PSOC™ 64 MCU

Programming specification

About this document

Scope and purpose

This document provides the information necessary to program the nonvolatile memory of the PSOC™ 64 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™ 64 MCUs.

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 own 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 **3**
1.1 Programmer 3
1.2 PSOC™ 64 MCU family overview 4
2 Nonvolatile memory subsystem **5**
2.1 Application flash 5
2.2 Auxiliary flash (AUXflash) 6
2.3 Execute-in-place (XIP) 6
3 Hex file **7**
3.1 Organization of the hex file 7
4 Protocol stack **8**
4.1 Communication interface 9
4.1.1 Program and debug interface 9
4.1.1.1 DAP security 10
4.1.1.2 DAP power domain 10
4.1.1.3 SWD/JTAG selection 10
4.1.2 Hardware access commands 11
4.1.3 Pseudocode 12
4.2 Physical layer 14
5 Programming algorithm **16**
5.1 High-level programming flow 16
5.2 Constants and subroutines used in the programming flow 17
5.2.1 Constants 17
5.2.2 Subroutines 19
5.3 Step 1 – Acquire PSOC™ 64 MCU 25
5.4 Step 2 – Identify silicon 28
5.5 Step 3 – Erase Application Flash 29
5.6 Step 4 – Program Application Flash 30
5.7 Step 5 – Verify Application Flash 32
5.8 Step 6 – Program and Verify AUXflash 33
6 Appendix A: Intel Hex File Format **37**
7 Appendix B: Serial Wire Debug (SWD) Protocol **39**
Revision history **41**
Disclaimer **42**

1 Introduction

This document provides the information necessary to program the nonvolatile memory of the PSOC™ 64 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. 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™ 64 MCUs.

1.1 Programmer

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

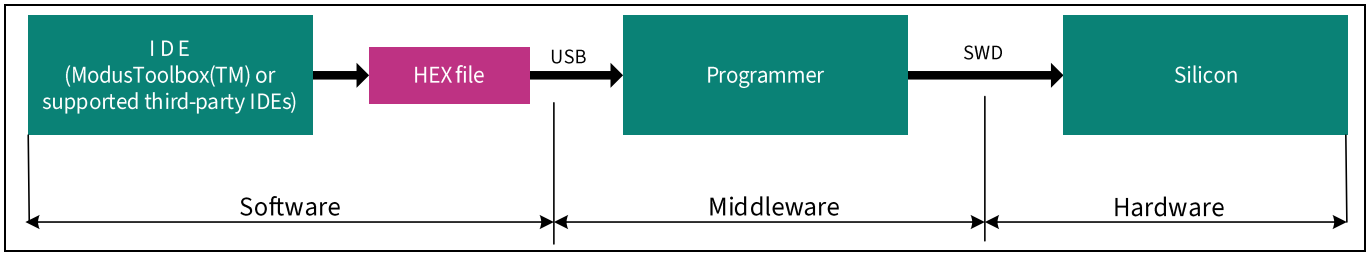


Figure 1 Programmer in development environment

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 the programmer performs three functions:

- Parses the binary file and extracts the necessary information
- Interfaces with the silicon as a serial wire debug (SWD) master
- Implements the programming algorithm by translating the data from a binary file into SWD 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 between the protocol (such as 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, MiniProg4 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™ 64 MCU family overview

The PSOC™ 64 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 protocol.

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

The part can be programmed after it is installed in the system by way of the SWD 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 reference manual.

This document includes the following appendices:

- Appendix A: Intel Hex File Format
- Appendix B: Serial Wire Debug (SWD) Protocol

2 Nonvolatile memory subsystem

This chapter describes the nonvolatile memory subsystem of the PSOC™ 64 MCU silicon.

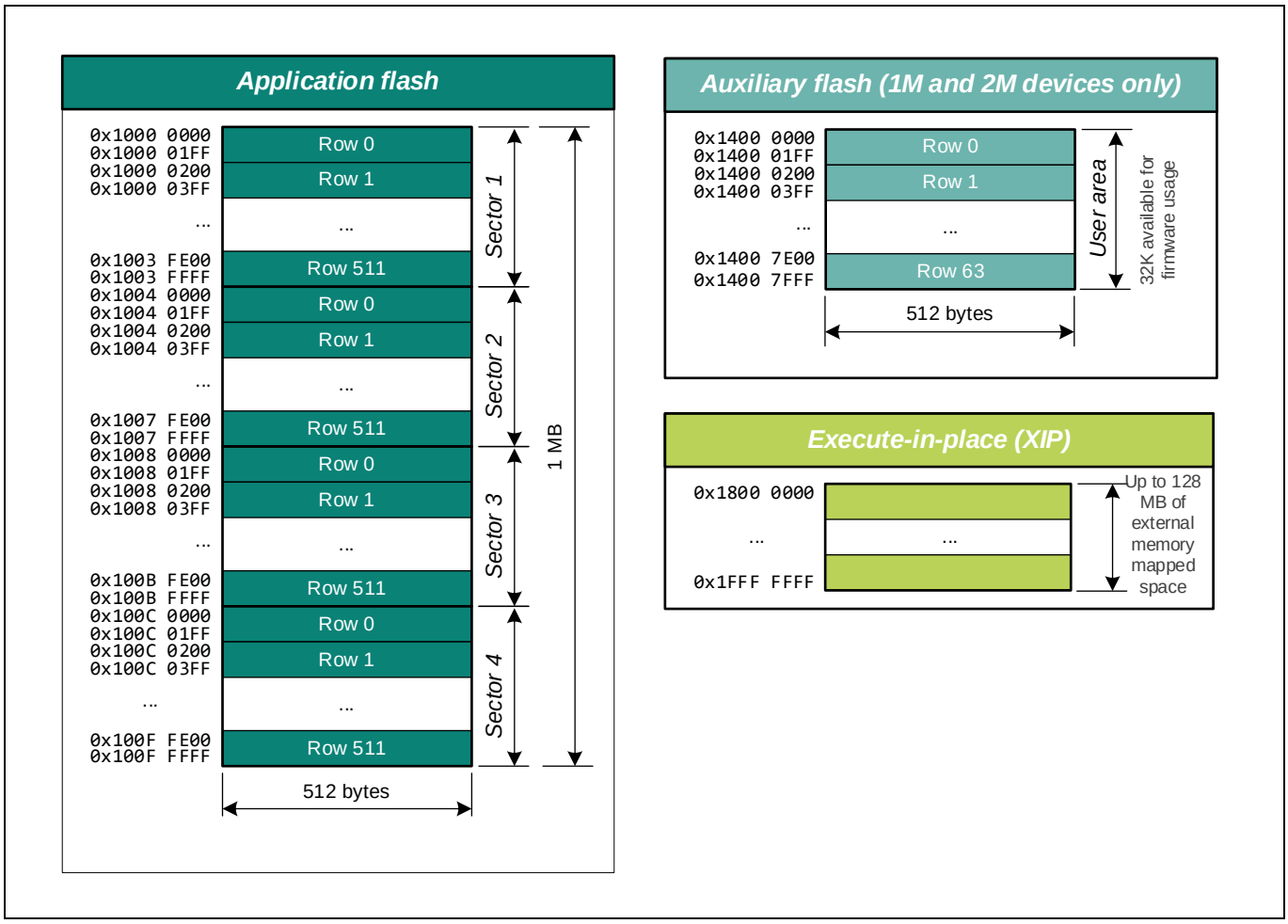


Figure 2 Nonvolatile subsystem

2.1 Application flash

The application flash is organized into sectors. The sector size is 256 KB. There are eight such sectors for parts with 2 MB flash, four sectors for parts with 1 MB flash, and two sectors for 512 KB parts. There are 512 rows in the sector each consisting of 512 bytes.

Some regions of the application flash are reserved for system use and thus are not available for programming. Depending on the total size of the application flash, the following regions are reserved:

- 2 MB parts: 0x101D0000 – 0x101FFFFF
- 1 MB parts: 0x100D0000 – 0x100FFFFF
- 512 KB parts: 0x10060000 – 0x1007FFFF

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

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

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

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

On 512 KB parts, the auxiliary flash is reserved for system use and is therefore not available for programming.

2.3 Execute-in-place (XIP)

Unlike other memory regions, the XIP region is not associated with any physical memory in PSOC™ 64 MCUs. The purpose of the XIP region is to map the address space of the external memory devices, which are connected to PSOC™ 64 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 the CMSIS standard. Such algorithms are loaded into target SRAM by programming software and executed from there for memory bank programming.

