XMC5xxx MCU programming specification

About this document

Scope and purpose

This specification provides the information to program the non-volatile memory of the XMC5100/XMC5200/XMC5300 MCU families, describes the communication protocol to provide access for an external programmer and explains the programming algorithm.

Intended audience

This document is written for experienced hardware and software engineers that require information on how to program non-volatile memory of the XMC5000 MCU.

Table of contents

1 Introduction

This programming specification:

- Provides the information to program the non-volatile memory of the XMC5000 MCU family
- Describes the communication protocol to provide access for an external programmer
- Explains the programming algorithm
- Describes the physical connection basics

The programming algorithms described hereafter are compatible with all XMC5000 MCUs.

The pin locations, electrical, and timing specifications of the physical connection are not part of this document. For these details, see the device datasheet.

1.1 Programmer

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

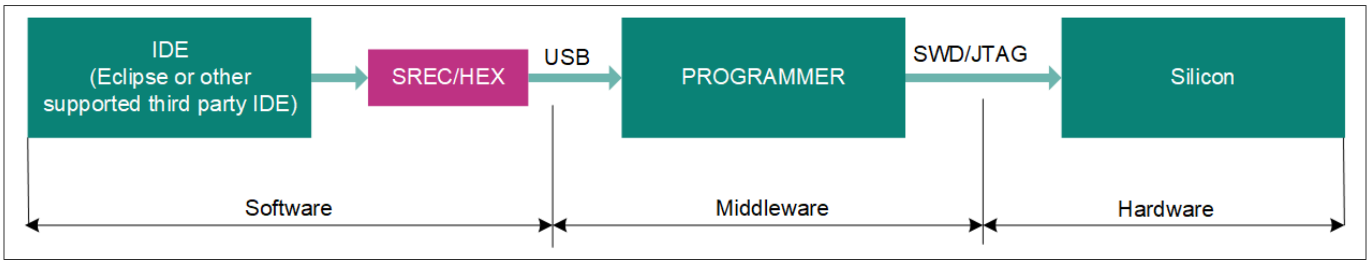


Figure 1 **Figure 1 Programmer in development environment**

In the manufacturing environment, the integrated development environment (IDE) block is absent because its main purpose is to produce a hex file. As shown in Figure 1, the programmer performs three functions:

- Parses the srec/hex file and extracts information
- Interfaces with the XMC5000 MCU silicon using a Serial Wire Debug (SWD) or JTAG master
- Implements the programming algorithm by translating the srec/hex data into SWD or JTAG signals

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

In a software-centric structure, the programmer's hardware works as a bridge between the protocol (such as USB) and SWD or JTAG. An external device (software) passes all SWD/JTAG commands to the hardware through the protocol. The bridge is not involved in parsing the hex file and programming algorithm. This is the task of the upper layer (software). Examples of such programmers are MiniProg4 and Segger J-Link.

A firmware-centric structure is an independent hardware design in which all functions of the programmer are implemented in one device, including storage for the hex 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 XMC5000 MCU family overview

The XMC5000 is a family of MCUs with dual solution, with both the Arm® Cortex®-M4F and Cortex®-M0+ processor cores. This MCU family supports the Arm® SWJ-DP Interface for programming and debugging operations, using the SWD or JTAG protocols.

The non-volatile subsystem of the chip consists of a flash memory system. The flash memory system stores the user's program.

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

1 Introduction

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

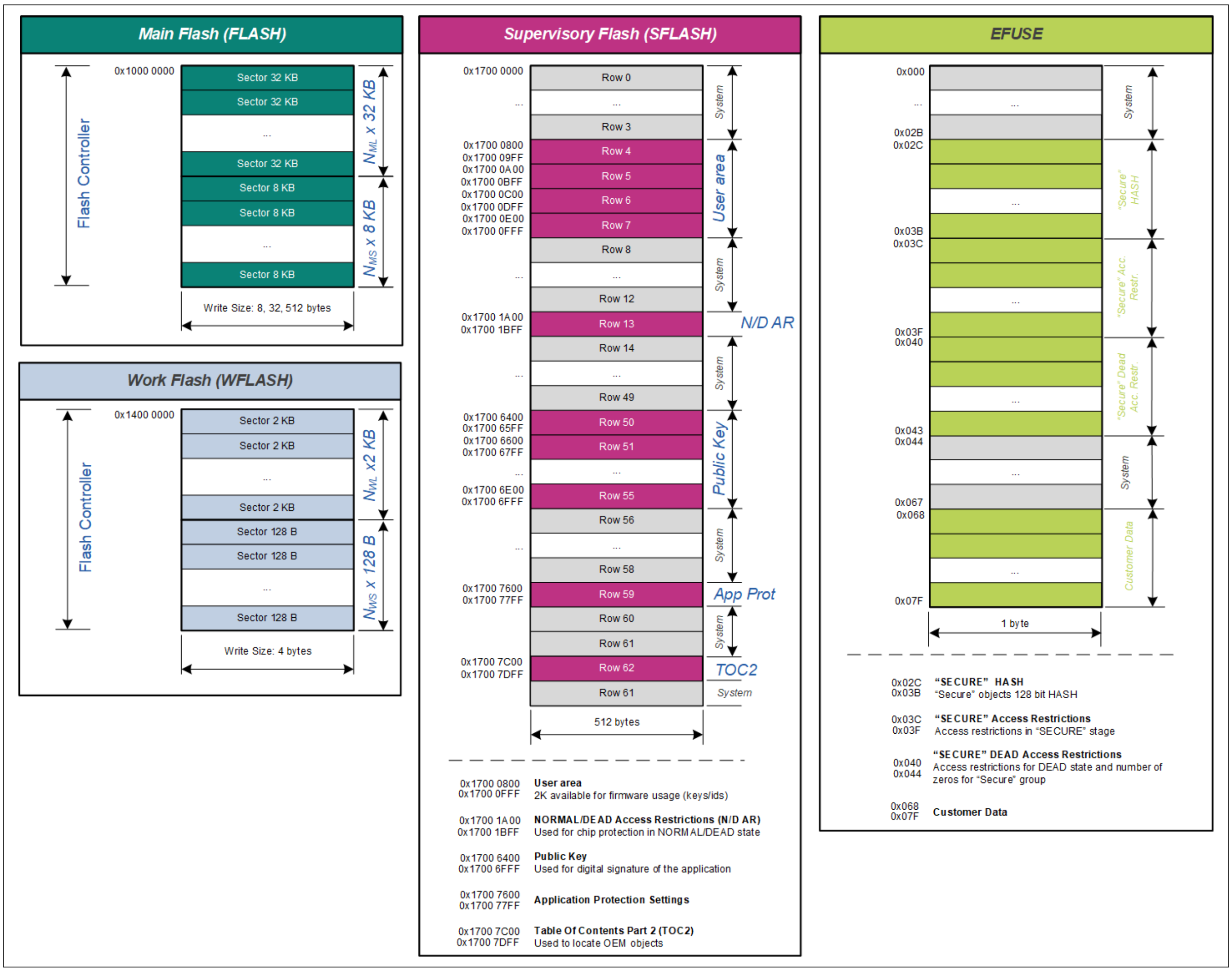
This document focuses on the specific programming operations without referencing the chip architecture. Many important topics are detailed in the appendices. Other device-specific information can be found in the device's datasheet or technical reference manual.

This document includes three appendices:

- Appendix A: Intel hex file format
- Appendix B: Serial wire debug (SWD) protocol
- Appendix C: Joint test action group (JTAG) protocol

2 Non-volatile memory subsystem

This chapter describes the non-volatile memory subsystem of the XMC5000MCU.



0x1700 0800 0x1700 0FFF	**User area** 2K available for firmware usage (keys/ids)
0x1700 1A00 0x1700 1BFF	**NORMAL/DEAD Access Restrictions (N/D AR)** Used for chip protection in NORMAL/DEAD state
0x1700 6400 0x1700 6FFF	**Public Key** Used for digital signature of the application
0x1700 7600 0x1700 77FF	**Application Protection Settings**
0x1700 7C00 0x1700 7DFF	**Table Of Contents Part 2 (TOC2)** Used to locate OEM objects

0x02C 0x03B	**"SECURE" HASH** "Secure" objects 128 bit HASH
0x03C 0x03F	**"SECURE" Access Restrictions** Access restrictions in "SECURE" stage
0x040 0x044	**"SECURE" DEAD Access Restrictions** Access restrictions for DEAD state and number of zeros for "Secure" group
0x068 0x07F	**Customer Data**

Figure 2 **XMC5000MCU non-volatile subsystem**

Table 1 **Programming parameters of XMC5000 MCU**

Feature	**XMC5100**	**XMC5200**	**XMC5300**
Start address	0x1000 0000		
Number of large (32 KB) sectors (N_{ML})	14	30	62
Number of small (8 KB) sectors (N_{MS})	16	16	16
Write (Row) size, B	8, 32, and 512		
Erase size	Sector size		
Total size, KB	576	1088	2112

(table continues...)

Table 1 (continued) Programming parameters of XMC5000 MCU

Feature	XMC5100	XMC5200	XMC5300
Flash work region (WFLASH)			
Start address	0x1400 0000		
Number of large (2 KB) sectors (N_{WL})	24	36	48
Number of small (128 B) sectors (N_{WS})	128	192	256
Write (Row) size, B	4		
Erase size			
Total size, KB	64	96	128

Table 2 SFlash/eFuse programming parameters for XMC5000 MCUs

SFLASH	
Start address	0x1700 0000
Sectors	Single sector of 32 KB
Write (row) size, B	512
Erase size	N/A. Required a WriteRow API call with empty data
Total size, KB	32
SFLASH_ALT	
Start address	0x1780 0000
Sectors	Single sector of 32 KB
Note	This Flash region is not intended to be programmed by the user. The purpose of this region is to store backup copy of the SFLASH. When system call is interrupted during SFLASH programming (e.g. due to power loss) the contents of the SFLASH can be damaged. A special recovery mechanism is implemented in the boot SROM which will restore SFLASH contents from the backup.
EFUSE	
Start address	0x9070 0000 (virtual address for OpenOCD, see Electronic fuses (eFuse) for details)
Sectors	Single sector of 128 B
Write (row) size	Single bit
Erase size	N/A. One-time programming
Total size, KB	0,128

* FLASH, WFLASH, and SFLASH memories are mapped directly to the CPU's address space. Firmware or an external programmer can read its content directly from the given address.

2.1 Flash main region (FLASH)

The flash contains code to be executed. See Figure 2 and Table 1 for details.

2.2 Flash work region (WFLASH)

The flash contains the WFLASH to store application-specific information or emulate EEPROM memory. See Figure 2 and Table 1 for details.

2.3 Flash supervisory region (SFLASH)

The flash contains the SFLASH to store application-specific data. See Figure 2 and Table 1 for details.

Table 3 SFLASH's regions

Address region	Description
0x1700 0800 - 0x1700 0FFF	The user's area. 2 KB are used to store arbitrary data.
0x1700 1A00 - 0x1700 1A03	NORMAL Access Restrictions (NAR). Used for chip protection in the NORMAL Life Cycle stage.
0x1700 1A04 - 0x1700 1A07	NORMAL DEAD Access Restrictions (NDAR). Used for chip protection in the NORMAL DEAD Life Cycle stage.
0x1700 6400 - 0x1700 6FFF	The Public Key. Used for a digital signature of the application
0x1700 7600 - 0x1700 77FF	Application protection settings
0x1700 7C00 - 0x1700 7DFF	The Table of Contents, Part 2 (TOC2). Used to locate OEM objects.

The accessible SFLASH sub-regions are listed in Table 5. Writing to the sub-regions is not possible at the "SECURE" chip life cycle stage. Writing to any SFLASH address outside the specified sub-regions is not possible at any Life Cycle stage except VIRGIN which is a factory-only stage.

2.4 Electronic fuses (eFuse)

Electronic fuses (eFuses) are 1024-bit (128 bytes) non-volatile memories where each bit is a one-time programmable (OTP). The eFuses are implemented as a regular Advanced High-performance Bus (AHB) peripheral with the following characteristics and assumptions:

- 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 again.
- Programming a fuse 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 is set to 3.0-5.5 V.
- Eight bits of the eFuse array can be read at a time using ReadFuseByte system call.
- Fuses are programmed one bit at a time using BlowFuseBit system call.

See Figure 2 and Table 1 for details.

Table 4 eFuse regions

Address region	Description
0x2C - 0x3B	"SECURE" HASH. The 128-bit (16 bytes) AES_HASH is used by boot code to authenticate objects in the Table of Contents, Part 2 (TOC2).

(table continues...)

Table 4 (continued) eFuse regions

Address region	Description
0x3C – 0x3F	“SECURE” Access Restrictions (SAR). Access restrictions for “SECURE” protection state in “SECURE” lifecycle stage.
0x40 - 0x43	“SECURE” DEAD Access Restrictions (SAR). Access restrictions for DEAD protection state in “SECURE” lifecycle stage and number of zeros for “Secure” fuse group
0x68 – 0x7F	Customer data

Table 4 lists the eFuse bytes accessible for production programming.

eFuses can be read or programmed using two dedicated system calls – ReadFuseByte and BlowFuseBit.

ReadFuseByte call reads one data byte (eight eFuse bits) at a time. It accepts eFuse byte offset as a parameter. The value of this parameter directly corresponds to the address shown in Figure 2. For instance, to read first byte of the ‘Customer Data’ region a value of 0x68 should be passed to ReadFuseByte system call.

