where,

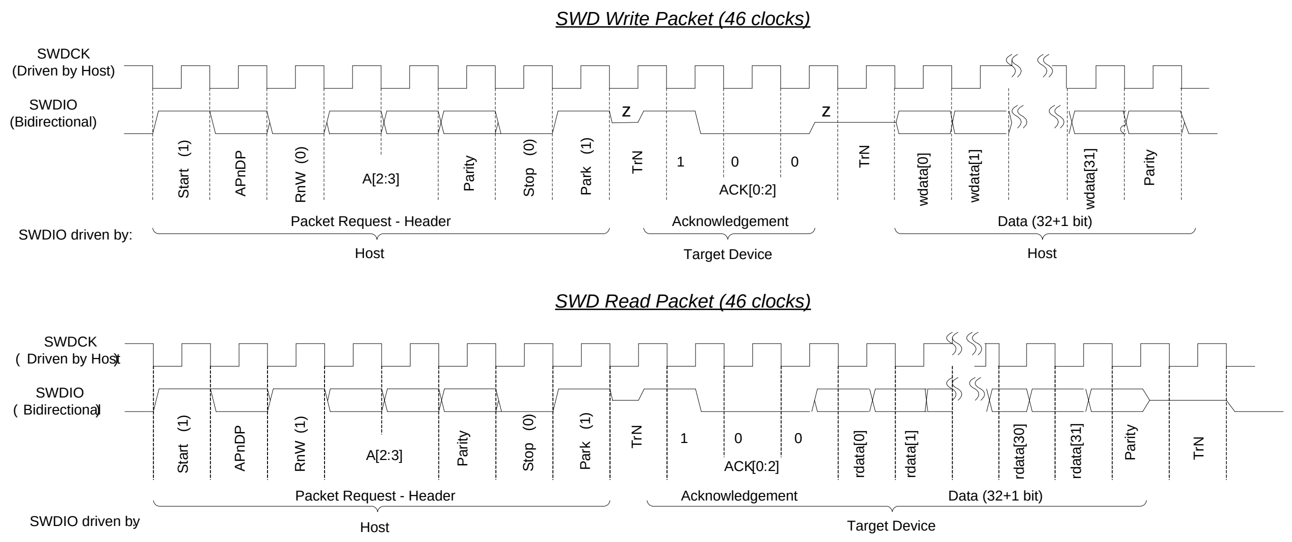
Bit field name	Purpose
CM0_DISABLE	Disable the debug access to CM0+ CPU
CM4_DISABLE	Disable the debug access to CM4
SYS_DISABLE	Disable the debug access to the System access port
SYS_AP_MPU_ENABLE	Enable the system access port MPU
SFLASH_ALLOWED	Allow the SYS AP MPU protection of SFlash. Only a portion of SFlash starting at the bottom of the area is exposed: “0” - entire region; “1” - 1/2; “2” - 1/4th; “3” – nothing
MMIO_ALLOWED	Allow the SYS AP MPU protection of MMIO: “0” - All MMIO registers are accessible; ‘1’ – Only IPC MMIO registers accessible (system calls); “2”, “3” – No MMIO access.
FLASH_ALLOWED	Allow the SYS AP MPU protection of Flash. Only a portion of application flash starting at the bottom of the area is exposed: “0” – the entire region; “1” – 7/8th; “2” – 3/4th; “3” – 1/2; “4” – 1/4th; “5” – 1/8th; “6” – 1/16th; “7” – nothing
SRAM_ALLOWED	Allow the SYS AP MPU protection of SRAM. Only a portion of SRAM starting at the bottom of the area is exposed. Encoding is the same as FLASH_ALLOWED.
SMIF_XIP_ALLOWED	Allow the SYS AP MPU protection of SMIF XIP: “0” – the entire region is accessible; “1” – nothing
DIRECT_EXECUTE_DISABLE	Disable the "direct execute" system call
NORMAL	The lifecycle stage of a device after trimming and testing is complete at the factory
SECURE_WITH_DEBUG	The same as the SECURE lifecycle stage, except the device allows for debugging. Prior to transitioning to this stage, the SECURE_HASH must be programmed in eFuse and valid application code must be programmed in the application flash
SECURE	The lifecycle stage of a secure device. Prior to transitioning to this stage, the SECURE_HASH must be programmed in eFuse and valid application code must be programmed in the application flash