This document does not cover external flash programming.

3 Hex file

This chapter describes the information that the programmer must extract from the hex file to program the PSOC™ 64 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™ 64 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™ 64 MCU defines these data sections in the hex file:

- User program (code) for the application flash region
- User data for the AUXflash region
- User data or code for the external flash memory

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

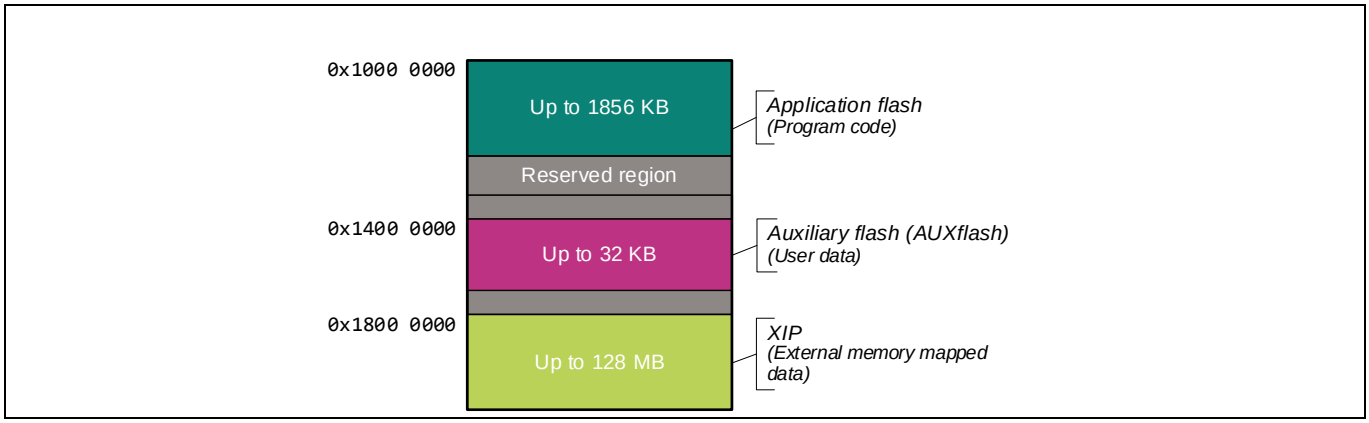


Figure 3 Hex file organization for PSOC™ 64 MCUs

- **0x1000 0000 – application flash**: This is the user code that must be programmed. The maximum size of this section must not exceed the available flash size of the PSOC™ 64 MCU (up to 1856 KB). The address space of this section in the hex file is directly mapped to the physical addresses of the CPU. See 2.1 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 project. See 2.2 Auxiliary flash (AUXflash) for the region description.
- **0x1800 0000 –XIP**: Can store up to 128 MB of the external memory mapped data. Availability of this section in hex file is optional and depends on linker scripts usage in user project. See 2.3 Execute-in-place (XIP) for the region description.

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.

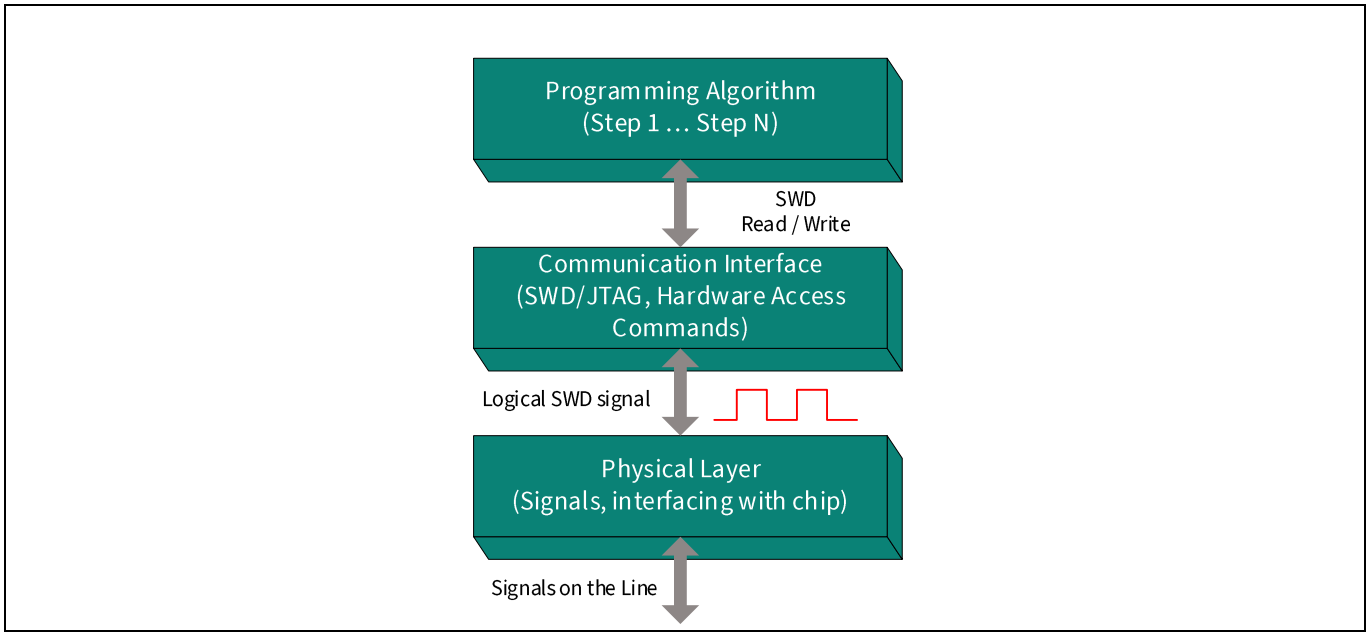


Figure 4 Programmer protocol stack

- **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 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 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. SWD 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 SWD 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 Infineon-supplied programmer and debugger or a third-party device that supports programming and debugging) can access internal resources through the program and debug interface provided in PSOC™ 64 MCU silicon. The serial wire debug (SWD) interface can be used as the communication protocol between the external device and the PSOC™ 64 MCU.

4.1.1 Program and debug interface

The main purpose of PSOC™ 64 MCU program and debug interface is to support programming and debugging through the 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™ 64 MCU silicon implements a debug access port (DAP), which integrates serial wire/JTAG debug port (SWJ-DP) and complies with the Arm® specification “Arm Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 (ARM IHI 0031C)”.

Although PSOC™ 64 MCU implements SWJ-DP interface, only SWD protocol is currently supported.

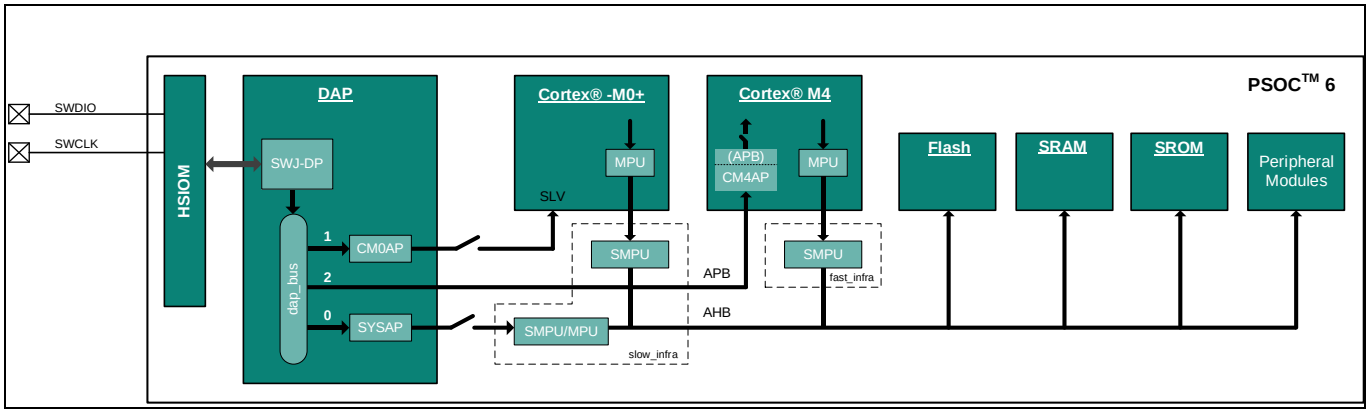


Figure 5 Top-level silicon architecture

The debug physical port pins communicate with the DAP through the Hi-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™ 64 MCU has several AHB masters, including 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.