BlowFuseBit call blows a single eFuse bit at a time. It accepts three parameters to identify eFuse bit to be blown: byte address, macro address, and bit address. These parameters have the following relation with the eFuse byte offset as shown in Figure 2:

- macro_address = byte_offset % 4
- byte_address = byte_offset / 4
- bit_address is the bit to be blown in the selected byte

For instance, to blow the least significant bit (LSb) of the first byte of the ‘Customer Data’ region (byte offset = 0x68, bit #0) these parameters can be calculated as follows:

- macro_address = 0x68 % 4 = 0x00
- byte_address = 0x68 / 4 = 0x1A
- bit_address = 0

The OpenOCD tool supports eFuse programming with a data stored in regular HEX, SREC, ELF or binary files. A virtual address range has been allocated for eFuse region for this purpose. This address range does not correspond to any internal hardware, instead, it is treated in a special way by the flash utility. The rules of the translation between virtual address range and physical eFuse bits is as follows:

- The virtual address range is 0x9070 0000 … 0x9070 03FF (1024 bytes in total)
- Each byte in the programming file corresponds to single eFuse bit (1024 eFuse bits in total)
- The bit order is LSB to MSB, e.g. the second bit of the ‘Customer Data’ region (byte offset = 0x68, bit #1) corresponds to the virtual address 0x90700000 + 0x68 * 8 + 1 = 0x90700341

Each byte in the programming file can have three possible values:

- 0x00 – Do not blow the eFuse bit. OpenOCD will check the actual state of the bit and will show an error if eFuse bit is blown, but data in programming file indicates that it should not be blown.
- 0x01 – Blow the eFuse bit. OpenOCD will check the actual state of the bit and will show a warning if eFuse bit is already blown.
- 0xFF – Ignore the state of the eFuse bit. The actual state of the bit is not checked, no warnings/errors will be generated.

Blowing an eFuse is an irreversible process, programming is recommended only in mass production programming under controlled factory conditions and not at prototyping stages. For more details, see the “eFuse Memory” section of the architecture technical reference manual (TRM).

3 Hex file

This chapter describes the information the programmer must extract from the hex file to program the XMC5000 MCU.

3.1 Organization of the hex file

The hexadecimal (hex) file describes the non-volatile configuration of the project. This file is the data source for the programmer.

The hex file for the XMC5000 MCU follows the Intel hex file format. Intel's specification is generic and defines only some types of records that can compose the hex file. The specification allows customizing the format for any possible chip architecture. The chip vendor defines the records' functional meaning which typically varies for different chip families. See Appendix A: Intel hex file format for details of the Intel hex file format.

The XMC5000 MCU defines the following data sections in the hex file:

- User's program (code) for FLASH region
- User's data for WFLASH region
- User's / OEM data for SFLASH region
- eFuse region

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

The programmer uses hex addresses (Figure 3) to read the sections from the hex file into its local buffer. Later, this data is programmed (translated) into the corresponding addresses of the chip.

See Table 1 for the region addresses described in the section.

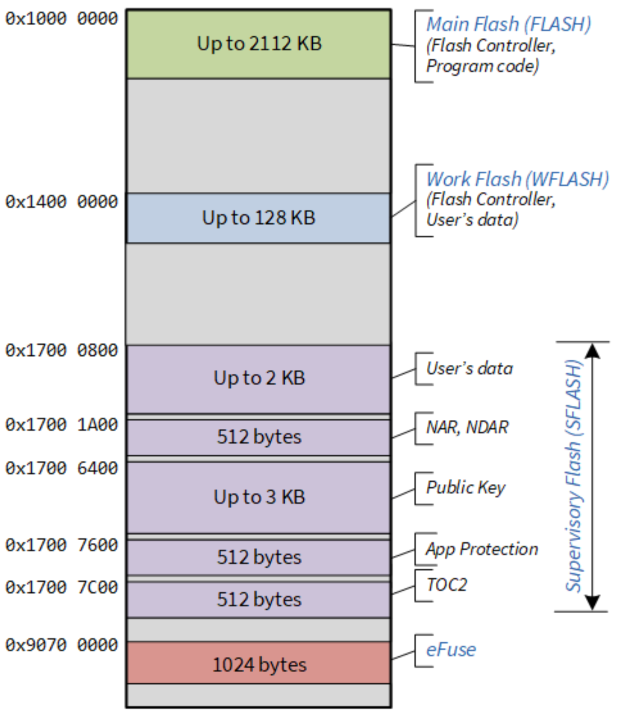


Figure 3 **Hex file organization for XMC5000 MCU**

FLASH: This is the user's code that must be programmed. The maximum size of this section must not exceed the FLASH size of the XMC5000 MCU. See Flash main region (FLASH) for the FLASH memory region description.

WFLASH: Stores application-specific information. The availability of this section in the hex file is optional and depends on the linker scripts usage in the user's project. See Flash work region (WFLASH) for the WFLASH memory region description.

SFLASH: Stores application-specific information in the five fragmented sub-sections. The availability of this section in the hex file is optional and depends on the linker scripts usage in the user's project. See Flash supervisory region (SFLASH) for the SFLASH memory region description.

eFuse: Each eFuse bit setting is stored in the hex file as a full byte. This is done for two reasons:

- The programmer can distinguish between bytes being set and bytes that are not cared about, or bytes whose value are not know
- More accurate reflection of how these bits are programmed, because the SROM API sets one bit per call.

The values are: 0x00 – Not blown, 0x01 – Blown, and 0xFF – Ignore.

***Note**:* *The programmer can only perform the "not blown" operation first and then the "blown" operation. First, the programmer reads the corresponding eFuse bit from the device and blows it only if the device value is 0 (not blown) and the hex value is 0x01 (blown). See* Electronic fuses (eFuse) *for more details.*

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 the hardware and software components.

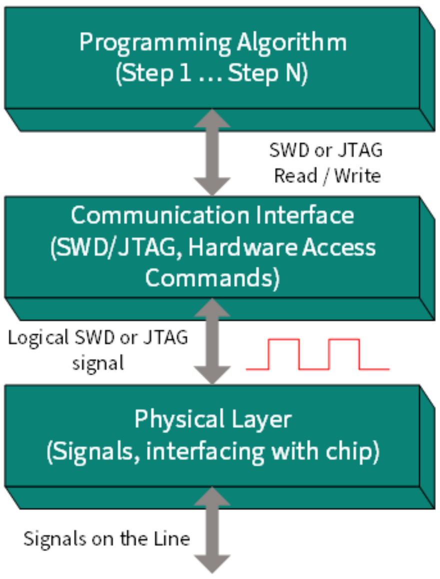


Figure 4 **Programmer's protocol stack**

- **The 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 APIs, stored in SROM (SROM APIs). 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 the 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 (4-byte for WorkFlash; 8-, 32- or 512-byte for MainFlash; 512-byte for SFlash).

  The Programming Algorithm protocol is the fundamental part of this specification. For more information on this algorithm, see 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 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 chip. The Serial Wire Debug (SWD) or the JTAG interface can be used as the communication protocol between the external device and the MCU.

4.1.1 Program and debug interface

The main purpose of the Program and Debug Interface is to support programming and debugging through the JTAG or SWD interface and to provide the Read/Write access to all memory and registers in the system during debugging, including the Cortex®-M4 and Cortex®-M0+ register banks when the core is running or halted.

The MCU implements a Debug Access Port (DAP), which integrates the SWJ-DP (Serial Wire/JTAG Debug Port) and complies with the Arm® specification ARM® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 (ARM IHI 0031C).

The debug port (DP) connects to the DAP bus which in turn connects to one of the Access Ports (AP):

- **CM0-AP** – Connects directly to the AHB debug slave port of the CM0+ and provides access to the CM0+ internal debug components and to the rest of the system through the CM0+ AHB master interface. This AP provides the debug host the same view as an application running on the CM0+ including the access to the MMIO of other debug components of the Cortex® M0+ subsystem. These debug components are accessed by the CM0+ CPU, but cannot be accessed through the other APs or by the CM4 core.
- **CM4-AP** – Locates inside the CM4, provides access to the CM4 internal debug components and 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 including the access to the debug components in the CM4 core through the External Peripheral Bus (EPB). These debug components are accessed by the CM4 CPU, but cannot be accessed through the other APs or by the CM0+ core.
- **System-AP** – Provides access to the rest of the system and to the System ROM table which cannot be accessed in any other way.

4.1.1.1 DAP security

The 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 architecture TRM for more details of the security settings for the XMC5000 MCU.

4.1.1.2 DAP power domain

Almost all debug components are part of the Active power domain. The only exception is the SWD/JTAG-DP, which is part of the DeepSleep power domain. This allows the debug host to connect during DeepSleep mode, while the application is running or powered down. This enables the infield debugging for low-power applications in which the chip is mostly in DeepSleep 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 DP.CTL/STAT.CDBGPWRUPREQ and DP.CTL/STAT.CSYSPWRUPREQ they 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 (SWJ_SWCLK_TCLK and SWJ_SWDIO_TMS) are operational during DeepSleep power mode – the JTAG pins are operational only in Active mode. The JTAG debug and JTAG boundary scan are not available when the system is in DeepSleep mode.

4.1.1.3 SWD/JTAG selection

The JTAG and SWD are mutually exclusive because of Arm’s SWJ-DP implementation and sharing pins. An external programmer/debugger must be able to switch to the required protocol. The watcher circuit implemented in the SWJ-DP detects a specific 16-bit select sequence on SWJ_SWDIO_TMS and decides if the JTAG or SWD interface is active. By default, the JTAG operations are selected on a power-up reset and the JTAG protocol can be used from the reset without sending a select sequence. The protocol switches only when the selected interface is in its reset state (test-logic-reset for JTAG and line-reset for SWD).

To switch the SWJ-DP from the JTAG to SWD operation:

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

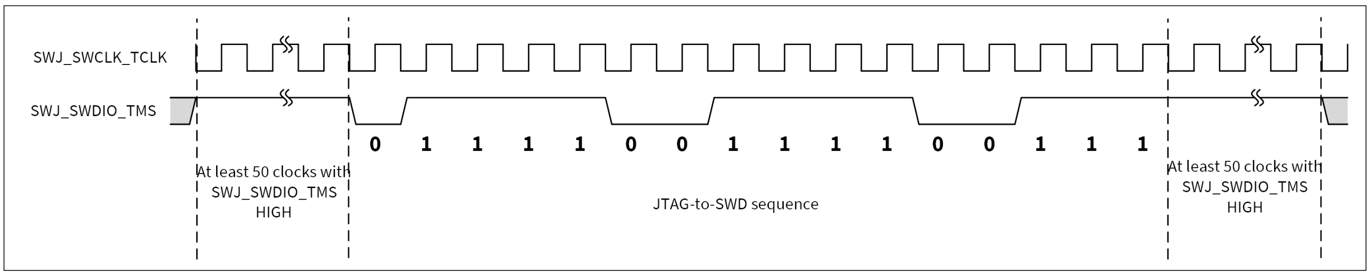

SWJ_SWCLK_TCLK
SWJ_SWDIO_TMS
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1
At least 50 clocks with SWJ_SWDIO_TMS HIGH
JTAG-to-SWD sequence
At least 50 clocks with SWJ_SWDIO_TMS HIGH

Figure 5 **JTAG-to-SWD sequence timing**

To switch the SWJ-DP from the SWD to JTAG operation:

- Send 50 or more SWJ_SWCLK_TCLK cycles with SWJ_SWDIO_TMS HIGH. This ensures 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 to SWJ_SWDIO_TMS: first 0b0011 1100 1110 0111, MSb. This can be represented as 0x3CE7, MSb transmitted first or 0xE73C, LSb transmitted first.
- Send five or more SWJ_SWCLK_TCLK cycles with SWJ_SWDIO_TMS HIGH. This ensures the JTAG TAP enters the test-logic-reset state if before sending the select sequence the SWJ-DP was already in the JTAG operation.

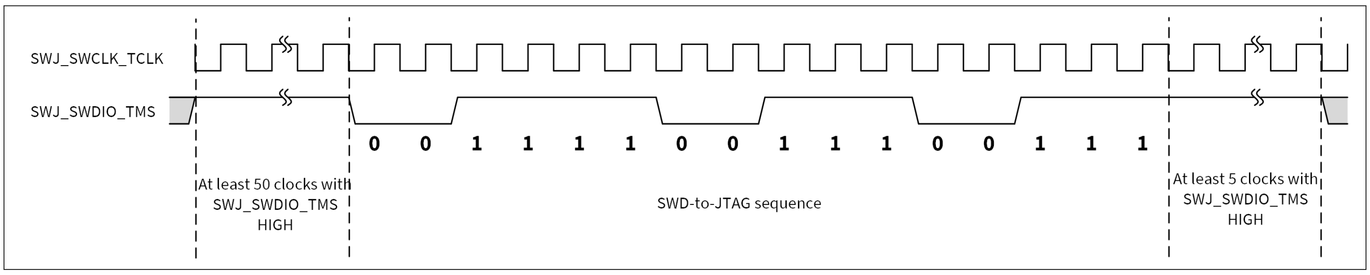

SWJ_SWCLK_TCLK
SWJ_SWDIO_TMS
0 0 1 1 1 1 0 0 1 1 1 0 0 1 1 1
At least 50 clocks with SWJ_SWDIO_TMS HIGH
SWD-to-JTAG sequence
At least 5 clocks with SWJ_SWDIO_TMS HIGH

Figure 6 **SWD-to-JTAG 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. The upper protocol, Programming Algorithm, requires two extra commands to manipulate the hardware: Power(state) and ToggleReset(). Table 5 lists the hardware access commands used by the software layer.

Table 5 **Hardware access commands**

Command	Parameters	Description
DAP_JTAGtoSWD		A standard Arm® command to switch the SWJ-DP from the JTAG to SWD operation. This sequence synchronizes the programmer and chip; it is the first transaction in a programming flow. See SWD/JTAG selection for the implementation details.
SWD_Write	IN APnDP, IN addr, IN data32, OUT ack	Sends 32-bit data to the DAP specified register 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 DAP specified register using the SWD interface. The register is defined by the APnDP (1 bit) and addr (2 bit) parameters. The DAP returns 32-bit data, the 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 DAP specified register using the JTAG interface. The register is defined by the APnDP (1 bit) and addr (2 bit) parameters. The DAP returns the 3-bit status in ack.
JTAG_Read	IN APnDP, IN addr, OUT data32, OUT ack	Reads 32-bit data from the DAP specified register using the JTAG interface. The register is defined by the APnDP (1 bit) and add" (2 bit) parameters. The DAP returns 32-bit data and the status.
ToggleReset		Generates a reset signal for the XMC5000 MCU. The programmer must have a dedicated pin connected to the XRES_L pin of the XMC5000 MCU.
Power	IN state	If the programmer powers the XMC5000 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. For information on the structure of the JTAG, see Appendix C: Joint test action group (JTAG) protocol.

The SWJ Read/Write commands allow accessing registers of the SWJ-DP module from Figure 5. The DAP functionally is split into two control units:

- Debug Port (DP) – 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).

An 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 6 shows the DAP registers used during programming.