8 Appendix C: Serial wire debug (SWD) Protocol

The SWD protocol is a packet-based serial transaction protocol. At the pin level, it uses a single bidirectional data connection (SWDIO) and a clock connection (SWDCK). The host programmer always drives the clock line, while either the programmer or the PSOC™ 6 MCU drives the data line. A complete data transfer (one SWD packet) requires 46 clocks and consists of three phases:

- **Packet Request** – The host programmer issues a request to the PSOC™ 6 MCU (silicon).
- **Acknowledge Response** – The PSOC™ 6 MCU (silicon) sends an acknowledgement to the host.
- **Data Transfer Phase** – The data transfer is either from the PSOC™ 6 MCU to the host, following a read request (RDATA), or from the host to the PSOC™ 6 MCU, following a write request (WDATA). This phase is present only when a packet request phase is followed by a valid (OK) acknowledge response.

Figure 23 shows the timing diagrams of the read and write SWD packets.



a) Host Write Cycle – host sends data on the SWDIO line on falling edge of SWDCK and target will read that data on next SWDCK rising edge (for example, 8-bit header data).

b) Host Read Cycle – target sends data on SWDIO line on rising edge of SWDCK and the Host should read that data on next SWDCK falling edge (for example, ACK phase (ACK[2:0]), Read Data (rdata[31:0])).

c) The Host should not drive the SWDIO line during TrN phase. During first TrN phase (½ cycle duration) of SWD packet, target starts driving the ACK data on the SWDIO line on the rising edge of SWDCK. The host should read the data on the subsequent falling edge of SWDCK. The second TrN phase is 1.5 clock cycles as shown in figure above. Both target and host will not drive the line during the entire second TrN phase (indicated as 'z'). Host should start sending the Write data (wdata) on next falling edge of SWDCK after second TrN phase.

Figure 23 Write and Read SWD packet timing diagrams

The SWD packet is transmitted in this sequence:

- The start bit initiates a transfer; it is always logical 1
- The APnDP bit determines whether the transfer is an AP access (indicated by '1'), or a DP access (indicated by '0')
- The next bit is RnW, which is '1' for read from the MCU or '0' for a write to the MCU

The ADDR bits (A[3:2]) are register select bits for the access port or debug port. See Table 3 for register definition.

- The parity bit contains the parity of APnDP, RnW, and ADDR bits. This is an even parity bit. If the number of logical 1s in these bits are odd, then the parity must be '1', otherwise it is '0'.
- If the parity bit is not correct, the PSOC™ 6 MCU ignores the header, and there is no ACK response. From the host standpoint, the programming operation should be aborted and retried by doing a device reset.
- The stop bit is always logic 0
- The park bit is always logic 1 and should be driven high by the host

The ACK bits are device-to-host response. Possible values are shown in Table 9. Note that ACK in the current SWD transfer reflects the status of the previous transfer. OK ACK means that the previous packet was successful. WAIT response requires a data phase, as explained in the following list. For a FAULT status, the programming operation should be aborted immediately.

- For a WAIT response, if the transaction is a read, the host should ignore the data read in the data phase. The PSOC™ 6 MCU does not drive the line, and the host must not check the parity bit as well.
- For a WAIT response, if the transaction is a write, the data phase is ignored by the PSOC™ 6 MCU. However, the host must still send the data to be written from the standpoint of implementation. The parity data parity bit corresponding to the data should also be sent by the host.
- For a WAIT response, it means that the PSOC™ 6 MCU is processing the previous transaction. The host can try for a maximum four continuous WAIT responses to see if an OK response is received. If it fails, then the programming operation should be aborted and retried.
- For a FAULT response, the programming operation should be aborted and retried by doing a device reset.

Table 9 ACK response for SWD transfers

ACK[2:0]	SWD
OK	001
WAIT	010
FAULT	100
NACK	111

- The data phase includes a parity bit (even parity)
- For a read packet, if the host detects a parity error, then it must abort the programming operation and try again
- For a write packet, if the PSOC™ 6 MCU detects a parity error in the data sent by the host, it generates a FAULT ACK response in the next packet

Turnaround (TrN) phase: There is a single-cycle turn-around phase between the packet request and the ACK phases, as well as between the ACK and data phases for write transfers as shown in Figure 23. According to the SWD protocol, both the host and the PSOC™ 6 MCU use the TrN phase to change the drive modes on their respective SWDIO lines. During the first TrN phase after packet request, the PSOC™ 6 MCU starts driving the ACK data on the SWDIO line on the rising edge of SWDCK in the TrN phase. This ensures that the host can read the ACK data on the next falling edge. Thus, the first TrN cycle lasts for only a half-cycle duration. The second TrN cycle of the SWD packet is one and one-half cycle long. Neither the host nor the PSOC™ 6 MCU should drive the SWDIO line during the TrN phase, as indicated by 'z' in in Figure 23.