4.1.1.1 DAP security

For security reasons, all three APs each 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 life-cycle stage. This feature can be used to block debug access during normal operation but re-enable some debug access after a 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 life-cycle stage or based on NAR settings in Supervisory flash for NORMAL life-cycle. 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).

See the "Device Security" and "Protection Units" sections of the reference manual for more details of the security settings for the PSOC™ 64 MCU.

4.1.1.2 DAP power domain

Almost all the debug components are part of the Active power domain. The only exception is the SWD-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 infield 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-DP has a register (DP_CTL_STAT) with two power request bits. The two bits are CDBGPWRUPREQ and CSYSPWRUPREQ, which request for debug power and system power, respectively. These bits must remain set for the duration of the debug session.

4.1.1.3 SWD/JTAG selection

The JTAG and SWD are mutually exclusive because of ARM's 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 power up reset. Because PSOC™ 64 MCU supports only SWD protocol, sending JTAG-to-SWD switching sequence is required to gain access to the system.

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.

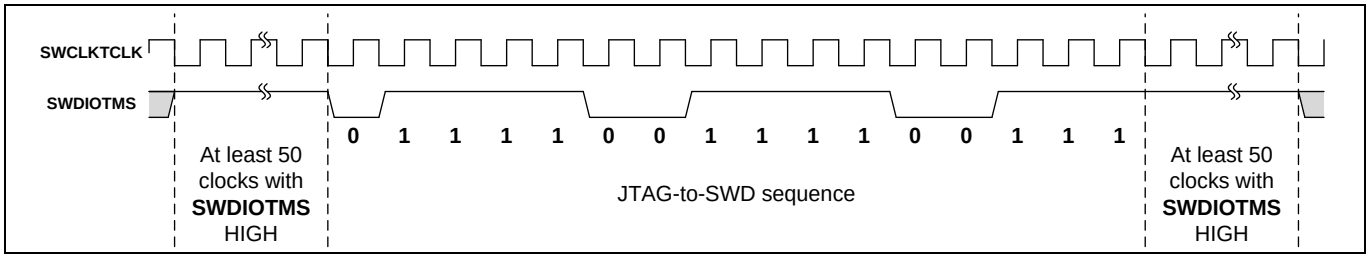


Figure 6 JTAG-to-SWD sequence timing

For a more detailed description, see the “SWD and JTAG select mechanism” section in “Arm Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 (ARM IHI 0031C)”.

4.1.2 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 1 lists the hardware access commands used by the software layer.

Table 1 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 a first transaction in programming flow. See SWD/JTAG selection for implementation details.
SWD_Write	IN APnDP, IN addr, IN data32, OUT ack	Sends a 32-bit data to the specified register of the DAP using 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 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.
ToggleReset	–	Generates the reset signal for PSOC™ 64 MCU. The programmer must have a dedicated pin connected to the XRES pin of the PSOC™ 64 MCU.
Power	IN state	If the programmer powers the PSOC™ 64 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 B: Serial Wire Debug (SWD) 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 2 shows the DAP registers that are used during programming.

Table 2 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 the Arm® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2.

4.1.3 Pseudocode

This document uses 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. Make sure 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 2). 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 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 1) is as follows.

```
//------------ DAP Read/Write subroutines ---------------------------------
SWJ_Status Write_DAP (Register, data32) {
    SWD_Write (Register.APnDP, Register.Addr, data32, OUT ack);
    return ack;
}

SWJ_Status Read_DAP (Register, OUT data32) {
  ack = ACK_FAIL;
  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
  }
  return ack;
}
```

The programming code in Programming is based mostly on the Write and Read pseudo commands and some commands in Table 1.

4.2 Physical layer

This section summarizes the hardware connection between the programmer and the PSOC™ 64 MCU for programming. Figure 7 shows the generic connection between the PSOC™ 64 MCU and the programmer. See Table 3 for pins/signals description.

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

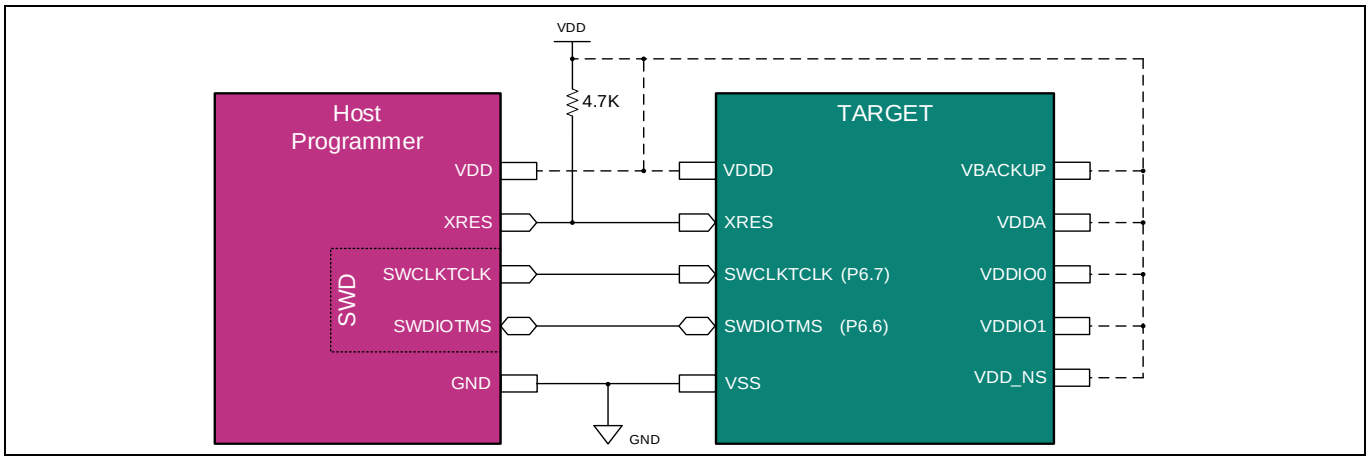


Figure 7 Connection schematic of programmer

Table 3 Pins/signals description

Pin	SWD		Description
	Signal name	Mandatory	
SWCLKTCLK	Serial wire clock (SWCLK)	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 PSOC™ 64 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. The host should perform all read or write operations on the SWDIO line on the falling edge of SWDCK. PSOC™ 64 MCU performs read or write operations on SWDIO on the rising edge of SWDCK.
SWDIOTMS	Serial wire data input/output (SWDIO)	Yes	SWDIO is a bidirectional data input/output signal
XRES	External reset (XRES)	Yes	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.
GND	Ground(GND)	Yes	Negative supply voltage (ground)

Pin	SWD		Description
	Signal name	Mandatory	
VDD	Voltage drain drain (VDD)	NO [1]	Positive supply voltage. PSOC™ 64 MCU can be powered by external power supply or by programmer

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 1). The power on the PSOC™ 64 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™ 64 MCU and then starts sending the SW/JTAG commands. The XRES line is not used.

The programmer should implement PSOC™ 64 MCU acquisition in Reset mode. It is also the only way to acquire the PSOC™ 64 MCU if the board consumes too much current, which the programmer cannot supply. The Power Cycle mode support is optional and should be used only if a) the XRES pin is not available on the part's package; or b) the third-party programmer does not implement the XRES line but can supply power to the PSOC™ 64 MCU.

[1] VDD pin is mandatory for "Power Cycle" PSOC™ 64 MCU acquisition mode, where the programmer powers the PSOC™ 64 MCU and external power is not applied. For the "Reset" acquisition mode, the source of the power supplier does not matter, so the pin is optional.

5 Programming algorithm

This chapter describes in detail the programming flow of the PSOC™ 64 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 Constants and subroutines used in the programming flow. The ToggleReset() and Power() commands are also used (see Table 1).

5.1 High-level programming flow

Figure 8 shows the sequence of steps that must be executed to program the PSOC™ 64 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™ 64 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.