Table 6 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
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. 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 6). The pseudo-commands correspond to the Read or Write SWJ transactions. Examples:

```
Write_DAP (TAR, MEM_BASE_SRAM)
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, the 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 failed. Depending on the programming context, the 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 5) is as follows.

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

//-------------------------------------------------------------------------------
SWJ_Status Read_DAP (Register, OUT data32)
{
    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
        {
            Parity_data32 = 0x00;
            for (i=0; i<32; i++)
                Parity_data32 ^= ((data32 >> i) & 0x01);
            if (Parity_data32 != parity)
                Ack = 3'b111; //NACK
        }
}
    else if (Interface == “JTAG”)
        JTAF_Read (Register.APnDP, Register.Addr, OUT data32, OUT Ack);

    return Ack;
}
//-------------------------------------------------------------------------------
```

The programming code in the Programming algorithm is based mostly on the Write and Read pseudo-commands and some commands in Table 5.

4.2 Physical layer

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

See the device datasheet for the pins location, electrical, and timing specifications.

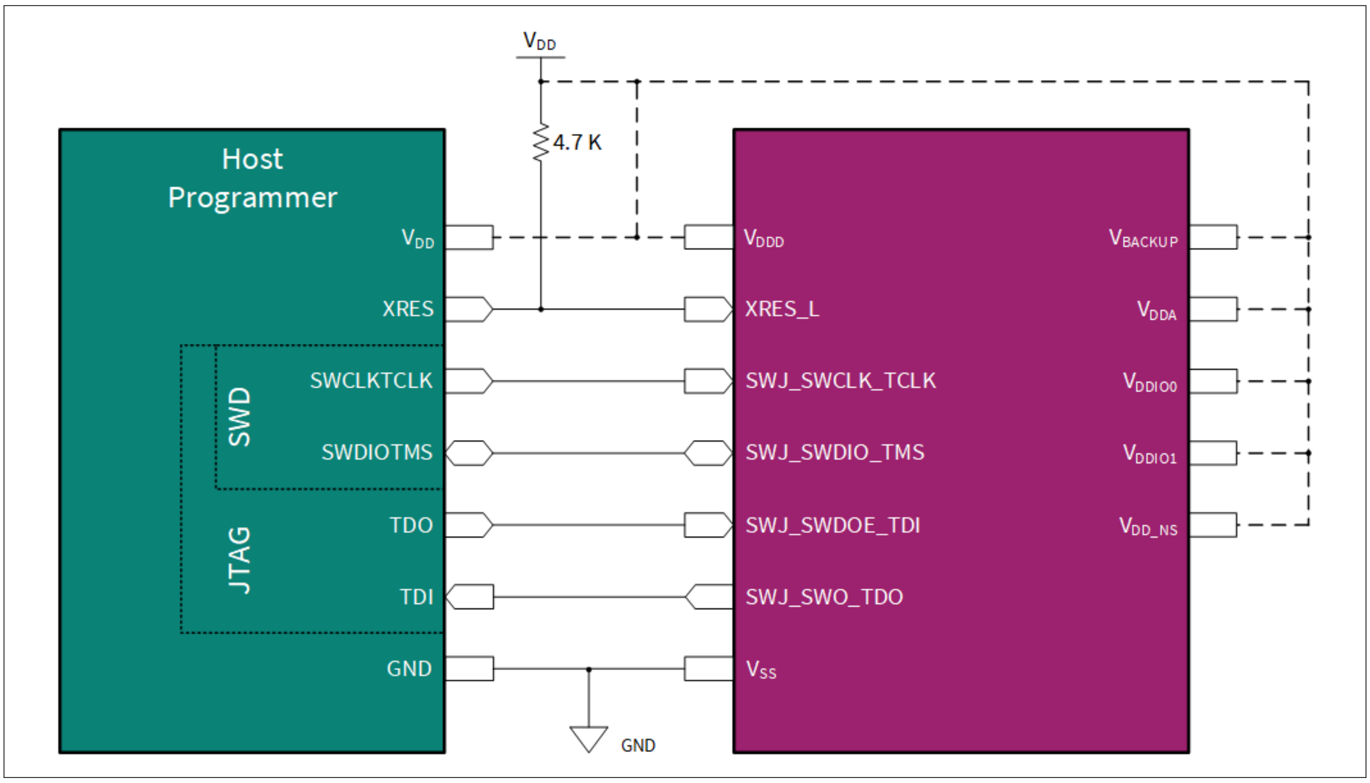


Figure 7 **Connection schematic of programmer**

Table 7 Pins/signals description

Pin	SWD		JTAG		Description
	Signal name	**Mandatory**	**Signal name**	**Mandatory**	
SWJ_SWCLK_TCLK	SWCLK (Serial Wire Clock)	YES	TCLK (Test Clock)	YES	The 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 XMC5000 MCU family is 1.5 MHz. It is needed only in the first step to acquire the chip during the boot window. After that, the programming frequency can be as low as needed. For the SWD, the host should perform all Read or Write operations on the SWDIO line on the falling edge of the SWDCK. The XMC5000 MCU performs the Read or Write operations on the SWDIO on the rising edge of the SWDCK. For the JTAG, the host writes to the TMS and TDI pins of the XMC5000 on the falling edge of the TCK and the XMC5000 MCU reads data on its TMS and TDI lines on the rising edge of the TCK. XMC5000 MCU writes to its TDO line on the falling edge of TCK and the host reads from the TDO line of the XMC5000 MCU on the rising edge of the TCK.
SWJ_SWDIO_TMS	SWDIO (Serial Wire Data Input/ Output)	YES	TMS (Test Mode Select)	YES	The SWDIO is a bidirectional data input/ output signal The TMS is the JTAG Test Mode Select signal sampled at the rising edge of the TCK to determine the next state.

(table continues...)

Table 7 (continued) Pins/signals description

Pin	SWD		JTAG		Description
	Signal name	Mandatory	Signal name	Mandatory	
SWJ_SWO_TDO	SWO (Serial Wire Output)	NO	TDO (Test Data Out)	YES	The SWO signal (also known as TRACESWO) is required for the Serial Wire Viewer (SWV) and not required for SWD programming. It provides real-time data trace information from the XMC5000 MCU via the SWO pin, while the CPU continues running at full speed. The data trace via the 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.
SWJ_SWDOE_TDI	-	-	TDI (Test Data In)	YES	The TDI signal represents data shifted into the device's test or programming logic. It is sampled at the rising edge of the TCK.
XRES_L [1)]	XRES_L (External Reset)	NO	XRES_L (Reset)	NO	The external reset active LOW signal. The XRES_L is not related to the Arm® standard. It is used to reset the part as a first step in a programming flow. ***Note**: XRES_L pin/signal is not TRST (Test Reset) signal for the JTAG Interface which is the optional pin that asynchronously resets only the JTAG test logic.*
GND	GND (Ground)	YES	GND (Ground)	YES	Negative supply voltage (Ground).
VDDD [2)]	VDD	NO	VDD	NO	Positive supply voltage. The XMC5000 MCU can be powered by external power supply or by a 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.

1 The XRES_L pin is mandatory for Reset XMC5000 MCU acquisition mode, but not used for Power Cycle mode.

2 The VDDD pin is mandatory for Power Cycle XMC5000 MCU acquisition mode, where the programmer powers the XMC5000 MCU and external power is not applied. For Reset acquisition mode, the source of the power supplier does not matter, so the pin is optional.

- **Reset mode**: To start programming, the host toggles the XRES_L line and then sends SWD/JTAG commands (see Table 5). The power on the XMC5000 MCU board can be supplied by the host or by an external power adapter (the VDDD line can be optional).
- **Power cycle mode**: To start programming, the host powers on the XMC5000 MCU and then starts sending the SW/JTAG commands. The XRES_L line is not used.

The programmer should implement XMC5000 MCU acquisition in Reset mode. It is also the only way to acquire the XMC5000 MCU if the board consumes too much current, which the programmer cannot supply. Power Cycle mode support is optional and should be used only if: a) the XRES_L pin is not available on the part's package; b) the third-party programmer does not implement the XRES_L line, but can supply power to the XMC5000 MCU.

5 Programming algorithm

This chapter describes in detail the programming flow of the XMC5000. It starts with a high-level description of the algorithm and then describes every step using pseudocode. All code is based on the 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 5).

5.1 High-level programming flow

Figure 8 shows the sequence of steps that must be executed to program the 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 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 fails, then programming must be aborted.

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

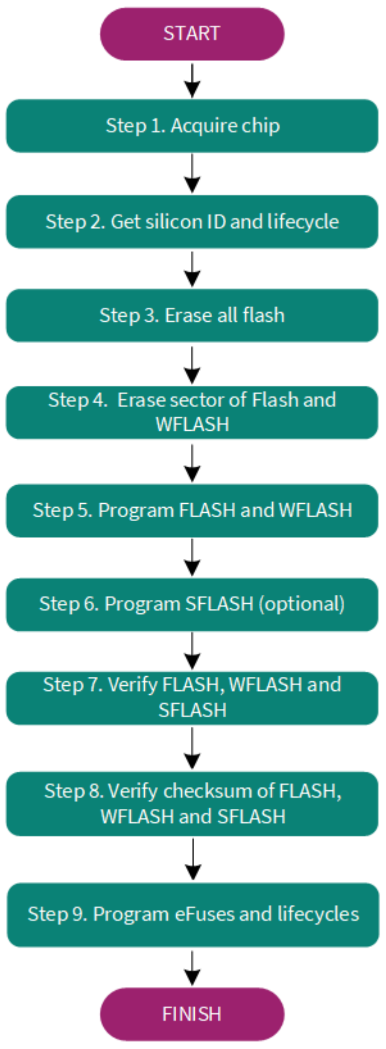

START
Step 1. Acquire chip
Step 2. Get silicon ID and lifecycle
Step 3. Erase all flash
Step 4. Erase sector of Flash and WFLASH
Step 5. Program FLASH and WFLASH
Step 6. Program SFLASH (optional)
Step 7. Verify FLASH, WFLASH and SFLASH
Step 8. Verify checksum of FLASH, WFLASH and SFLASH
Step 9. Program eFuses and lifecycles
FINISH

Figure 8 High-level programming flow of XMC5000 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. Table 8 lists the constants used in the programming scripts.

Table 8 Constants used in programming scripts

Name	XMC5100/XMC5200/XMC5300	Comment
MEM_BASE_SRAM	0x08000000	Base address of SRAM
MEM_BASE_FLASH	0x10000000	Base address of Main Flash region
MEM_BASE_WFLASH	0x14000000	Base address of Work Flash region
MEM_BASE_SFLASH	0x17000000	Base address of Supervisory Flash region
MEM_BASE_IPC	0x40220000	Base address for IPC structures
IPC_INTR_STRUCT_SIZE	0x20	
IPC_STRUCT_SIZE	0x20	
IPC_STRUCT0	0x40220000	CM0+ address
IPC_STRUCT1	0x40220020	CM4 address
IPC_STRUCT2	0x40220040	DAP address
IPC_INTR_STRUCT	0x40221000	IPC_INTR structure addr
IPC_STRUCT_ACQUIRE_OFFSET	0x00	Used to acquire a lock
IPC_STRUCT_NOTIFY_OFFSET	0x08	Used for Notification events
IPC_STRUCT_LOCK_STATUS_OFFSET	0x1C	IPC lock status
IPC_STRUCT_DATA0_OFFSET	0x0C	32-bit data element
IPC_STRUCT_DATA1_OFFSET	0x10	32-bit data element
IPC_STRUCT_LOCK_STATUS_ACQUIRED_MSK	0x80000000	Is lock acquired
IPC_STRUCT_ACQUIRE_SUCCESS_MSK	0x80000000	Is acquired
MXS40_SROMAPI_DATA_LOCATION_MSK	0x00000001	
MXS40_SROMAPI_STATUS_MSK	0xF0000000	
MXS40_SROMAPI_STAT_SUCCESS	0xA0000000	
MXS40_SROMAPI_SILID_CODE	0x00000001	SiliconId API code
MXS40_SROMAPI_WRITEROW_CODE	0x05000100	WriteRow API code
MXS40_SROMAPI_PROGRAMROW_CODE	0x06000100	ProgramRow API code
MXS40_SROMAPI_ERASEALL_CODE	0x0A000001	EraseAll API code
MXS40_SROMAPI_CHECKSUM_CODE	0x0B000001	Checksum API code
MXS40_SROMAPI_CHECKSUM_DATA_MSK	0x0FFFFFFF	Checksum mask
MXS40_SROMAPI_BLOW_FUSE_CODE	0x01000001	BlowFuse API code

(table continues...)

Table 8 (continued) Constants used in programming scripts

Name	XMC5100/XMC5200/XMC5300	Comment
MXS40_SROMAPI_READ_FUSE_CODE	0x03000001	ReadFuse API code
MXS40_SROMAPI_GENERATE_HASH_CODE	0x1E000000	GenerateHASH API code
MXS40_SROMAPI_CHECK_FACTORY_HASH_CODE	0x27000001	CheckFactoryHASH API code
ARM_DHCSR	0xE000EDF0	Debug Halting Control and Status Register
ARM_DHCSR_S_HALT_MSK	0x00020000	S_HALT
ARM_DHCSR_S_LOCKUP_MSK	0x00080000	S_LOCKUP
SRAM_SCRATCH_ADDR	0x08003000	
SRSS_TST_MODE	0x40261100	TST_MODE register
SRSS_TST_MODE_TEST_MODE_MSK	0x80000000	TEST_MODE_Mask

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

Table 9 Subroutines used in programming flow

Subroutine	Description
bool WriteIO (addr32, data32)	Writes a 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 6). By default, all accesses are 32-bit long. Returns true if all SWDJ transactions succeeded (ACKed).
bool Ipc_PollLockStatus (ipcId, isLockExpected)	Depending on the isLockExpected parameter, waits until the LOCK status bit of the IPC structure is released or acquired. The ipcId input parameter determines the number of the IPC structure (ipcId = 0 : CM0+ IPC_STRUCT; ipcId = 1 : CM4 IPC_STRUCT; ipcId = 2 : DAP IPC_STRUCT). Timeout is 1 second. Returns true if the LOCK status bit corresponds to the desired status; otherwise, returns false.
bool Ipc_Acquire(ipcId)	Acquires the IPC structure. The timeout is 1 second. Returns true if the IPC structure is acquired; otherwise, returns false.

(table continues...)