- The address, ACK, and read and write data are always transmitted LSB first.
- According to the SWD protocol, the host can generate any number of SWD clock cycles between two packets with the SWDIO low. You should generate several dummy clock cycles (three) between two packets or make the clock free running in IDLE mode.

Note: The SWD interface can be reset by clocking 50 or more cycles with the SWDIO kept high. To return to the idle state, SWDIO must be clocked low once.

9 Appendix D: Joint test action group (JTAG) protocol

The PSOC™ 6 JTAG interface complies with the IEEE 1149.1 specification and provides additional instructions. There are two TAPs in the silicon. One is in the IOSS for boundary scan and the other is in the CPUSS DAP (IDCODE 0x6BA00477), which is used for device debug and programming. The two TAPs are connected in series, where the TDO of the IOSS TAP is connected to the TDI of the DAP TAP. This is illustrated in Figure 24.

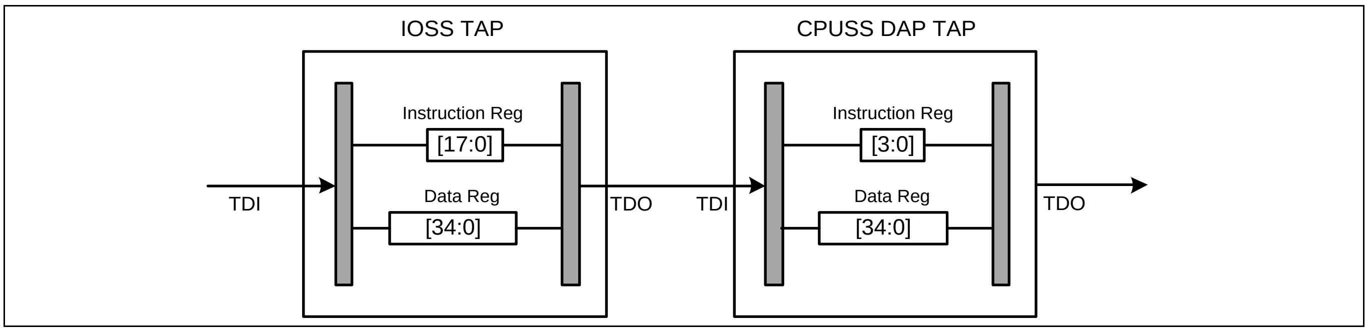


Figure 24 IOSS/DAP TAP connection

Each TAP consists of a 35-bit data register (called DP/AP access register). The size of the instruction register is 4-bits for DAP TAP and 18-bits for IOSS TAP. The important instructions to program the device through JTAG are listed in Table 10.

Table 10 PSOC™6 JTAG instructions

Bit Code [3:0]	Instruction	PSOC™ 6 function
1110	IDCODE	Connects TDI and TDO to the device 32-bit JTAG ID code
1010	DPACC	Connects TDI and TDO to the DP/AP access register (35-bit), for access to the Debug Port registers
1011	APACC	Connects TDI and TDO to the DP/AP access register (35-bit), for access to the Access Port registers
1111	BYPASS	Bypasses the device, by providing 1-bit latch (bypass register) connected between TDI and TDO

Table 10 also lists which instructions are applicable for each TAP. If an instruction that is not applicable is shifted into a TAP, the TAP goes into bypass mode. In bypass mode, the data register is only 1-bit long with the contents of 0. The bypass mode is used to isolate the PSOC™ 6 MCU TAP. For example, if targeting the IOSS TAP, the DAP TAP is put in bypass mode by shifting in the BYPASS instruction into its instruction register and if targeting the DAP TAP, the IOSS TAP will be placed in bypass mode. See the examples of TAPs configuration in Figure 25.