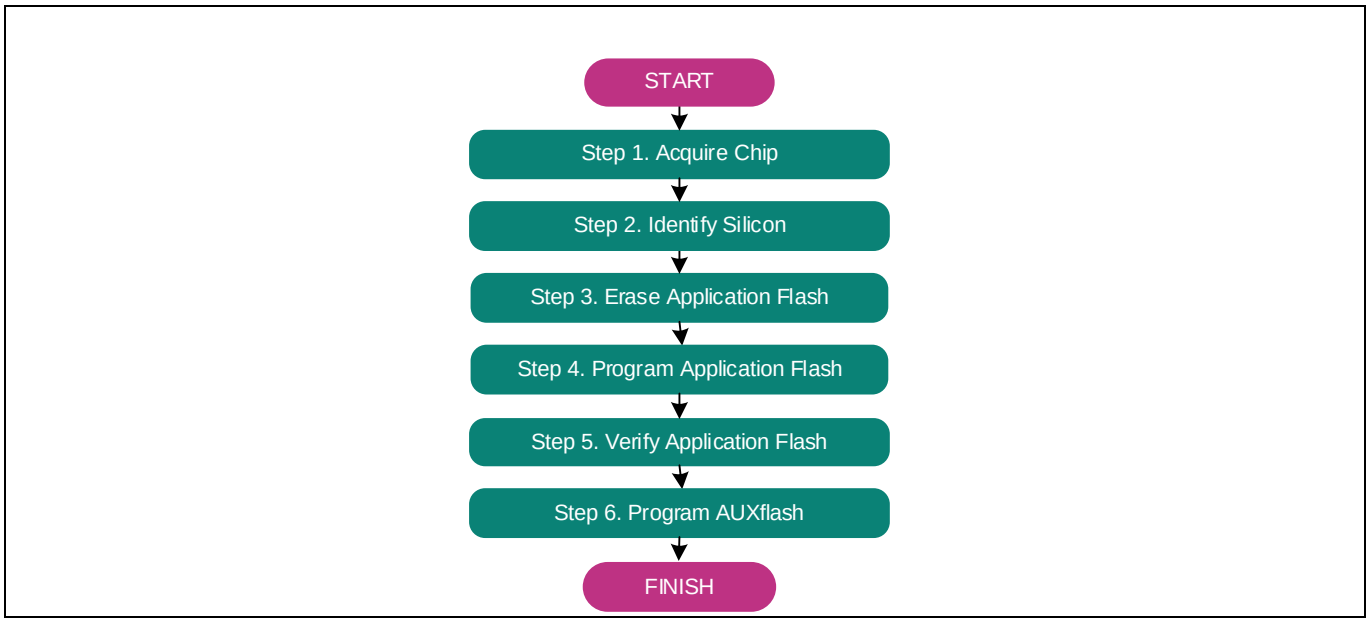


Figure 8 High-level programming flow of the PSOC™ 64 MCU

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(PSOC64_1M) /* Specific for 1M devices ------------------------
-----*/


/* Base addresses */
#define MEM_BASE_IPC                    0x40230000 // Base address for IPC
structures
/* IPC and Structures definitions */
#define IPC_INTR_STRUCT                 0x40231000 // IPC_INTR structure
address
#define IPC_STRUCT_LOCK_STATUS_OFFSET   0x10       // IPC lock status
/* RAM buffer address used to store SROM API parameters */
#define SRAM_SCRATCH_ADDR               0x0802E000
#define SRAM_SCRATCH_SIZE               0x1800
/* Acquisition acknowledge location (magic number address) */
#define SRAM_MAGIC_NUMBER_ADDR          0x08044804
#elif defined(PSOC64_2M) /* Specific for 2M devices ----------------------
-----*/


/* Base addresses */
#define MEM_BASE_IPC                    0x40220000 // Base address for IPC
structures
/* IPC and Structures definitions */
#define IPC_INTR_STRUCT                 0x40221000 // IPC_INTR structure
address
#define IPC_STRUCT_LOCK_STATUS_OFFSET   0x1C       // IPC lock status
/* RAM buffer address used to store SROM API parameters */
#define SRAM_SCRATCH_ADDR               0x080EE000
#define SRAM_SCRATCH_SIZE               0x1800
/* Acquisition acknowledge location (magic number address) */
#define SRAM_MAGIC_NUMBER_ADDR          0x080FE004
#else /* Specific for 512K devices ----------------------------------------
-----*/
/* Base addresses */
#define MEM_BASE_IPC                    0x40220000 // Base address for IPC
structures
```

```
/* IPC and Structures definitions */
#define IPC_INTR_STRUCT                 0x40221000 // IPC_INTR structure
address
#define IPC_STRUCT_LOCK_STATUS_OFFSET   0x1C       // IPC lock status
/* RAM buffer address used to store SROM API parameters */
#define SRAM_SCRATCH_ADDR               0x0802E000
#define SRAM_SCRATCH_SIZE               0x1800
/* Acquisition acknowledge location (magic number address) */
#define SRAM_MAGIC_NUMBER_ADDR          0x0803E004

#endif /* End of target-specific constants definition ---------------------
------*/
/* Base addresses */
#define MEM_BASE_FLASH                  0x10000000 // Base address of Main
flash
#define MEM_BASE_AUXFLASH               0x14000000 // Base address of
Auxiliary flash
#define MEM_SIZE_AUXFLASH               0x00008000 // Size of auxiliary
flash
/* Magic Number value */
#define SRAM_MAGIC_NUMBER_VAL           0x12344321
/* IPC and Structures definitions */
#define IPC_STRUCT_SIZE                 0x20
#define IPC_STRUCT2                     MEM_BASE_IPC + 2*IPC_STRUCT_SIZE //
DAP 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
/* SROM API masks */
#define SROMAPI_DATA_LOCATION_MSK 0x00000001 // [0]: 1 - arguments
IPC.DATA; 0 - in SRAM
#define SROMAPI_STATUS_MSK        0xF0000000
#define SROMAPI_STAT_SUCCESS      0xA0000000
/* Sys call IDs (SROM API Opcodes) */
#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
/* Misc. */
#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 4 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 2). 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 hardware access command to set the SWJ-DP in known state and reads the DP.IDCODE register. This sequence is repeated until the read IDCODE register request is acknowledged or until timeout.
bool DAP_Init ()	Initialize the Debug Port for programming operations.
bool Ipc_PollLockStatus (isLockExpected)	Depending on isLockExpected parameter, waits until LOCK status bit of IPC structure is released or acquired. The timeout is 1 second. Returns “true” (success) if LOCK status bit corresponds to desired status; otherwise, returns “false”.
bool Ipc_Acquire ()	Acquires IPC structure. The timeout is 1 second. 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). The timeout is 1 second. Output parameter data32 is the status/result word, provided by 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. Input parameter is the API opcode and parameters word. Output parameter data32 is the status/result word, provided by 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 Hardware access and Pseudocode. It uses the constants defined in this chapter.

```
//- CPU/MMIO registers Read/Write subroutines ----------------------------
-----------
bool WriteIO (addr32, data32) {
  ackOK = 3b'001 /*SWD*/;
  ack1 = Write_DAP (TAR, addr32);
  ack2 = Write_DAP (DRW, data32);
  return (ack1 == ackOK) && (ack2 == ackOK);
}

bool ReadIO (addr32, OUT data32) {
  ackOK = 3b'001 /*SWD*/;
  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  = 15000 ms;
  ackOK    = 3b'001     /*SWD*/;
  targetID = 0x6BA02477 /*SWD*/;
  // 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 {
    DAP_JTAGtoSWD();
    ack = Read_DAP (IDCODE, OUT id);
  } while ((ack != ackOK) && (TimeElapsed < timeout));
```

```
  return (TimeElapsed <= timeout) && (id == targetID);
}

bool DAP_Init (apNum) {
  // Handshake (e.g. after reset)
  if (DAP_Handshake() == false) return false;
  // 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 System Access Port and set bank 0 in APACC space
  Write_DAP (SELECT, 0);
  // 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;
}
```