Table 9 (continued) Subroutines used in programming flow

Subroutine	Description
bool PollSromApiStatus (addr32, OUT data32)	Waits until the SROM command is completed and then checks its status. addr32 is the address where the SROM API status word is expected (IPC_STRUCT.DATA field or address in RAM if parameters for SROM API are passed in RAM). The timeout is 1 second. Output parameter data32 is the status/result word provided by the SROM API. Returns true if the command is completed and its status is successful; otherwise, returns false.
bool CallSromApi(callIdAndParams, OUT data32)	Executes the SROM API. The input parameter is the API OpCode and parameters word. Output parameter data32 is the status/result word, provided by SROM API. Returns true if the SROM API is executed and returned the success status; otherwise, returns false.

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

```
//-------------------------------------------------------------------------------
// “WriteIO” Subroutine
bool WriteIO (addr32, data32)
{
    if (interface == “SWD”)
        Ack_OK = 3b’001;
    else // JTAG
        Ack_OK = 3b’010;

    Ack1 = Write_DAP (TAR, addr32);
    Ack2 = Write_DAP (DRW, data32);
    return (Ack1 == Ack_OK) && (Ack2 == Ack_OK);
}

//-------------------------------------------------------------------------------
// “ReadIO” Subroutine
bool ReadIO (addr32, OUT data32)
{
    if (interface == “SWD”)
        Ack_OK = 3b’001;
    else // JTAG
        Ack_OK = 3b’010;

    Ack1 = Write_DAP (TAR, addr32);
    Ack2 = Read_DAP (DRW, OUT data32);
    Ack3 = Read_DAP (DRW, OUT data32);
    return (Ack1 == Ack_OK) && (Ack2 == Ack_OK) && (Ack3 == Ack_OK);
}

//-------------------------------------------------------------------------------
// “Ipc_PollLockStatus” Subroutine
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))
  if (TimeElapsed >= 1 sec) return false; // timeout

  return true;
}

//-------------------------------------------------------------------------------
// “Ipc_Acquire” Subroutine
bool Ipc_Acquire (ipcId)
```

```
{
  IpcAddr = IPC_STRUCT0 + IPC_STRUCT_SIZE * ipcId;
  do
  {
    // To acquire the IPC[2] (IPC structure for DAP),
    // the debugger must write any value to the 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)
    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)
    // the master must read the IPC[x].ACQUIRE register
    // If the SUCCESS field returns 1, the Read acquired the lock.
    // If the SUCCESS field returns 0, the Read did not acquire the lock.
    // Note: 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))

  if (TimeElapsed >= 1 sec) return false; // timeout
  else return true;
}

//-------------------------------------------------------------------------------
// “PollSromApiStatus” Subroutine
bool PollSromApiStatus (addr32, OUT data32)
{
  do
  {
    ReadIO (addr32, OUT data32 );
    Status = data32 & MXS40_SROMAPI_STATUS_MSK;
  }
  while (Status != MXS40_SROMAPI_STAT_SUCCESS) && (TimeElapsed < 1 sec))
  if (TimeElapsed >= 1 sec) return false; // timeout

  return true;
}

//-------------------------------------------------------------------------------
// “CallSromApi” Subroutine
bool CallSromApi (callIdAndParams, OUT data32)
{
  // System Calls can be invoked either by the code running on the CPU core
  // or by the DAP directly. Each source of System Calls has it’s dedicated
  // IPC structure. For instance, on XMC5000  chips the code running on CM4
  // core must use IPC structure #1, System Calls invoked by the DAP must
  // use IPC#2. The mapping of IPC structures is shown in
  // the table below:
  //
  //              | CM0+ | CM4 | DAP |
  // -------------------------------|
  // XMC5000      |  0   |  1  |  2  |
  //
  // In this example IPC #2 Is used.

  IpcId  = 2;
  IpcAddr = IPC_STRUCT0 + IPC_STRUCT_SIZE * IpcId;

  // Check where the arguments for the SROM API are located
  // [0]: 1 - arguments are passed in IPC.DATA. 0 - arguments are passed in SRAM
  IsDataInRam = (callIdAndParams & MXS40_SROMAPI_DATA_LOCATION_MSK) == 0;

  // Acquire IPC_STRUCT[IpcId]
  Status = Ipc_Acquire(IpcId);
  if (Status == false) return false;

  // Write one of these to IPC_STRUCT[IpcId].DATA:
  // a) SROM API 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_DATA0_OFFSET, SRAM_SCRATCH_ADDR);
  }
  else {
    WriteIO (IpcAddr + IPC_STRUCT_DATA0_OFFSET, callIdAndParams);
  }

  // Enable the notification interrupt of IPC_INTR_STRUCT[0](CM0+) for IPC_STRUCT[IpcId]
  WriteIO IPC_INTR_STRUCT + IPC_INTR_STRUCT_INTR_IPC_MASK_OFFSET, 1 << (16 + 2));

  // Notify to IPC_INTR_STRUCT[0]. IPC_STRUCT[IpcId].MASK <- Notify
  // This starts the SROM API execution
  WriteIO (IpcAddr + IPC_STRUCT_NOTIFY_OFFSET, 1 << 0 /*IPC_INTR_STRUCT0*/);

  // The poll lock status for the released state
  Status = Ipc_PollLockStatus (IpcId , false);
  if (Status == false) return false;

  // The poll data word
  if (IsDataInRam) {
    Status = PollSromApiStatus (SRAM_SCRATCH_ADDR, data32);
  }
  else {
    Status = PollSromApiStatus (IpcAddr + IPC_STRUCT_DATA0_OFFSET, data32);
  }

  return Status;
}
//-----------------------------------------------------------------------------
```

5.3 Step 1.A – acquire XMC5000 MCU

The first step in programming the XMC5000 MCU is to put it into Test mode (or Programming mode). This is special mode in which the CPU is controlled by an external programmer which can also access other system resources such as SRAM and registers. The main purpose of this step is to prevent execution of the user's code from the main 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. If there is corrupted user's code in the main 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 9 shows the timing diagram for entering Test mode.

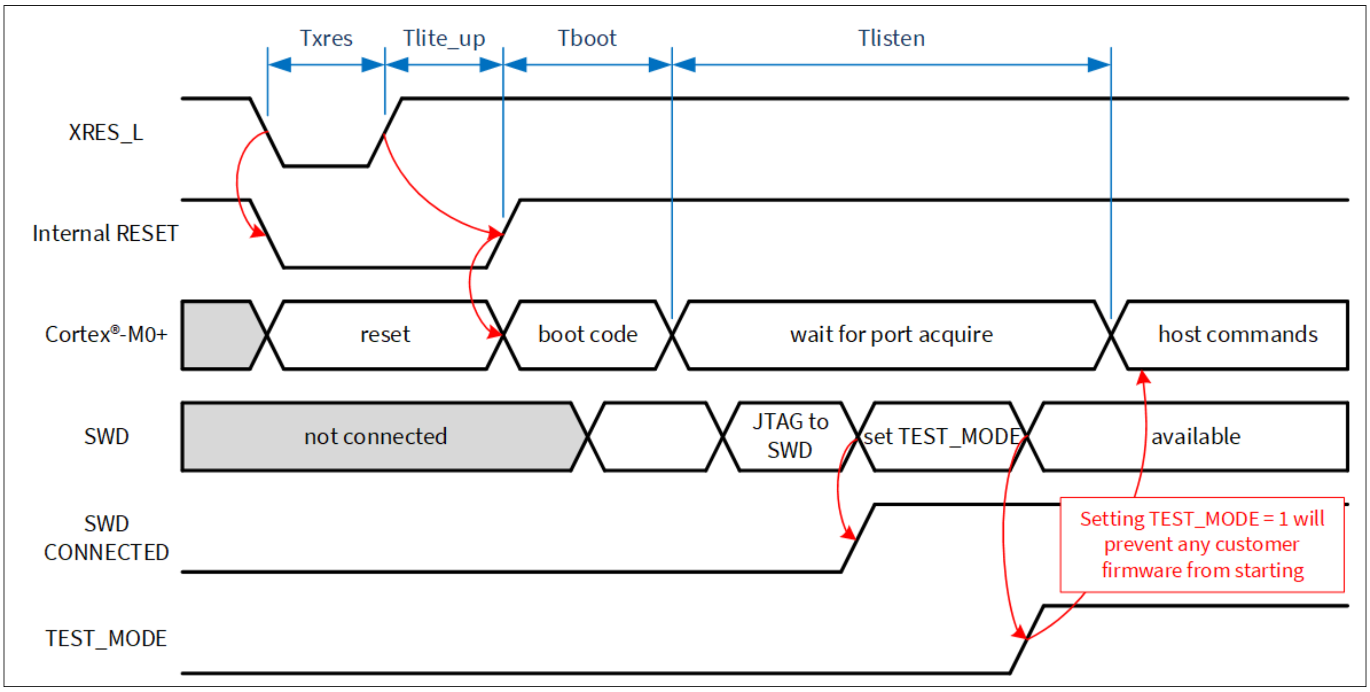


Figure 9 Timing diagram for entering test mode

This diagram details the chip's internal signals while entering Test mode. Everything starts from toggling the XRES_L line (or applying power), so the chip enters Internal Reset mode for a `Tlite_up` period. After that, the system boot code starts execution. When completed, the CPU waits during a `Tlisten` 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 main FLASH region. Timing parameters may vary depending on the boot code execution flow (see the corresponding Architecture TRM for boot timing parameters). Therefore, the best way to enter Test mode is to start sending an acquire sequence immediately after XRES_L 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).

Figure 10 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.

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

1. 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.
2. 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 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.

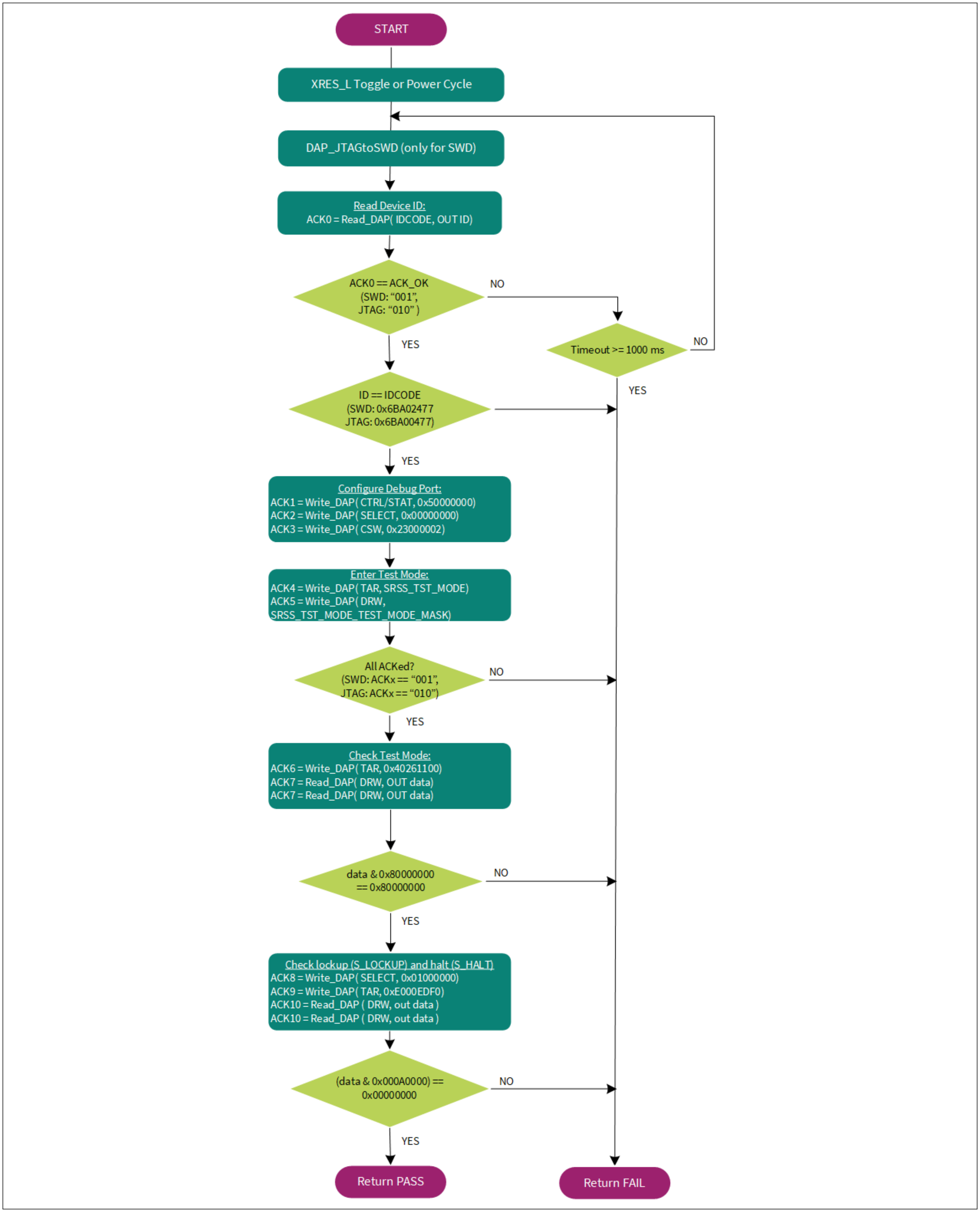

START
XRES_L Toggle or Power Cycle
DAP_JTAGtoSWD (only for SWD)
Read Device ID:
ACK0 = Read_DAP(IDCODE, OUT ID)
ACK0 == ACK_OK (SWD: "001", JTAG: "010")
NO
Timeout >= 1000 ms
NO
YES
YES
ID == IDCODE (SWD: 0x6BA02477 JTAG: 0x6BA00477)
YES
Configure Debug Port:
ACK1 = Write_DAP(CTRL/STAT, 0x50000000)
ACK2 = Write_DAP(SELECT, 0x00000000)
ACK3 = Write_DAP(CSW, 0x23000002)
Enter Test Mode:
ACK4 = Write_DAP(TAR, SRSS_TST_MODE)
ACK5 = Write_DAP(DRW, SRSS_TST_MODE_TEST_MODE_MASK)
All ACKed? (SWD: ACKx == "001", JTAG: ACKx == "010")
NO
YES
Check Test Mode:
ACK6 = Write_DAP(TAR, 0x40261100)
ACK7 = Read_DAP(DRW, OUT data)
ACK7 = Read_DAP(DRW, OUT data)
data & 0x80000000 == 0x80000000
NO
YES
Check lockup (S_LOCKUP) and halt (S_HALT)
ACK8 = Write_DAP(SELECT, 0x01000000)
ACK9 = Write_DAP(TAR, 0xE000EDF0)
ACK10 = Read_DAP (DRW, out data)
ACK10 = Read_DAP (DRW, out data)
(data & 0x000A0000) == 0x00000000
NO
YES
Return PASS
Return FAIL