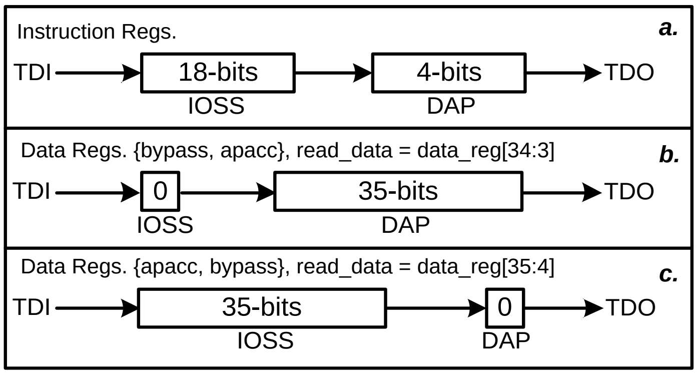

Instruction Regs.
a.
TDI
18-bits
IOSS
4-bits
DAP
TDO
Data Regs. {bypass, apacc}, read_data = data_reg[34:3]
b.
TDI
0
IOSS
35-bits
DAP
TDO
Data Regs. {apacc, bypass}, read_data = data_reg[35:4]
c.
TDI
35-bits
IOSS
0
DAP
TDO

Figure 25 OSS/DAP TAP configuration examples

- Instructions registers combined. 22 bits total
- Access the DAP's APACC registers for device debugging and programming. IOSS TAP in bypass mode. 36 bits total
- Access the IOSS APACC registers to enable test modes. DAP TAP in bypass mode. 36 bits total

References

[1] 002-31353: PSOC™ 64 MCU Programming Specification.

[2] Arm® specification: Arm® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 (ARM IHI 0031C).

Revision history

Document revision	Date	Description of changes
**	2016-08-10	New specification. This is preliminary version of PSOC 6000 BLE2 programming specification, created for PR3 milestone of the product. It contains To Be Defined (TBD) items, which will be filled/resolved in further revisions of this specification.
*A	2017-03-06	Chapter 1: Removed 'TBD' notice. Chapter 2: Added eFuse memory to Nonvolatile Subsystem Section 3.1: Organization of the Hex File. Added eFuse description Sections 3.1and 5.5: Added more Silicon Id information (CDT 235611, CDT 227304) Section 4.1Reworked Communication Interface and daughter sections: Added Program and Debug Interface description Added description of AP resource access possibilities, DAP security and DAP power domain (taken from TRM: 002-18176 Rev. **). Reworked "Top-Level Silicon Architecture" diagram Added SWD<->JTAG switching description and diagrams Section 4.34.2.5: Reworked Physical layer connection diagram, pin and acquisition modes description. Section 4.2.5: Added note that the C style pseudo code is used in document Section 5.2: Modified "Ipc_Acquire" subroutine: added write to IPC.ACQUIRE to be compatible with CFR bit file (per CDT 251547) Added missed constants (defines) for IPC_xxx subroutines Modified "ProgramRow" API OpCode to use blocking mode (CDT 260497) Added "WriteRow" OpCode for WFLASH/SFLASH operations Modified "Ipc_PollLockStatus" and 'CallSromApi' subroutines to poll for correct status. (CDT 259104) Modified "CallSromApi" subroutine to use correct IPC for DAP. Added comments for flash loader use cases. Updated interrupt notification steps. Modified parameters and usage of "CallSromApi" "PollSromApiStatus" subroutines to return the status/result word (required for Silicon Id SROM API). Modified "Ipc_Acquire" subroutine to poll for lock status Fixed some issues with capitalization of variables names in pseudo-code Section 0: Correct AP selection code and block scheme (synced with PP implementation) Reworked timing diagram and description Section 5.6: Removed TBD, slightly modified description