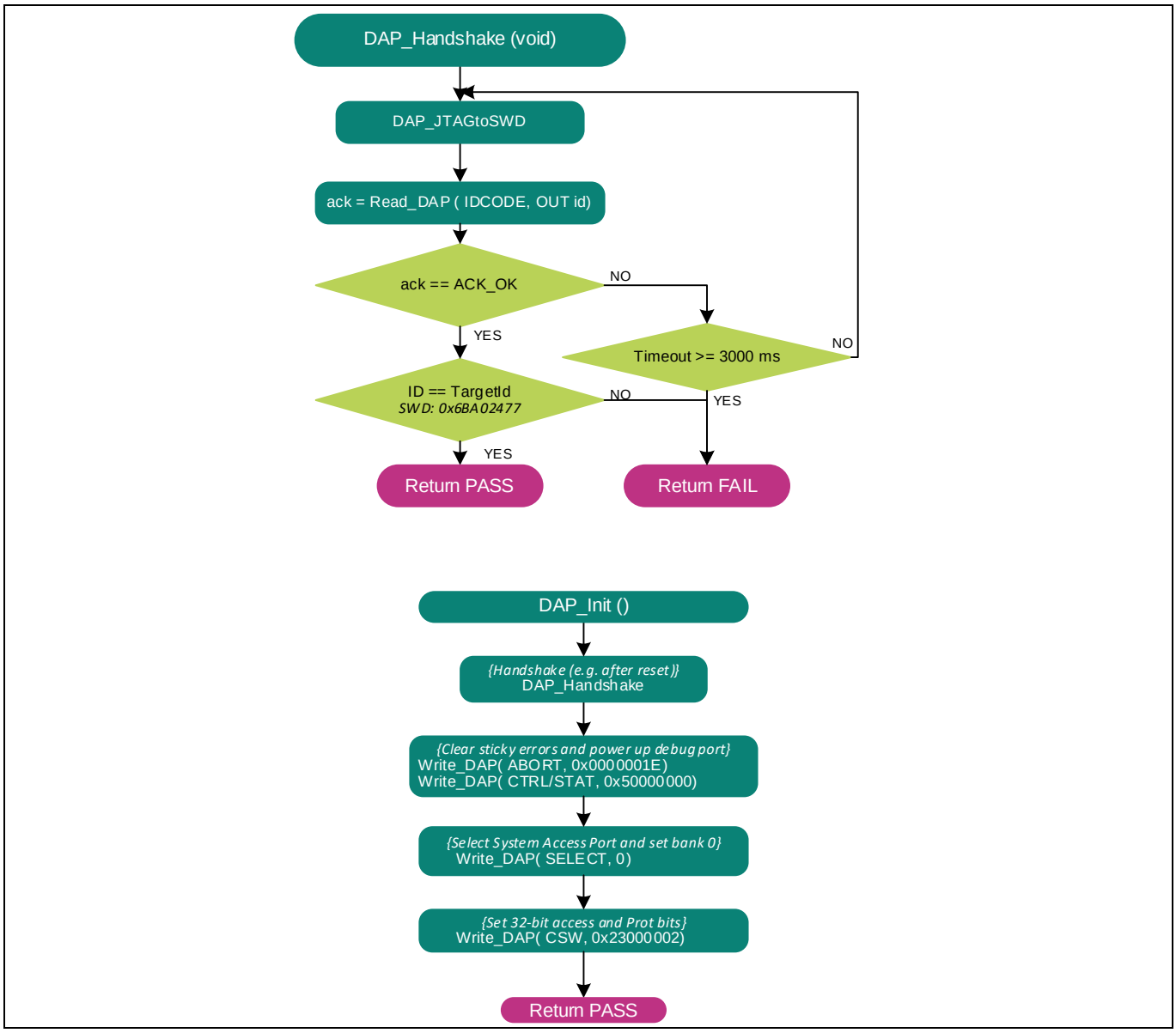


Figure 9 DAP initialization subroutines

```
//- SROM API usage subroutines ---------------------------------------------
----------
bool Ipc_PollLockStatus (isLockExpected) {
  ipcAddr = IPC_STRUCT2;
  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 () {
  ipcAddr = IPC_STRUCT2;
  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.
    status = WriteIO (ipcAddr + IPC_STRUCT_ACQUIRE_OFFSET,
                      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 DAP  (IPC_STRUCT2)
  ipcAddr = IPC_STRUCT2;

  // 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 the these to IPC_STRUCT[ipcId].DATA:
  // a) SROM function 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 + 2);
  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
  if (Ipc_PollLockStatus(ipcId, false) == false) return false;
```

```
  // Poll Data word
  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 – Acquire PSOC™ 64 MCU

The first step in programming the PSOC™ 64 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's code from the application flash region. After the user's code starts, it can repurpose the SWJ pins (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, 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 10 shows the timing diagram for entering Test mode.

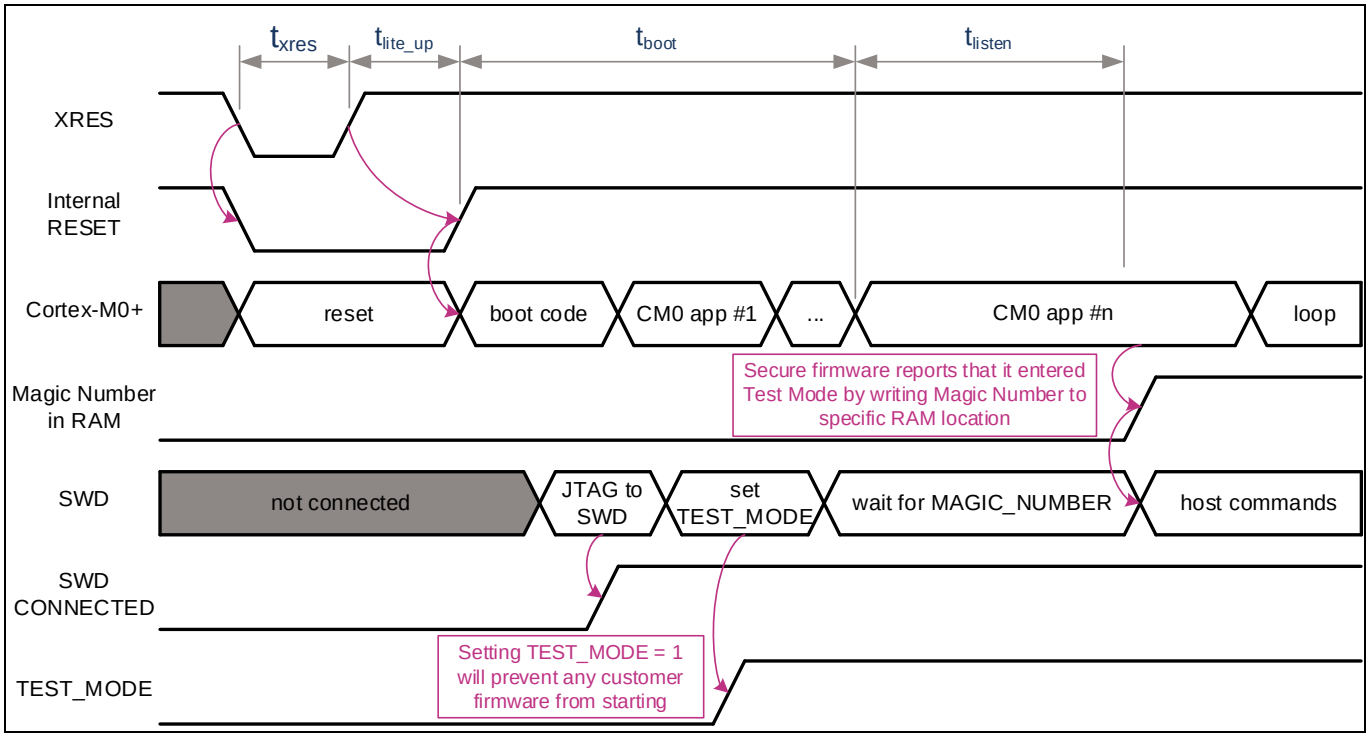


Figure 10 Timing diagram for Entering Test mode

Figure 10 details the chip's internal signals while entering Test mode. Everything starts from toggling the XRES line, so the chip enters Internal Reset mode for the t_{lite_up} period. After that, the system boot code starts execution. When completed, the boot code starts the first CM0+ application.

Depending on system configuration and provisioning, there may be several CM0+ applications started sequentially, but eventually one of them will enter the so-called “listen window” – it will wait for up to t_{listen} period allowing the debugger to set the TEST_MODE bit. At the end of listen window, the CM0+ application checks if the TEST_MODE bit is set, and if set, it will enter an endless loop preventing the next user application from running. It will also write a special ‘Magic Number’ to a predefined RAM location. The presence of the Magic Number is an indication to the debugger that the device has entered Test Mode successfully.

If the TEST_MODE bit is not present at the end of the listen window, the CM0+ application will proceed with the execution of the next application. Timing parameters may vary depending on boot code execution flow (see Table 5). Therefore, the best way to enter Test mode is to start sending an acquire sequence immediately after XRES is toggled. This sequence is sent iteratively until it succeeds (all SWJ transactions are ACKed and all conditions are met).

Table 5 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 (tlisten) 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	20	ms

Figure 11 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.

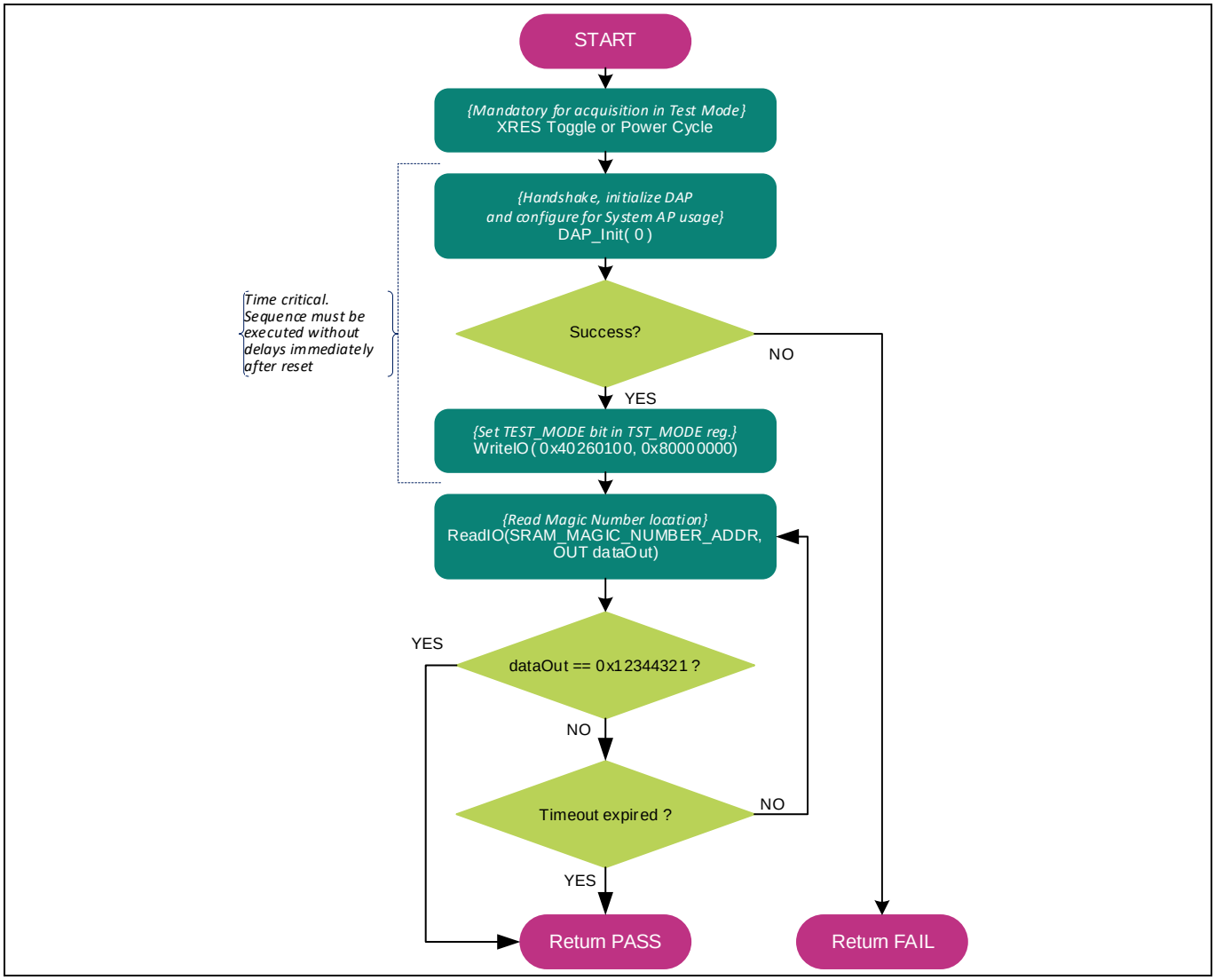


Figure 11 Flowchart of the Acquire Chip step

Code Listing 1 Pseudocode – Step 1.Acquire Chip