Figure 10 **Flowchart of acquire chip step**

5.3.1 Pseudocode – Step 1.A: Acquire chip

```
//--------------------------------------------------------------------------------
// Reset the target depending on Acquire mode – Reset or Power Cycle
if (AcquireMode == “Reset”) ToggleXRES(); // Toggle the XRES_L pin, the target must be powered
else if (AcquireMode == “Power Cycle”) PowerOn();// Supply power to the target

if (Interface == “SWD”)
{
    Ack_OK = 3b’001;
    idcode = 0x6BA02477;
}
else // JTAG
{
    Ack_OK = 3b’010;
    idcode = 0x6BA00477;
}

// Execute the connection sequence – acquire a port.
// This is used as a handshake between the debugger and target device.
// After the target device replied to the request to read the IDCODE,
// it means that the device is already booted after a reset and ready to communicate.
do
{
    if (Interface == “SWD”)
        DAP_JTAGtoSWD();
    Ack = Read_DAP (IDCODE, OUT ID);
}while ((Ack != Ack_OK) && TimeElapsed < 1000 ms);

if ((ID != idcode)  || (TimeElapsed >= 1000 ms)) return FAIL;
// For PowerCycle, the timeout must be longer. For example: ~3000 ms.

// Power up the debug port using the next bits in the CTRL/STAT register:
// [30]:CSYSPWRUPREQ and [28]:CDBGPWRUPREQ - power-up requests.
// [5]:STICKYERR, [4]:STICKYCMP and [1]:STICKYORUN – sticky errors bits
// Note: for the JTAG, sticky error bits are Read/Write enabled and writing 1
// to these bits clears the associated sticky errors.
// For the SWD, these bits are Read-only and to clean the sticky errors,
// write to the appropriate bits of the DP.ABORT register
if (Interface == “SWD”)
{
    Write_DAP (CTRL/STAT, 0x50000000);
}
else // JTAG
{
    Write_DAP (CTRL/STAT, 0x50000032);
}
Write_DAP (SELECT, 0x00000000); // Select SYSAP (APSEL=0x00)
Write_DAP (CSW, 0x23000002); // Set 32-bit transfer mode

// Enter the CPU into Test Mode
// Set the TEST_MODE bit in the TEST_MODE reg from the CPU space
WriteIO (SRSS_TST_MODE, SRSS_TST_MODE_TEST_MODE_MSK);
```

```
ReadIO  (SRSS_TST_MODE, OUT DataOut);
if ((DataOut & SRSS_TST_MODE_TEST_MODE_MSK) == 0) return FAIL;

// The steps above are time critical and must be executed
// without a delay immediately after a reset.
// The steps below are to ensure that the target was successfully acquired.
// These steps are not time critical.

// Select CM0AP (APSEL=0x01) to check whether the core is not in the LOCKUP state
// and for further operations with flash
Write_DAP (SELECT, 0x01000000);

// Check the lockup (S_LOCKUP == 0) and halt (S_HALT == 0)
ReadIO (ARM_DHCSR, OUT DataOut);
if ((DataOut & ARM_DHCSR_S_LOCKUP_MSK) != 0
    || (DataOut & ARM_DHCSR_S_HALT_MSK) != 0) return FAIL;

return PASS
//-----------------------------------------------------------------------------
```

5.4 Step 1.B – Acquire XMC5000 MCU (alternate method)

The Acquire Chip sequence in the previous section is based on entering the XMC5000 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_L pin or the power supply to the device. Programming by entering Test mode using XRES_L 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 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_L pin or the power supply to the XMC5000 MCU. Only the SWJ protocol pins are available for programming.
- The host programmer software application is unable to meet the timing requirements to enter XMC5000 MCU Test mode after triggering a hard-reset condition. In such a scenario, the MCU enters the user’s 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 the XRES_L/power supply toggling, and 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 XMC5000 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 the 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 XMC5000 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 11 shows the acquire chip (alternate method) procedure.

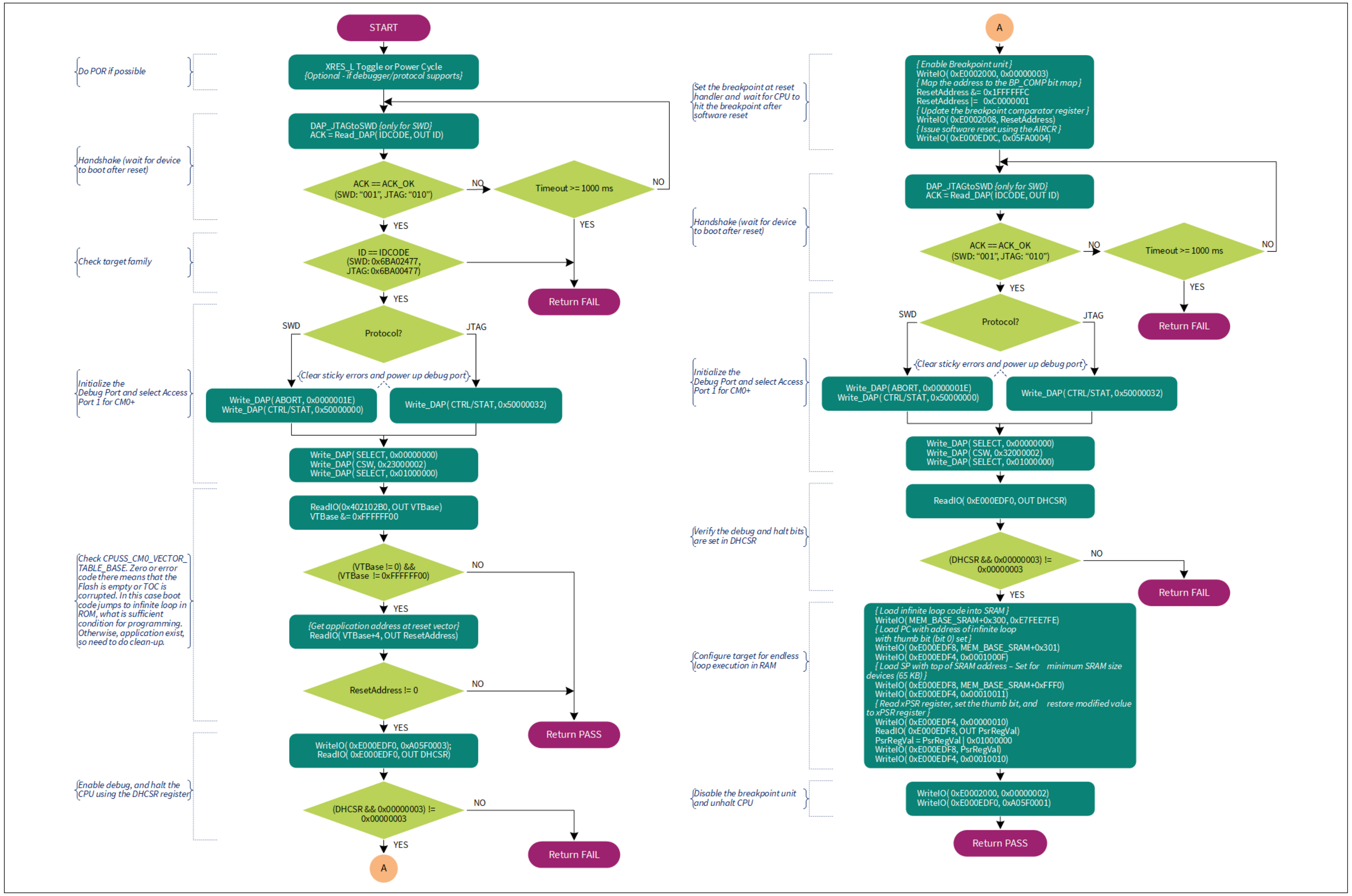

START
XRES_L Toggle or Power Cycle
{Optional - if debugger/protocol supports}
DAP_JTAGtoSWD {only for SWD}
ACK = Read_DAP(IDCODE, OUT ID)
ACK == ACK_OK
(SWD: "001", JTAG: "010")
NO
Timeout >= 1000 ms
NO
YES
YES
ID == IDCODE
(SWD: 0x6BA02477,
JTAG: 0x6BA00477)
Return FAIL
YES
SWD
Protocol?
JTAG
{Clear sticky errors and power up debug port}
Write_DAP(ABORT, 0x0000001E)
Write_DAP(CTRL/STAT, 0x50000000)
Write_DAP(CTRL/STAT, 0x50000032)
Write_DAP(SELECT, 0x00000000)
Write_DAP(CSW, 0x23000002)
Write_DAP(SELECT, 0x01000000)
ReadIO(0x402102B0, OUT VTBase)
VTBase &= 0xFFFFFF00
(VTBase != 0) &&
(VTBase != 0xFFFFFF00)
NO
YES
{Get application address at reset vector}
ReadIO(VTBase+4, OUT ResetAddress)
ResetAddress != 0
NO
Return PASS
YES
WriteIO(0xE000EDF0, 0xA05F0003);
ReadIO(0xE000EDF0, OUT DHCSR)
(DHCSR && 0x00000003) !=
0x00000003
NO
Return FAIL
YES
A
{Do POR if possible}
{Handshake (wait for device to boot after reset)}
{Check target family}
{Initialize the Debug Port and select Access Port 1 for CM0+}
{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 exist, so need to do clean-up.}
{Enable debug, and halt the CPU using the DHCSR register}
A
{ 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)
DAP_JTAGtoSWD {only for SWD}
ACK = Read_DAP(IDCODE, OUT ID)
ACK == ACK_OK
(SWD: "001", JTAG: "010")
NO
Timeout >= 1000 ms
NO
YES
YES
Return FAIL
SWD
Protocol?
JTAG
{Clear sticky errors and power up debug port}
Write_DAP(ABORT, 0x0000001E)
Write_DAP(CTRL/STAT, 0x50000000)
Write_DAP(CTRL/STAT, 0x50000032)
Write_DAP(SELECT, 0x00000000)
Write_DAP(CSW, 0x32000002)
Write_DAP(SELECT, 0x01000000)
ReadIO(0xE000EDF0, OUT DHCSR)
(DHCSR && 0x00000003) !=
0x00000003
NO
Return FAIL
YES
{ Load infinite loop code into SRAM }
WriteIO(MEM_BASE_SRAM+0x300, 0xE7FEE7FE)
{ Load PC with address of infinite loop
with thumb bit (bit 0) set }
WriteIO(0xE000EDF8, MEM_BASE_SRAM+0x301)
WriteIO(0xE000EDF4, 0x0001000F)
{ Load SP with top of SRAM address – Set for minimum SRAM size
devices (65 KB) }
WriteIO(0xE000EDF8, MEM_BASE_SRAM+0xFFF0)
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)
WriteIO(0xE0002000, 0x00000002)
WriteIO(0xE000EDF0, 0xA05F0001)
Return PASS
{Set the breakpoint at reset handler and wait for CPU to hit the breakpoint after software reset}
{Handshake (wait for device to boot after reset)}
{Initialize the Debug Port and select Access Port 1 for CM0+}
{Verify the debug and halt bits are set in DHCSR}
{Configure target for endless loop execution in RAM}
{Disable the breakpoint unit and unhalt CPU}