Document revision	Date	Description of changes
		Section 5.7: Changed title to "Step 4 – Verify **Blank** Checksum" to highlight difference with step #9 (checksum verification after program) Section Figure 16: Program Protection Settings Appendix C: Added Joint Test Action Group (JTAG) Protocol information Overall: Added WFLASH and extended SFLASH description and diagram Removed appendixes about electrical/timing specifications: one source of truth – per email discussion and, as result, comment #12 in CDT 239146) Renamed (title, footer and across document) from "PSOC 6000 BLE2" to "PSOC® 6" to be common for whole family More comments and format corrections for pseudo-code Formatting and corrections per CDT 233048 and CDT 239146 Updated copyrights Updated logo and added Confidential note in the footer Section 5.14: Added section Verify Protection Settings
*B	2017-08-21	- Title changed to - "PSOC 6 MCU PROGRAMMING SPECIFICATIONS". - Removed registered trade-mark across document. Section 1. Rephrased target audience. Section 1.1 - Removed useless sentence about programmer environment - Figure 1 1: Removed PSOC Creator version, added 3rd party debuggers Section 2. Figure 2 1: Updated Nonvolatile Subsystem map: - Added public SFlash regions in *A silicon - Added XIP region - Minor correction in eFuse region description - Added end address for each used row Section 2.2: Removed statement about optional programming of WFLASH region Section 2.3: - Added *A silicon public SFlash regions and description - Added notes about SFlash regions accessibility - Removed statement about optional programming of SFLASH region Section 2.4: Added more description for eFuse memory usage, layout and limitations Section 2.5: Added new section for XIP address space description (SMIF) Section 3.1: - Combined sections 3.1 and 3.2 - Added SFlash, WFlash and XIP to the list of regions in the hex file - Removed device features description, which is not relevant to programming - Reworked Figure 3 1 with WFLASH, SFLASH and XIP

Document revision	Date	Description of changes
		- Corrected Main Flash description - Added description of WFLASH, SFLASH and XIP regions - Corrected hex file version for target family Section 4.1.1: - Removed debug features description (not relevant with programming) - Removed excessive sentence about JTAG availability - Added a not about priority of DAP secure bits - Reworked Top-Level Architecture figure - Simplified DAP security section, added reference to TRM for protection settings Section 4.1.2: Removed 'parity' for JTAG command Section 5.3: Figure 5 3: - Increased timeout from 2ms up to 1000ms (for secure applications). The same is increased in pseudo-code in sections 5.3 and 5.4 - Corrected ACK value for SWD - Corrected SWD/JTAG IDCODE verification Section 5.2: Added CheckFactoryCMAC SROM API code. Section 5.3: - Updated timings for SECURE application - Updated statement about M0+ possible lockup state - Renamed "Tsrss_up" to "Tlite_up" per SAS - Added a note about check for CPUSS.SWD_CONNECTED before repurposing pins - Added note about configurable TListen parameter - Minor corrections Section 5.4: - Added Figure 5-4 with the Acquire Chip (Alternate Method) procedure. - Increased timeout from 100ms up to 600ms (for secure applications) - Removed excessive ID check after reset - Added address verification at reset vector for the case of empty flash or secure application (CDT 281851, CDT 282643) Section 5.5: Added Life Cycle stage to SiliconID SROM API. Section 5.12: Corrected SFLASH region description for *A silicon Section 5.13: Updated Program Protection Settings section per VENN-39*C. Fixed footer for JTAG appendix Initial public release
*C	2017-11-27	Section 2, Figure 2-1: - Renamed CMAC to HASH; - Added customer data eFuse region and extended eFuse area Section 2.4: - Extended description of eFuse memory organization

Document revision	Date	Description of changes
		- Added customer data eFuse region - Minor text corrections Section 3.1: - Corrected eFuse size in HEX file – up to 4096 bytes; - Added custom data eFuse area; - Corrected size of user data in SFLASH memory Section 4.1.1: Removed excessive and uncomplete sentence about SYS-AP. Section 4.1.2: - Added SWDtoJTAG to Table 4-1 - Added DP.RDBUFF register to Table 4-2 Section 4.1.3: Minor correction in pseudo-code Section 4.2: Renamed debug pins in Figure 4-5 to not use subscript Section 5.2: - Replaced DRW with RDBUFF register in ReadIO subroutine; - Added DAP_Handshake, DAP_Init and DAP_ScanAP subroutines with pseudo code and flow charts - Added missing bracket and minor formatting correction in pseudo-code Section 5.3: - Changed font for time periods - Reworked pseudo-code and flow chart to use DAP_Init and DAP_ScanAP subroutines - Removed verification for lockup state from pseudo-code and Figure 5-3 Section 5.4 - Reworked pseudo-code and flow chart to use DAP_Init and DAP_ScanAP subroutines - Added CM0_VECTOR_TABLE_BASE register verification instead of flash value - Added polling for halt status (CDT 287776) - Fixed using logical instead of bitwise operation checking DHCSR register Section 5.13 - Renamed CMAC to HASH in flow chart Overall: - Replaced all instances of CMAC or cmac with HASH or hash - Minor corrections in pseudo-code across document - Changed line width in flow charts
*D	2018-01-29	Section 5.2: - Added ROM and SFLASH definitions used in Test Mode verification (CDT 298216) - Removed DHCSR register definitions as not used in document