```
//-------------------------------------------------------------------------
-----------
// Reset Target
ToggleXRES(); // Toggle XRES pin, target must be powered

// 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 very time critical and must be executed
// without delays immediately after reset.
// No hurry for further steps - target already acquired in Test Mode
// Check TEST_MODE bit is set
ReadIO (SRSS_TST_MODE, OUT dataOut);
if ((dataOut & SRSS_TST_MODE_TEST_MODE_MSK) == 0) return FAIL;

timeout  = 15000 ms;
// Wait for the Magic Number.
// This is used as handshake between the debugger and target device.
do {
  ReadIO (SRAM_MAGIC_NUMBER_ADDR, OUT dataOut);
} while ((dataOut != SRAM_MAGIC_NUMBER_VAL) && (TimeElapsed < timeout));
// Absence of the Magic Number should not be treated as error.
// Flash programming will usually work fine even if timeout happened.
return PASS;
```

5.4 Step 2 – Identify silicon

This step is required to identify the acquired PSOC™ 64 MCU verify that it corresponds to the input data file.

Code Listing 2 Pseudocode – Step 2. Check Silicon ID

```
//-------------------------------------------------------------------------
--------------
// Read "Silicon ID" from the target using SROM request
// Type 1: Get Silicon ID and protection state
opCode = SROMAPI_SILID_CODE + (0x0000FF00 & (1 << 8));
status = CallSromApi(opCode, OUT dataOut);
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
// Identify Device Family:
// Device Family | Family ID | Si ID Range
// PSOC64_1M     | 0x100     | E200-E2FF
// PSOC64_2M     | 0x102     | E400-E4FF
```

```
// PSOC64_512K   | 0x105      | E700-E7FF
DeviceFamily = UNKNOWN;
if (FamilyIdHi == 0x01) {
  if ((FamilyIdLo == 0x00) && (SiliconIdHi == 0xE2))
    DeviceFamily = PSOC64_1M;
  else if ((FamilyIdLo == 0x02) && (SiliconIdHi == 0xE4))
    DeviceFamily = PSOC64_2M;
  else if ((FamilyIdLo == 0x05) && (SiliconIdHi == 0xE7))
    DeviceFamily = PSOC64_512K;
}

if (DeviceFamily == UNKNOWN || DeviceFamily != ExpectedFamily )
  return FAIL;
..return PASS;
```

5.5 Step 3 – Erase Application Flash

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

Code Listing 3 Pseudocode – Step 3. Erase Application Flash