Figure 11 **Flowchart of acquire chip step (alternate method)**

5.4.1 Pseudocode – Step 1.B: Acquire chip (alternate method)

```
if (interface == “SWD”)
{
    Ack_OK = 3b’001;
    idcode = 0x6BA02477;else // JTAG
}
else // JTAG
    Ack_OK = 3b’010;
    idcode = 0x6BA00477;
}

//------------------------------------------------------------------------------
// Execute the SWJ connect sequence.
// 1000 ms time-out is the worst case for a secure application.
do {
    if (Interface == “SWD”)
        DAP_JTAGtoSWD();
    Ack = Read_DAP(IDCODE, OUT ID);
}while((Ack != Ack_OK) && TimeElapsed < 1000 ms);

if ((ID != idcode)  || (TimeElapsed >= 1000 ms)) return FAIL;

// Power up the debug port by setting bits [30]:CSYSPWRUPREQ
// and [28]:CDBGPWRUPREQ in the CTRL/STAT register.
// Also, clear previous errors if any previous firmware upgrade
// operation was aborted, or invalid transaction executed.
if (Interface == “SWD”)
{
    // Clear previous errors: write to the AP ABORT register in Debug Port
    // (APnDP bit – 0, Address is 2’b00, Access - W for the AP ABORT register
    Write_DAP (ABORT, 0x0000001E);
    // Power up debug port
    Write_DAP (CTRL/STAT, 0x50000000);
}
else // JTAG
{
    // Power up the debug port and clear previous errors by writing bits
    // [5]:STICKYERR, [4]:STICKYCMP and [1]:STICKYORUN
    // Note: for the JTAG, sticky error bits are Read/Write enabled and writing 1
    // to these bits clears the associated sticky errors.
    // For the SWD, these bits are Read-only and to clean the sticky errors,
    // write to the appropriate bits of the DP.ABORT register
    Write_DAP (CTRL/STAT, 0x50000032);
}
Write_DAP (SELECT, 0x00000000); // Select SYSAP (APSEL=0x00)
Write_DAP (CSW, 0x23000002);    // Set 32-bit transfer mode
Write_DAP (SELECT, 0x01000000); // Select CM0AP (APSEL=0x01)

// Get an address at the reset vector
// If flash is empty (address at default reset vector == 0), then the boot code
// jumps to an infinite loop in ROM. That is the sufficient condition for programming
ReadIO (0x10000004, OUT ResetAddress);
```

```
if (ResetAddress == 0) return PASS;

// Enable the debug, and halt the CPU using the DHCSR register
WriteIO (0xE000EDF0, 0xA05F0003);
// Verify the debug enable, cpu halt bits are set
ReadIO (0xE000EDF0, OUT Status);
if((Status & 0x00000003) != 0x00000003) return FAIL;

// Enable the Breakpoint unit using the BP_CTRL (Breakpoint Control Register)
// Set bits [0]: ENABLE =1, [1]: KEY=1
WriteIO (0xE0002000, 0x00000003);
// Map the address bits to the breakpoint compare register
// bit map, set the enable breakpoint bit, and the match bits
ResetAddress = (ResetAddress & 0x1FFFFFFC) | 0xC0000001;
//Update the breakpoint compare register
WriteIO (0xE0002008, ResetAddress);
// Issue a software reset
// using the AIRCR (Application Interrupt and Reset Control Register)
WriteIO (0xE000ED0C, 0x05FA0004);

// Let the target do some initialization before polling for IDCODE
Delay(5 ms);

// Repeat a portion of the acquire sequence again
// No need to check the ID value again
do {
    if (Interface == “SWD”)
        DAP_JTAGtoSWD();
    Ack = Read_DAP(IDCODE, OUT ID);
}while((Ack != Ack_OK) && TimeElapsed < 1000 ms);

if (TimeElapsed >= 1000 ms) return FAIL;

// Power up the debug port by setting bits [30]:CSYSPWRUPREQ
// and [28]:CDBGPWRUPREQ in the CTRL/STAT register.
// Also, clear previous errors if any previous firmware upgrade
// operation was aborted, or invalid transaction executed.
if (Interface == “SWD”)
{
    // Clear previous errors: write to the AP ABORT register in Debug Port
    // (APnDP bit - 0, Address is 2’b00, Access - W for the AP ABORT register
    Write_DAP (ABORT, 0x0000001E);
    // Power up the debug port
    Write_DAP (CTRL/STAT, 0x50000000);
}
else // JTAG
{
    // Power up the debug port and clear previous errors by writing bits
    // [5]:STICKYERR, [4]:STICKYCMP and [1]:STICKYORUN
    // Note: for the JTAG, sticky error bits are Read/Write enabled and writing 1
    // to these bits clears the associated sticky errors.
    // For the SWD, these bits are Read-only and to clean the sticky errors,
    // write to the appropriate bits of the DP.ABORT register
    Write_DAP (CTRL/STAT, 0x50000032);
}
Write_DAP (SELECT, 0x00000000); // Select SYSAP (APSEL=0x00)
Write_DAP (CSW, 0x00000002); // Set 32-bit transfer mode
Write_DAP (SELECT, 0x01000000); // Select CM0AP (APSEL=0x01)

// Verify the debug enable, cpu halt bits are set in the DHCSR register
ReadIO (0xE000EDF0, OUT Status);
if((Status & 0x00000003) != 0x00000003) return FAIL;

// Load infinite for loop code in SRAM address MEM_BASE_SRAM + 0x300
WriteIO (MEM_BASE_SRAM + 0x300, 0xE7FEE7FE);
// Load the PC with the address of infinite for the loop SRAM address with the thumb bit (bit
0) set
WriteIO (0xE000EDF8, MEM_BASE_SRAM + 0x301);
WriteIO (0xE000EDF4, 0x0001000F);
// Load the SP with the top of the SRAM address - Set for the minimum SRAM size devices (65 KB
size)
WriteIO (0xE000EDF8, MEM_BASE_SRAM + 0xFFF0);
WriteIO (0xE000EDF4, 0x00010011);
//Read the xPSR register, set the thumb bit, and restore the modified value to the xPSR register
WriteIO (0xE000EDF4, 0x00000010);
ReadIO (0xE000EDF8, OUT PsrRegVal);
PsrRegVal = PsrRegVal | 0x01000000;
WriteIO (0xE000EDF8, PsrRegVal);
WriteIO (0xE000EDF4, 0x00010010);
// Disable the Breakpoint unit
WriteIO (0xE0002000, 0x00000002);
// Unhalt the CPU
WriteIO (0xE000EDF0, 0xA05F0001);

return PASS;
//----------------------------------------------------------------------------
```

5.5 Step 2 – Check silicon ID

This step is required to verify that the acquired XMC5000 MCU corresponds to the hex file. It reads the ID from the hex file and compares it with the ID obtained from the XMC5000 MCU, using the Silicon ID SROM API.

5.5.1 Pseudocode – Step 2: Check silicon ID

```
//------------------------------------------------------------------------------
// Read Silicon ID from the hex file, 4 bytes from address 0x9050 0002 (big endian):
// HexID[0] - Silicon ID Hi
// HexID[1] - Silicon ID Lo
// HexID[2] - Revision ID
// HexID[3] - Family ID
// HEX_ReadSiliconID() must be implemented.
HexID = HEX_ReadSiliconID();

// Read Silicon ID from the target using a SROM request
// Type 0: Get Family ID & Revision ID
Params1 = MXS40_SROMAPI_SILID_CODE + (0x0000FF00 & (0 << 8));
Status = CallSromApi(Params1, OUT DataOut0);
if(!Status) return FAIL;

// Type 1: Get Silicon ID and protection state
Params2 = MXS40_SROMAPI_SILID_CODE + (0x0000FF00 & (1 << 8));
Status = CallSromApi(Params2, OUT DataOut1);
if(!Status) return FAIL;

FamilyIdHi =    (DataOut0 &  0x0000FF00) >>  8;  // Family ID High
FamilyIdLo =    (DataOut0 &  0x000000FF) >>  0;  // Family ID Low
RevisionIdMaj = (DataOut0 &  0x00F00000) >>  20; // Rev ID Major
RevisionIdMin = (DataOut0 &  0x000F0000) >>  16; // Rev ID Major
SiliconIdHi =   (DataOut1 &  0x0000FF00) >>  8;  // Silicon ID High
SiliconIdLo =   (DataOut1 &  0x000000FF) >>  0;  // Silicon ID Low
ProtectState =  (DataOut1 &  0x000F0000) >>  16; // Protection state
LifeCycleStage =  (DataOut1 &  0x00F00000) >>  20; // Life cycle stage


// Compare Family ID and Silicon ID with the Read from the the HEX file
// Ignore Revision ID, it is not essential for the programming; there are many
if ((FamilyIdHi != HexID.FamilyIdHi) || (FamilyIdLo != HexID.FamilyIdLo) ||
     (SiliconIdHi != HexID.SiliconIdHi) || (SiliconIdLo != HexID.SiliconIdLo)
    return FAIL;

return PASS;
//------------------------------------------------------------------------------
```

5.6 Step 3 – Erase all flash

The FLASH must be erased before programming. This step erases all FLASH rows in the user's flash calling the Erase All SROM API.

5.6.1 Pseudocode – Step 3: Erase all flash

```
//----------------------------------------------------------------------------
Params = MXS40_SROMAPI_ERASEALL_CODE;
Status = CallSromApi(Params, OUT DataOut);
return Status;
//----------------------------------------------------------------------------
```

5.7 Step 4 – Erase sector of FLASH and WFLASH

The whole FLASH can be erased by the Erase All command. But sometimes it is necessary to erase only a few FLASH sectors. Besides, there is no Erase All command for the WFLASH. The Erase Sector SROM API resolves the issue.

5.7.1 Pseudocode – Step 4: Erase sector

```
//----------------------------------------------------------------------------
// 1. Prepare the EraseSector SROM API
// SRAM_SCRATCH: OpCode
WriteIO (SRAM_SCRATCH_ADDR , MXS40_SROMAPI_ERASESECTOR_CODE);

// SRAM_SCRATCH + 0x04: Any address inside of the sector to be erased
Params = 0x14000000; // The address of the first WFLASH sector
WriteIO (SRAM_SCRATCH_ADDR + 0x04 , Params);

// 2. Call the EraseSector SROM API
Status = CallSromApi(Params, OUT DataOut);
return Status;
//----------------------------------------------------------------------------
```

5.8 Step 5 – Program FLASH and WFLASH

The Flash memory is programmed in rows. The programmer must serially program each row individually. This step uses the Program Row SROM API. The API requires a row size as the input parameter. See Table 1 for the row sizes. See Table 10 for mapping the flash row size to the Program Row API input parameter value.

Table 10 Mapping row size to program row API parameter value

Flash region	Flash row size, bytes	Program API constant
Work Flash	4	2
Main Flash	8	3
Main Flash	32	5
Main Flash	512	9

Figure 12 illustrates this programming algorithm.

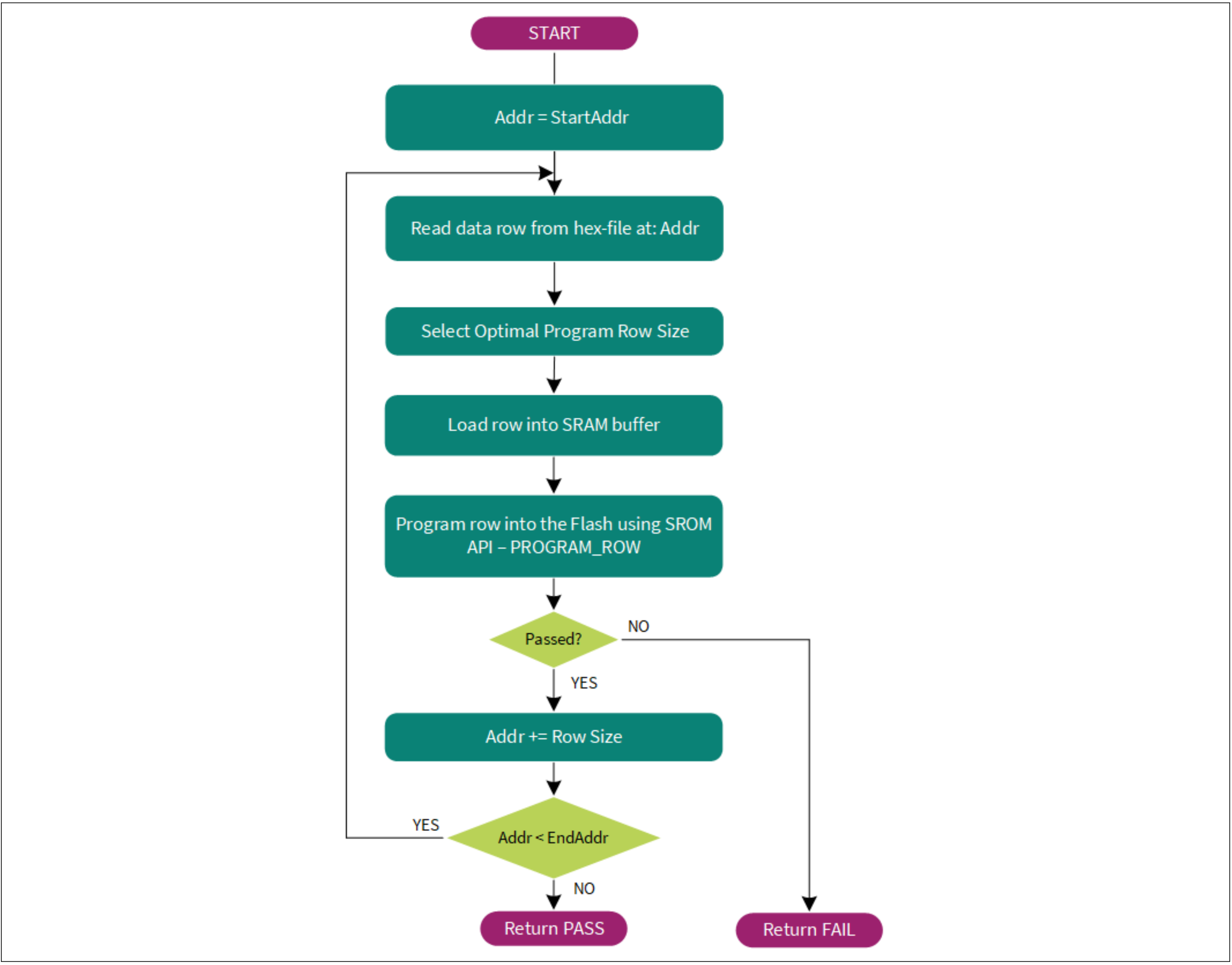

START
Addr = StartAddr
Read data row from hex-file at: Addr
Select Optimal Program Row Size
Load row into SRAM buffer
Program row into the Flash using SROM API – PROGRAM_ROW
Passed?
NO
YES
Addr += Row Size
YES
Addr < EndAddr
NO
Return PASS
Return FAIL

Figure 12 **Flowchart of the "program flash" step**

5.8.1 Pseudocode – Step 5: Program flash

```
//-----------------------------------------------------------------------------
// StartAddr and EndAddr are taken from the programming file
// and align them to the supported row size

// Program all data from the programming file

Addr = StartAddr;
while (Addr < EndAddr)
{
    // 1. Extract the row data from the hex-file from address
    // Addr and put into the buffer Data.
    // To speed up the programming, program the flash
    // with the maximum length of the row
    // HEX_ReadData() must be implemented by Programmer.
    Data = HEX_ReadData(Addr, MaxRowSize);

    // 2. Select an optimal row size
    if (Data.size() >= MaxRowSize)
    {
        RowSize = MaxRowSize;
    }
    else
    {
    // If the programming file does not contain the requested data, decrease
    // RowSize if possible or fill the rest of the Read data by erasing the cell’s value
    }

    // 3. Prepare ProgramRow SROM API
    // SRAM_SCRATCH: OpCode
    WriteIO (SRAM_SCRATCH_ADDR , MXS40_SROMAPI_PROGRAMROW_CODE);

    // SRAM_SCRATCH + 0x04: Data location/size and Integrity check
    Params =
        (6 << 0) +       // Data size: take the value from Table 10. E.g. 6 for 512 bytes
        (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:
    // The flash address to be programmed (in 32-bit system address format)
    WriteIO (SRAM_SCRATCH_ADDR + 0x08 , Addr);

    // SRAM_SCRATCH + 0x0C:
    // The pointer to the first data byte location
    DataRamAddr = SRAM_SCRATCH_ADDR + 0x10;
    WriteIO (SRAM_SCRATCH_ADDR + 0x0C, DataRamAddr);

    // Load the row bytes into the SRAM buffer
    for (i=0; i < RowSize || i < Data.size(); i+=4)
    {
        DataWord = (Data[i+3]<<24) +
                   (Data[i+2]<<16) +
                   (Data[i+1]<<8) +
                   (Data[i]<<0);
        WriteIO (DataRamAddr + i, DataWord);
    }

    // 4. Call the ProgramRow SROM API
    Status = CallSromApi(Params, OUT DataOut);
    if (!Status) return FAIL;
    Addr += RowSize;
}
return PASS;
//--------------------------------------------------------------------------
```

5.9 Step 6 – Program SFLASH (optional)

The SFLASH stores application-specific data such as calibration data, non-volatile parameters, and so on.