Document revision	Date	Description of changes
		- Added EraseSector SROM API code used in section 5.3 Section 5.3: - Pseudo code and flow diagram updated with PC verification (CDT 298216) - Do AP scanning before TEST_MODE read (sync with PP implementation) Section 5.6: - Use EraseSector SROM API for SECURE and DEAD state (CDT 275601) Section 5.13: - Added check in Figure 5-12 that Dead Access Restrictions match hex file - Updated pseudo-code due to changes done per CDT 296534 Overall: - Comments in flow chart placed before operation, not after
*E	2018-07-13	Section 5.2: - Added constants definition for CY8C6xx8, CY8C6xxA devices
*F	2018-12-14	Section 5.13: - Updated Figure 5-12 to include Customer Data eFuse programming steps - Updated pseudo-code to reflect changes made in Figure 5-12
*G	2019-02-13	- Optimized Figure 5-1 High-level programming flow in Section 5.1 - Modified alternative acquisition sequence to use correct address for CM0_VECTOR_TABLE_BASE register in 2M devices in Sections 5.2, 5.4 - Optimized constants definition in Section 5.2 - Corrected issues in pseudo-code for DAP_Init and DAP_Handshake subroutines in Section 5.2.2 - Added FACTORY_HASH_ZEROS validation for BLE devices in Section 5.13 - Added Appendix B: eFuse Data Mapping in Hex - Format corrected and simplified names for SROM API constants in pseudo-code - Optimized pseudo-code for smaller size and better readability - Corrected headers and numbering across the document
*H	2019-04-16	- Section 5.2.1: Added PSOC6A-512K definition - Section 5.2.2: Updated IPC structure handling for system calls to backup/restore interrupt mask - Section 5.4: Updated error mask for VTBASE to reflect changes in latest boot code - Section 5.11: Fixed typo in pseudo-code for WFLASH verification step
*I	2019-06-04	- Terminology updates throughout using terms “application flash, auxiliary flash, supervisory flash, and eFuse. - Reworked eFuse programming and verification steps. Included specific steps for PSOC2A-2M devices.

Document revision	Date	Description of changes
		- Reworked AUXflash and Sflash programming and verification steps. Included NAR, Public Key and TOC2 sub-regions programming/verification. Used common subroutines for AUXflash and Sflash regions. - Format and style corrections in pseudocode across the document. - Corrections and clarifications across the document.
*J	2019-07-30	Updated eFuse memory mapping and programming flow for PSOC6A-512K and PSOC6A-2M (starting at revision A1 (*A) devices: - Updated Figure 2-1 in Section 2 with differences in Customer Data eFuse bytes offsets - Updated Figure 5-1 in Section 5.1 with Step 2 renamed from “Check Silicon ID” to “Identify Silicon” - Renamed Step 2 in Section 5.5 from “Check Silicon ID” to “Identify Silicon”. Added family identification pseudocode. - Updated flow chart and pseudo code in Section 5.13 to blow the ENABLE_FLASH_BOOT_CHECK_IN_ NORMAL eFuse field before transition to SECURE life-cycle stage. - Updated Table B-1 in Appendix B with Customer Data offset changes and new eFuse fields.
*K	2019-11-23	- Section 5.2.2, Ipc_Acquire() function – described a workaround for SROM issue which leads to IPC acquisition failures - Updated phrasing “SROM API” to “SROM function” wherever appropriate - Fixed usage of apIndex variable in DAP_ScanAP() function
*L	2020-03-12	- Section 5.2.2, Ipc_Acquire() function – updated workaround code to read PPU unconditionally
*M	2020-05-13	Updated with CY8C6xx4 device
*N	2020-10-06	- Section 5.4: Updated error mask for VTBASE
*O	2021-04-27	Added note to scope to exclude PSOC 64 devices Updated to Infineon template
*P	2025-03-03	Added missed IPC_INTR_STRUCT_INTR_MASK_OFFSET definition. Unused definitions deleted. Updated PSoC™ to PSOC™. Added PSOC™ disclaimer.
*Q	2025-07-22	Updated acquisition flow (pseudo code and diagram) in Section 5.3 for single-core devices. Pseudo code formatting changes across the document for better readability (comments, font, spacing; no functional changes). Minor corrections in the title page

Edition 2025-07-22

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-15554 Rev.*Q