```
//------------------------------------------------------------------------
--------------
   // Use 'EraseSector' SROM function to erase entire application flash by
256 KB sectors
totalSectors = 4; // Use 2 for PSOC64_512 KB, use 8 for PSOC64_2M
sectorSize = 0x40000;
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;
```

5.6 Step 4 – Program Application Flash

The 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 ID of the PSOC™ 64 MCU with expected value. This step uses the ProgramRow SROM function.

Figure 12 illustrates this programming algorithm.

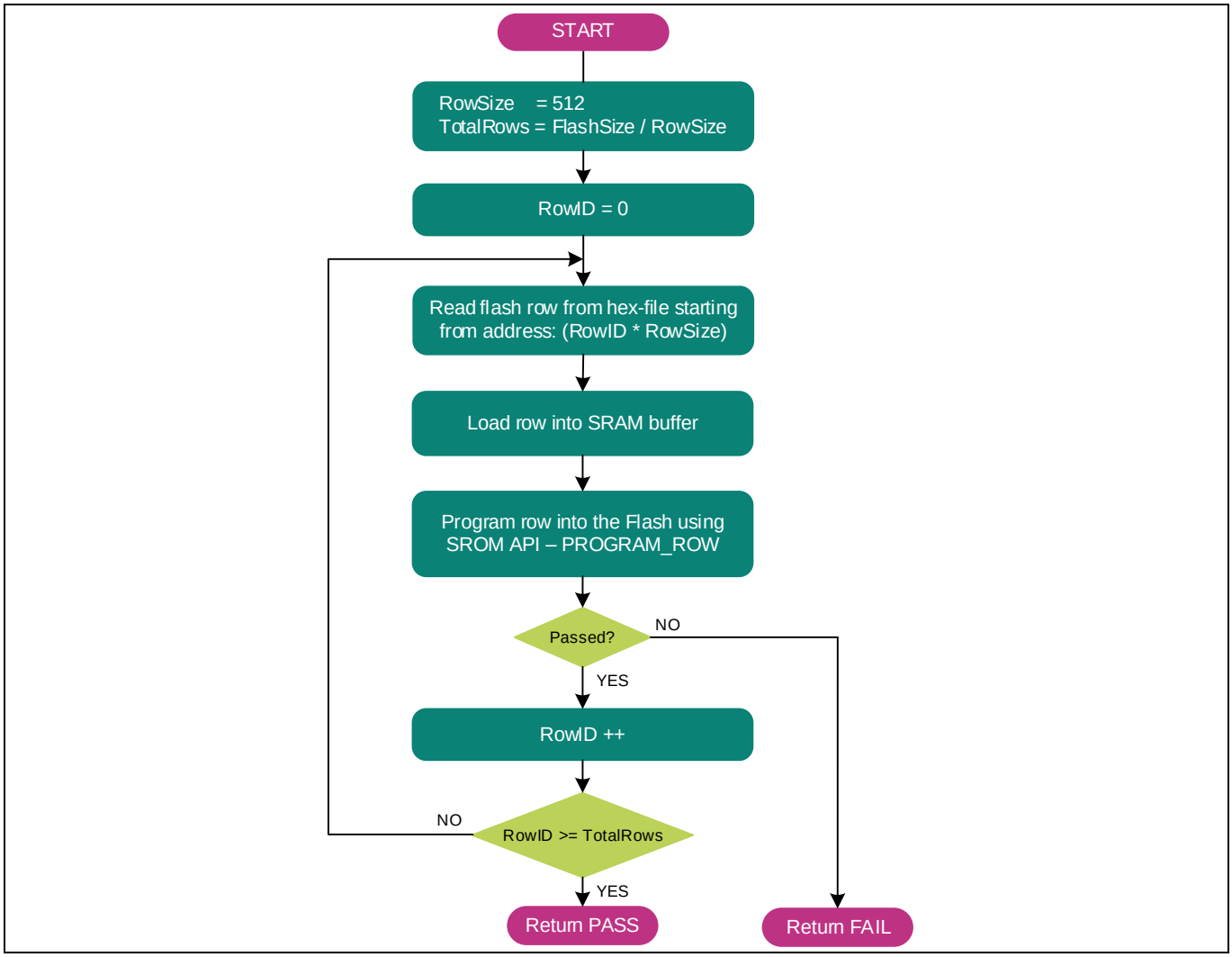


Figure 12 Flow Chart of the "Program Application Flash" Step

Code Listing 4 Pseudocode – Step 5. Program Application Flash

```
//--------------------------------------------------------------------------
--------------
// Flash Size must be provided.
totalRows = FlashSize / ROW_SIZE;

// Program all rows of the application flash
```

```
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
  WriteIO (SRAM_SCRATCH_ADDR, SROMAPI_PROGRAMROW_CODE);
  // SRAM_SCRATCH + 0x04: Data location/size and Integrity check
  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);
  // SRAM_SCRATCH + 0x08: Flash address to be programmed
  WriteIO (SRAM_SCRATCH_ADDR + 0x08, flashStartAddr);
  // SRAM_SCRATCH + 0x0C: Pointer to the first data byte location
  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;
```

5.7 Step 5 – 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:

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

Figure 13 illustrates the verification algorithm.

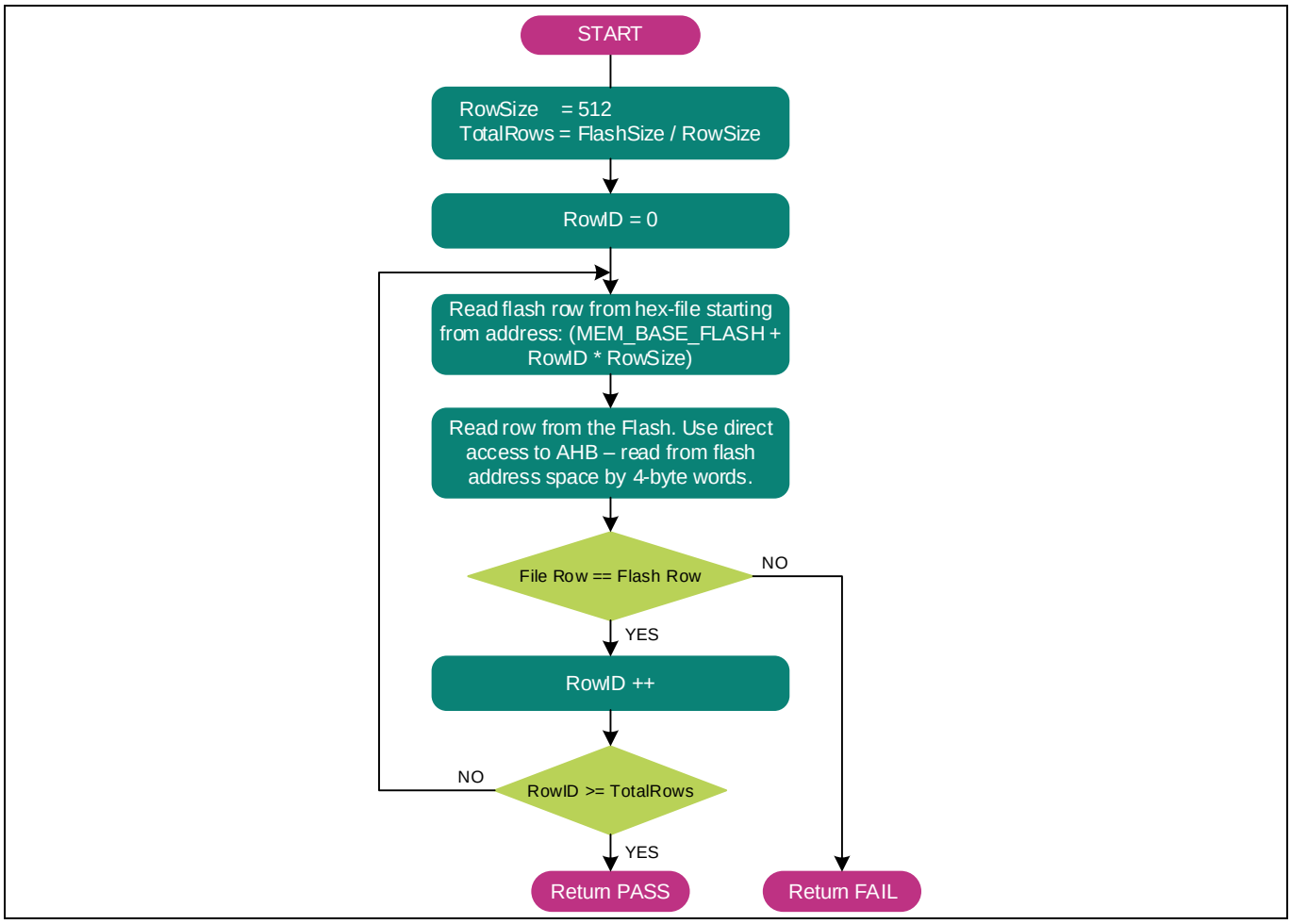


Figure 13 Flowchart of the "Verify Application Flash" Step

Code Listing 5 Pseudocode – Step 6. Verify Application Flash