See Table 3 for the address ranges in the SFLASH region that can be programmed.

The SFLASH programming step is optional—every application determines the need for this flash region and its purpose. The SFLASH rows are not stored in the hex file by default, so if your workflow requires that the SFLASH data be in the hex file, the linker scripts must contain the 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 APIs). During mass production, a vendor should define the programming process—where to get the SFLASH data and at which row/address to store it.

Figure 13 illustrates the programming algorithm for the SFLASH region.

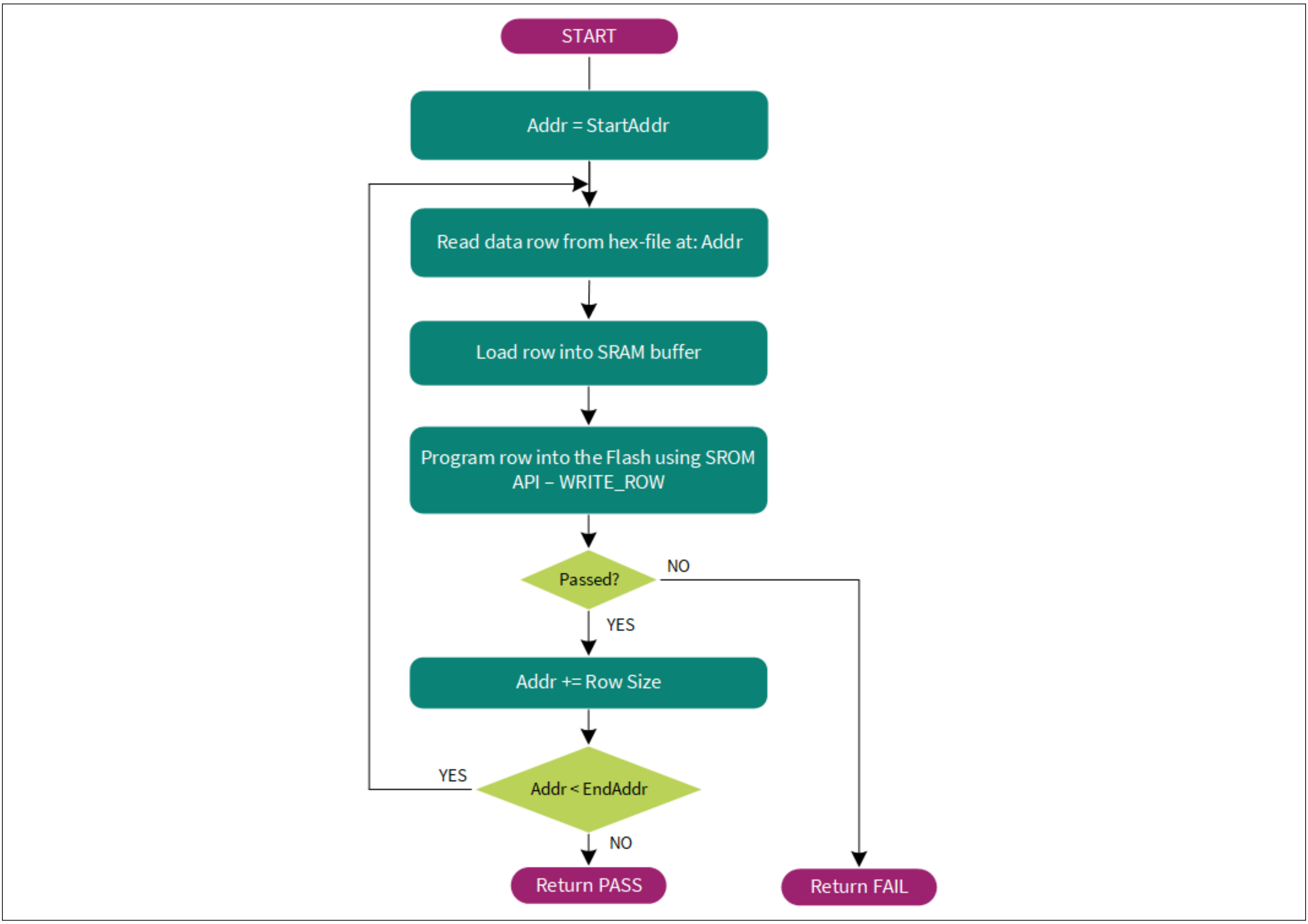

START
Addr = StartAddr
Read data row from hex-file at: Addr
Load row into SRAM buffer
Program row into the Flash using SROM API – WRITE_ROW
Passed?
NO
YES
Addr += Row Size
YES
Addr < EndAddr
NO
Return PASS
Return FAIL

Figure 13 **Flowchart of "Program SFLASH" step**

5.9.1 Pseudocode. Step 9: Program SFLASH

```
//------------------------------------------------------------------------------
// StartAddr and EndAddr are taken from the programming file
// and align them to the supported row size

// Program all data from the programming file

Addr = StartAddr;
while (Addr < EndAddr)
{
    // 1. Extract the row data from the hex-file from address
    // Addr and put into buffer Data.
    // HEX_ReadData() must be implemented by Programmer.
    Data = HEX_ReadData(Addr, RowSize);

    // 2. Prepare the WriteRow SROM API
    // SRAM_SCRATCH: OpCode
    WriteIO (SRAM_SCRATCH_ADDR , MXS40_SROMAPI_WRITEROW_CODE);

    // SRAM_SCRATCH + 0x04: Data location/size and Integrity check
    Params =
        (1 << 16);   // Verify row: 0 - Data integrity check is performed
    WriteIO (SRAM_SCRATCH_ADDR + 0x04 , Params);

    // SRAM_SCRATCH + 0x08:
    // The flash address to be programmed (in 32-bit system address format)
    WriteIO (SRAM_SCRATCH_ADDR + 0x08 , Addr);

    // SRAM_SCRATCH + 0x0C:
    // The pointer to the first data byte location
    DataRamAddr = SRAM_SCRATCH_ADDR + 0x10;
    WriteIO (SRAM_SCRATCH_ADDR + 0x0C, DataRamAddr);

    // Load the row bytes into the SRAM buffer
    for (i=0; i < RowSize || i < Data.size(); i+=4)
    {
        DataWord = (Data[i+3]<<24) +
                   (Data[i+2]<<16) +
                   (Data[i+1]<<8) +
                   (Data[i]<<0);
        WriteIO (DataRamAddr + i, DataWord);
    }

    // 4. Call the ProgramRow SROM API
    Status = CallSromApi(MXS40_SROMAPI_WRITEROW_CODE, OUT DataOut);
    if (!Status) return FAIL;
    Addr += RowSize;
}
return PASS;
//---------------------------------------------------------------------------
```

5.10 Step 7 – Verify FLASH, WFLASH, and SFLASH

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

During verification, the programmer reads a row from flash and the corresponding data from the hex 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, WFLASH, and SFLASH is achieved by direct access to the memory space of the CPU. No SROM API is required; simply read the word (32 bits) from the address range 0x10000000 to 0x100FFFFC, FlashSize – 4. For example:

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

Figure 14 illustrates the verification algorithm.

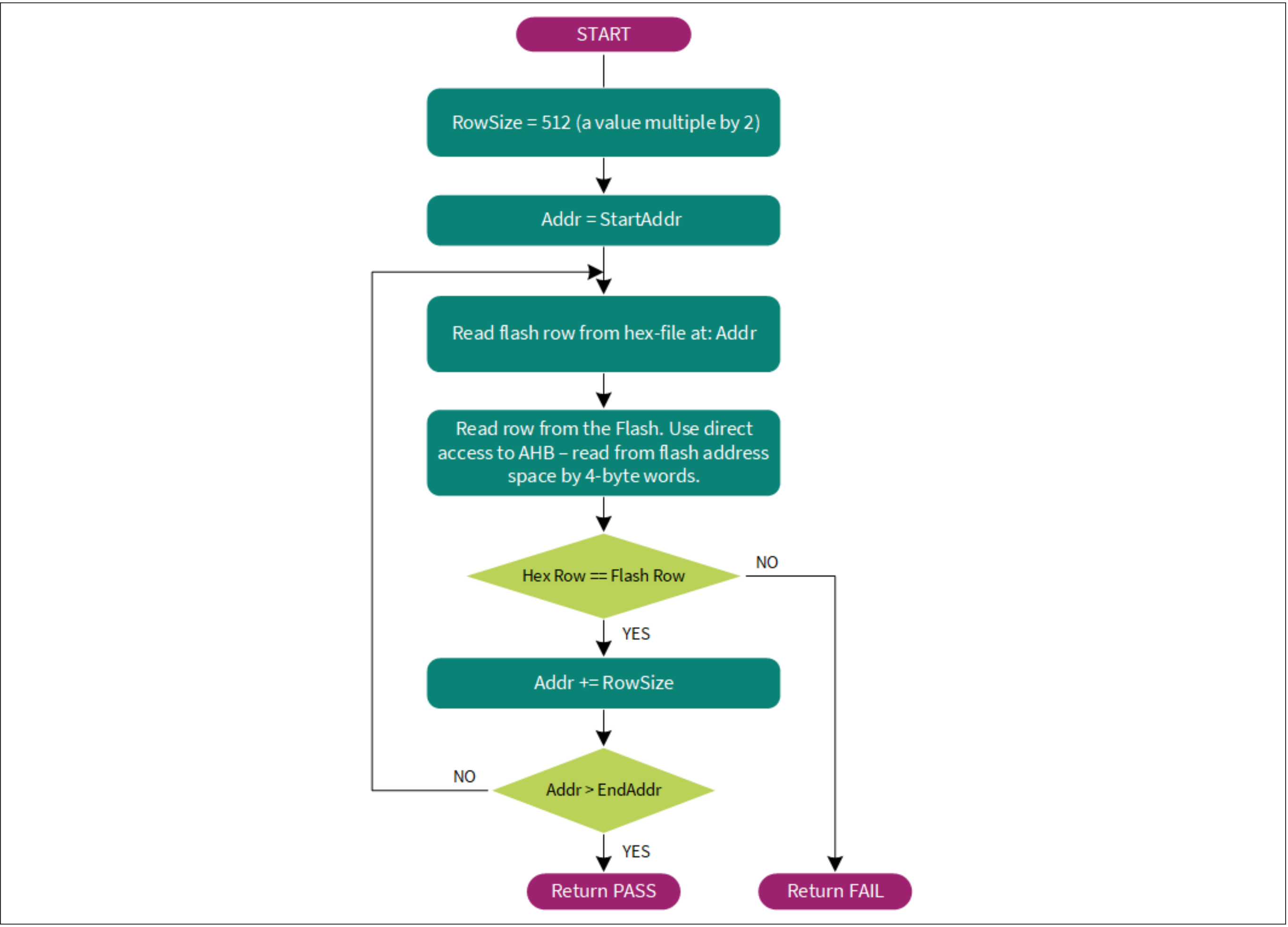

START
RowSize = 512 (a value multiple by 2)
Addr = StartAddr
Read flash row from hex-file at: Addr
Read row from the Flash. Use direct access to AHB – read from flash address space by 4-byte words.
Hex Row == Flash Row
NO
YES
Addr += RowSize
NO
Addr > EndAddr
YES
Return PASS
Return FAIL

Figure 14 **Flowchart of "Verify flash" step**

5.10.1 Pseudocode – Step 6: Verify flash

```
//---------------------------------------------------------------------------
// StartAddr and EndAddr are taken from the programming file
// and align them to the supported row size

// RowSize can be any value multiple of 2 and the flash size. Use 512 in this example
RowSize = 512;

//Read and Verify Flash rows
Addr = StartAddr;
while (Addr < EndAddr)
{
    //1. Read a row from the hex file
    //from address: Addr into the buffer Data.
    //HEX_ReadData() must be implemented by Programmer.
    HexData = HEX_ReadData(Addr, RowSize);

    //2. Read a row from the chip
    for (i = 0; i < RowSize; i += 4)
    {
        //Read flash via the AHB-interface
        ReadIO (Addr + 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 < RowSize; i++)
    {
        if (ChipData[i] != HexData[i]) return FAIL;
    }
}
return PASS;
//---------------------------------------------------------------------------
```

5.11 Step 8 – Verify checksum of FLASH, WFLASH, and SFLASH

This step validates the result of the flash programming process. It calculates the checksum of the user's rows written in Step 5 (Checksum SROM API is used) and compares this value with the 4-byte checksum from the hex file. The checksum operation cannot completely guarantee that the data is written correctly. For this reason, the Verify Flash step is also recommended.

5.11.1 Pseudocode – Step 7: Verify checksum

```
//-----------------------------------------------------------------------------
// Use the Checksum SROM API to get the checksum of the whole user's FLASH
// Byte 2 of the parameters select whether the checksum
// is performed on the whole FLASH, or a row of FLASH.
Params = MXS40_SROMAPI_CHECKSUM_CODE +
    (0 << 22) +         // Flash region: 0 – main , 1- work , 2 - supervisory
    (1 << 21));         // Whole FLASH: 0 – page , 1 – whole FLASH
Status = CallSromApi(Params, OUT DataOut);
if(!Status) return FAIL;

// Get checksum bits
IpcAddr = IPC_STRUCT0 + IPC_STRUCT_SIZE * IpcId;
ReadIO (IpcAddr + IPC_STRUCT_DATA1_OFFSET, OUT Checksum);

// Read 4-byte checksum of user code from hex-file
// HEX_ReadChecksum() must be implemented by Programmer.
Hex_Checksum = HEX_ReadChecksum();

// Compare silicon's vs hex's checksum
if (Checksum != Hex_Checksum) return FAIL;

return PASS;
//----------------------------------------------------------------------------
```

5.12 Step 9 – Program eFuses and lifecycles

The eFuse memory starts in address 32'h9070 0000 in the programming flow. The eFuse and lifecycle programming must be the last step in the programming flow. The table of contents must be programmed before this step. If any data is changed when the "secure" bit is blown, the XMC5000 MCU will go to the dead state. Figure 15 shows the sequence to program the eFuse and lifecycles.