```
// Flash Size must be provided.;
totalRows = FlashSize / ROW_SIZE;
// Read and Verify Flash rows
for (int rowID = 0; rowID < totalRows; rowID++) {
  //1. Read row from data file
  rowAddress = MEM_BASE_FLASH + rowID * ROW_SIZE; //line address of row in
flash
  // Extract 512-byte row from the binary file into buffer.
  // FILE_ReadData API must be implemented by Programmer.
  fileData[] = FILE_ReadData (rowAddress, ROW_SIZE);
  //2. Read row from chip
  for (i = 0; i < ROW_SIZE; i += 4) {
    //Read flash via AHB-interface
    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;
```

5.8 Step 6 – Program and Verify AUXflash

Note: Auxiliary flash is reserved for system use in PSOC 64_512K and cannot be programmed by the user.

Auxiliary flash (AUXflash) can store application-specific information and can be programmed using the same SROM functions as the application flash. However, 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, nonvolatile parameters, and so on.

AUXflash rows are the same size as the rows in the application flash region (512 bytes) and 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 14 illustrates the programming algorithm for AUXflash and Sflash regions.

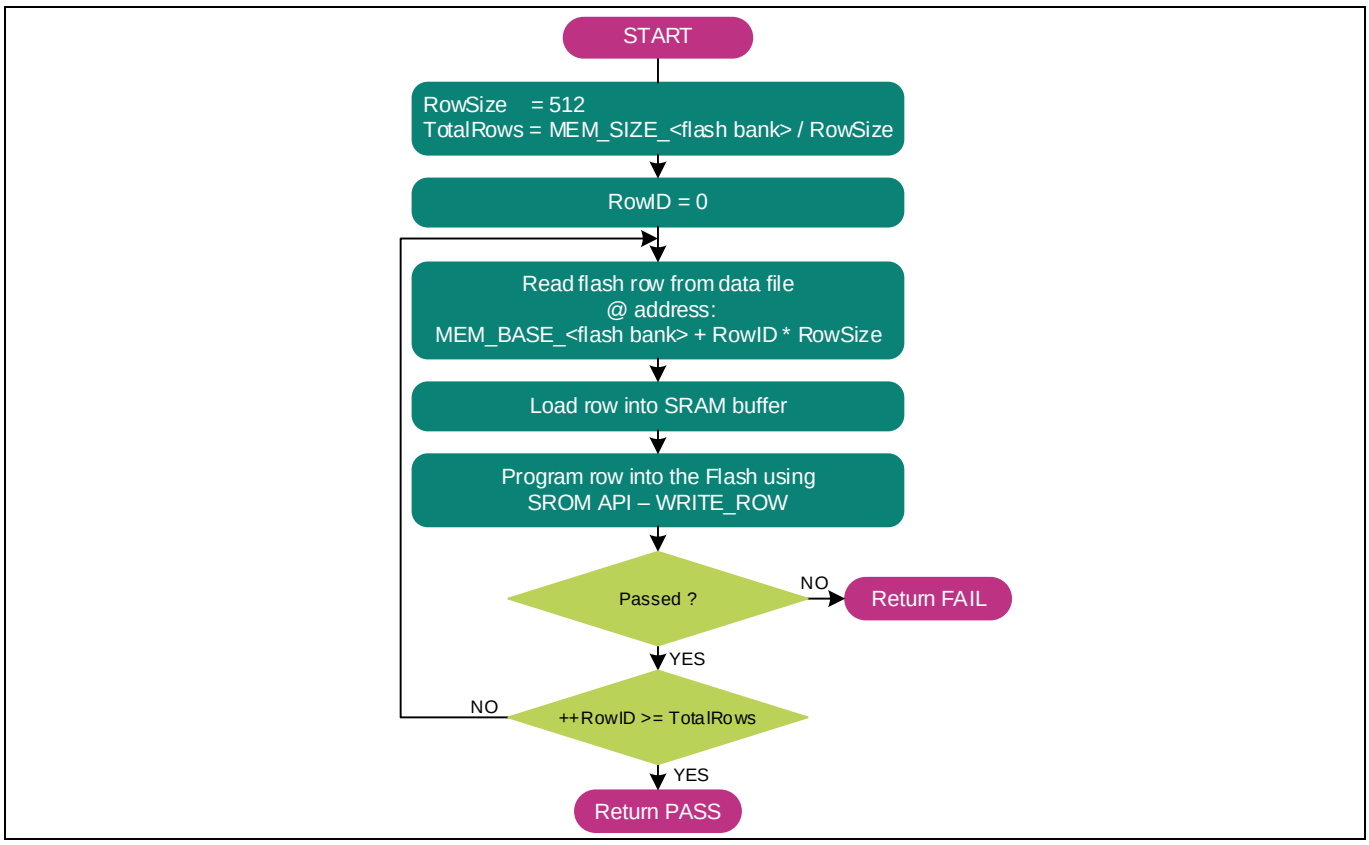


Figure 14 Flowchart of Program and Verify AUXflash step

Code Listing 6 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
  WriteIO (SRAM_SCRATCH_ADDR, SROMAPI_WRITEROW_CODE);
  // SRAM_SCRATCH + 0x04: Data location/size and Integrity check
  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);
  // SRAM_SCRATCH + 0x08: Flash address to be programmed
  WriteIO (SRAM_SCRATCH_ADDR + 0x08, address);
  // SRAM_SCRATCH + 0x0C: Pointer to the first data byte location
  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
  for (i = 0; i < ROW_SIZE; i += 4)  {
    // Read word of data (four bytes) from target's flash via AHB-interface
    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;
}
```

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 below:

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

Figure 15 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 a 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 of the different record types used in the hex file generated for the PSOC™ 64 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.

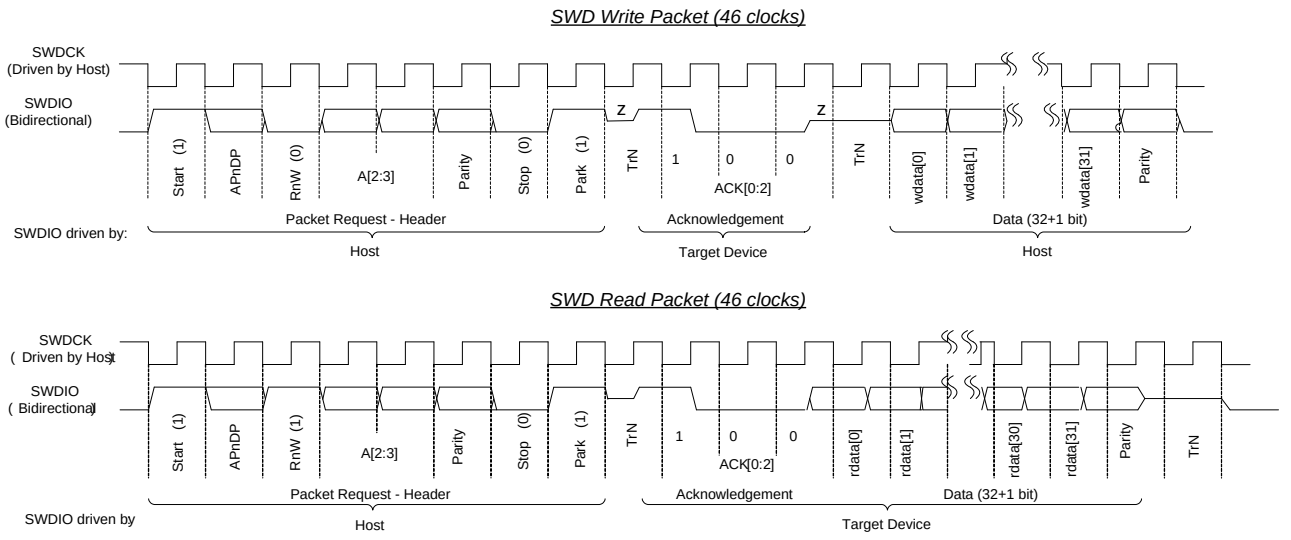
Note: *The data records of the multibyte regions in the hex file are all in little-endian format (LSB in lower address).*

7 Appendix B: 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™ 64 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™ 64 MCU (silicon).
- **Acknowledge Response** – The PSOC™ 64 MCU (silicon) sends an acknowledgment to the host.
- **Data Transfer Phase** – The data transfer is either from the PSOC™ 64 MCU to the host, following a read request (RDATA), or from the host to the PSOC™ 64 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 16 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 16 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 2 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 this bit is odd, then the parity must be '1', otherwise it is '0'.
- If the parity bit is not correct, the PSOC™ 64 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 6. 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™ 64 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™ 64 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™ 64 MCU is processing the previous transaction. The host can try for a maximum of 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 6 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™ 64 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 16. According to the SWD protocol, both the host and the PSOC™ 64 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™ 64 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 cycles long. Neither the host nor the PSOC™ 64 MCU should drive the SWDIO line during the TrN phase, as indicated by 'z' in in Figure 16.
- The address, ACK, and read and write data are always transmitted to 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.

Revision history

Document revision	Date	Description of changes
**	2020-10-09	Initial release.
*A	2021-03-10	Updated to Infineon template.
*B	2025-01-28	Added missed IPC_INTR_STRUCT_INTR_MASK_OFFSET definition. Unused definitions deleted. Updated the PSOC branding. Added PSOC disclaimer.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

Edition 2025-01-28

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2025 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-31353 Rev.*B

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”).

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.