See the "Device security" and "Protection units" sections of the technical reference manual (TRM) for more details of the security settings for the XMC5000 MCU.

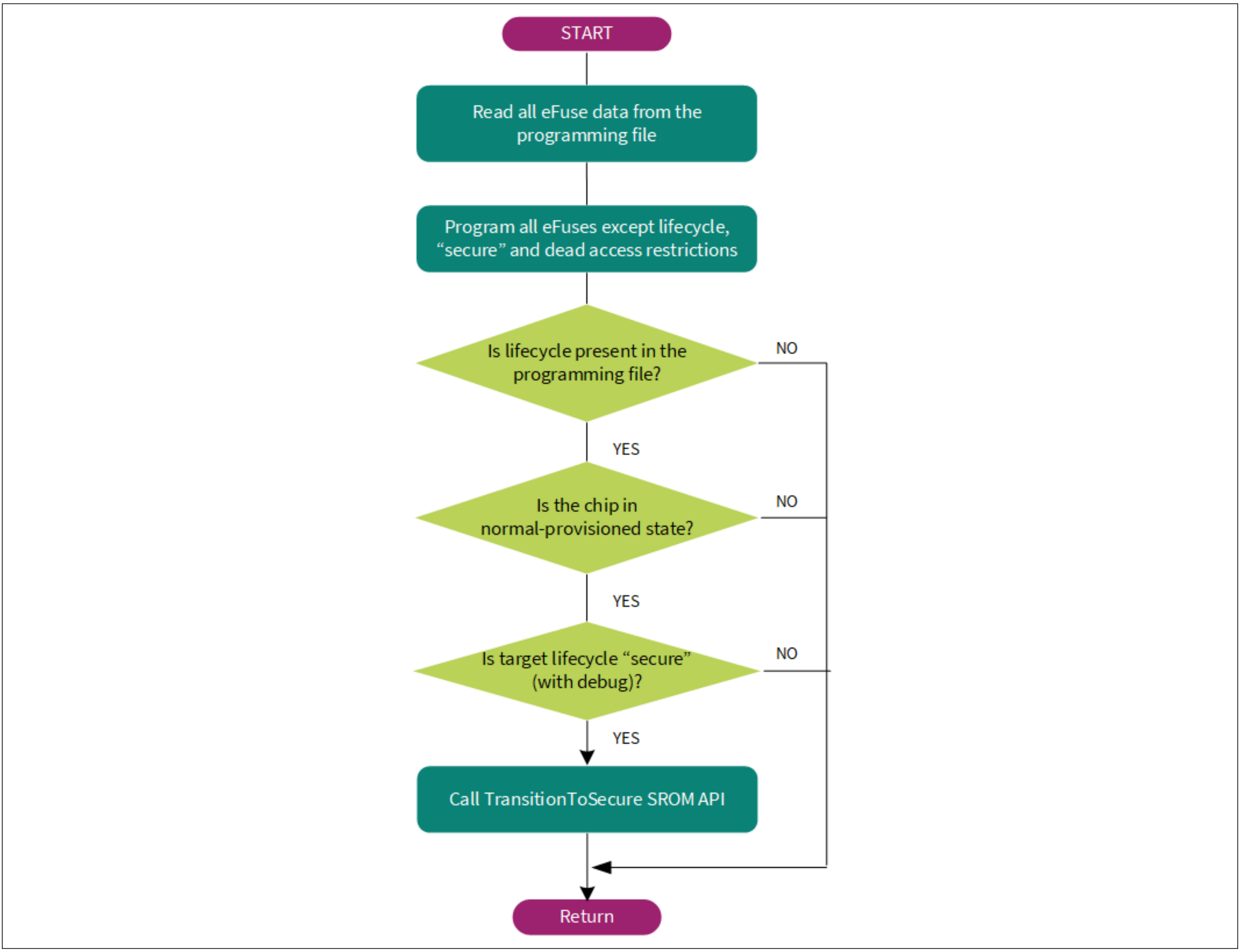

START
Read all eFuse data from the programming file
Program all eFuses except lifecycle, “secure” and dead access restrictions
Is lifecycle present in the programming file?
NO
YES
Is the chip in normal-provisioned state?
NO
YES
Is target lifecycle “secure” (with debug)?
NO
YES
Call TransitionToSecure SROM API
Return

Figure 15 **Flowchart of the “Program eFuses and lifecycles” step**

5.12.1 Pseudocode – Step 9: Program eFuses and lifecycles

```
//----------------------------------------------------------------------------
// ReadFuseBit
//
// Uses ReadFuseByte SROM API to read the eFuse byte and returns the specified bit.
//
// Input:
//  byteAddr – A byte offset in eFuse memory.
//  bitAddr – A bit address of the eFuse bit inside of the eFuse byte.
// Output:
//  bitValue – A read eFuse bit value.

// Reads the whole eFuse byte.
params = MXS40_SROMAPI_READ_FUSE_CODE + (byteAddr<<8);
CallSromApi(params, OUT byteValue);

// Returns the specific bit value.
return (byteValue & (1 << bitAddr)) != 0;

//----------------------------------------------------------------------------
// BlowFuseBit
//
// Uses BlowFuseBit SROM API to blow the eFuse bit.
//
// Input:
//  byteAddr – The byte offset in eFuse memory: 0-127.
//  bitAddr – The bit positions: 0-7.
// Output:
//  SUCCESS if the bit is blown, else - FAIL.

// Calculates a macro address.
// Updates the byte address (taking into account macro address).

if (1 == ReadFuseBit(byteAddr, bitAddr))
{
    WARNING("The efuse bit at address 0x%X[%X] has already been blown",
    byteAddr, bitAddr);
    return SUCCESS;
}

macroAddr = 0;
byte_Addr = byteAddr;
while (byte_Addr >= 32) // 32 – The eFuse macro size.
{
    Byte_Addr = byte_Addr - 32;
    macroAddr = macroAddr + 1;
}

params = MXS40_SROMAPI_BLOW_FUSE_CODE +     // BlowFuseBit SROM API code
    (0x00FF0000 & (byte_Addr<<16)) +        // The byte offset in the macro
    (0x0000F000 & (macroAddr<<12)) +        // Macro Address
    (0x00000700 & (bitAddr<<8));            // The bit position in the byte
```

```
CallSromApi(params, OUT dataOut);

if (0 == ReadFuseBit(byteAddr, bitAddr))
{
    ERROR("The efuse bit at address 0x%X[%X] was not blown",
    byteAddr, bitAddr);
    return FAIL;
}

//------------------------------------------------------------------------------
// Programs the eFuse region and performs a Life-Cycle transition.

#define DEAD_ACCESS_RESTR_OFFSET        0x40
#define SECURE_ACCESS_RESTR_OFFSET      0x3C

// Extracts all eFuse data from the programming file.
// NOTE Each eFuse bit is represented by a byte in the programming file.
   The programming file can contain 0x00, 0x01, and 0xFF.
   They correspond to the bit values: 0, 1, and “ignore” respectively.
   “Ignore” means that the value is not specified in the programming file.
   HEX_ReadEFuseData() must be implemented by the Programmer.
EFUSE_Data = HEX_ReadEFuseData();

// Programs all eFuse bits from the programming file.
Addr = 0;
while (Addr < 1024) // 128 eFuse bytes = 1024 eFuse bits.
{
   // Skips Secure Access Restriction, Dead Access Restrictions
   // and the Life-Cycle byte.

        byteAddr = Addr / 8;
        bitAddr = Addr % 8;

    if (EFUSE_Data[Addr] == 0)
    {
        if (ReadFuseBit(byteAddr, bitAddr))
        {
            ERROR("The eFuse bit at address 0x%X[%X] has been blown but didn’t
have to be", byteAddr, bitAddr);
            return FAIL;
        }
    }
    else
        BlowFuseBit(byteAddr, bitAddr);

    Addr++;
}

// Performs Life-Cycle transitions.
if (EFUSE_Data[1] & 0x0F)
    // Only the transition to Secure or Secure with Debug is supported.
{
    // Checks if the chip is in the Normal-Provisioned Life-Cycle.
    // LifeCycleStage is returned by SiliconID SROM API,
    // in Step 2 – “Get Silicon ID and Life-Cycle”, must be equal to 7.
    // Otherwise, exits the routine   .
    offset = DEAD_ACCESS_RESTR_OFFSET;
    deadAccess = EFUSE_Data[offset] | (EFUSE_Data[offset + 1] << 8)
        | (EFUSE_Data[offset+2] << 16)

    offset = SECURE_ACCESS_RESTR_OFFSET;
    secureAccess = EFUSE_Data[offset] | (EFUSE_Data[offset + 1] << 8)
        | (EFUSE_Data[offset+2] << 16)

    isDebug = (EFUSE_Data[1] & 0x0C) != 0;

    // Prepares the TransitionToSecure SROM API.
    // SRAM_SCRATCH: OpCode
    WriteIO (SRAM_SCRATCH_ADDR , MXS40_SROMAPI_TO_SECURE_CODE | isDebug);

    // SRAM_SCRATCH + 0x04: Secure Access Restriction
    WriteIO (SRAM_SCRATCH_ADDR + 0x04, secureAccess);

    // SRAM_SCRATCH + 0x04: Dead Access Restriction
    WriteIO (SRAM_SCRATCH_ADDR + 0x04, deadAccess);

    // Calls the TransitionToSecure SROM API.
    Status = CallSromApi(SRAM_SCRATCH_ADDR, OUT DataOut);
    if (!Status) 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 16.

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

Figure 16 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 – A data record with data and a 16-bit address.
  - 01 – The end of a 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 – An extended linear address record, 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 2_n hex digits.
- **Checksum** – Two hex digits (1 byte) which is the least significant byte of the two complements 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 example hex file are as follows: Consider that these three records form the hex file.

```
:020000043200C8
:041000001122334442
:00000001FF
```

The first record (`:020000043200C8`) 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 (`0x3200`) 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 `0x3200` (in other words, the base address is `0x32000000`). `0xC8` is the checksum byte for this record calculated as following:

```
0xC8 = 0x100 – (0x02+0x00+0x00+0x04+0x32+0x00).
```

Summary:

:020000043200C8

Address: $0000_{16} = 0_{10}$
Byte count: $02_{16} = 2_{10}$
Record type: 04_{16} = Extended Linear Address
Data: 3200_{32}
Checksum: $C8_{16}$

Calculated checksum: $C8_{16}$

The next record (`:0410000001122334442`) is a data record, as indicated by the value in the record type field (`00`). The byte count is 04, meaning there are four bytes in this record . The 32-bit starting address for these data bytes is at address `0x32001000`. The upper 16-bit address (`0x3200`) 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 `0x1000`. `0x42` is the checksum byte for this record.

Summary:

:0410000001122334442

Address: $1000_{16} = 4096_{10}$
Byte count: $04_{16} = 4_{10}$
Record type: 00_{16} = Data
Checksum: 42_{16}

Calculated checksum: 42_{16}

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.

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 (SWJ_SWDIO_TMS) and a clock connection (SWJ_SWCLK_TCLK). The host programmer always drives the clock line, while either the programmer or the XMC5000 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 XMC5000 MCU.
- **Acknowledge Response** – The XMC5000 MCU sends an acknowledgment to the host.
- **Data Transfer Phase** – The data transfer is either from the XMC5000 MCU to the host, following a read request (RDATA), or from the host to the XMC5000 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 17 shows the timing diagrams of the read and write SWD packets.

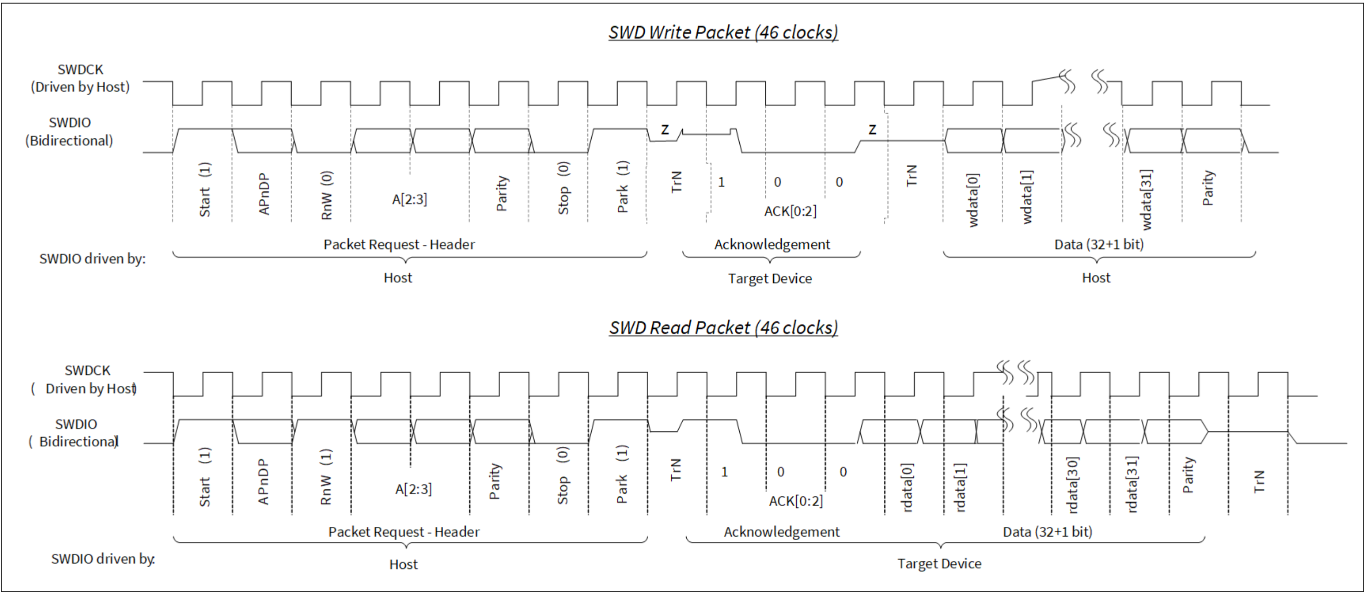


Figure 17 Write and read SWD packet timing diagrams

- Host Write Cycle – the Host sends data to the SWJ_SWDIO_TMS line on the falling edge of SWJ_SWCLK_TCLK and the target will read that data on the next SWJ_SWCLK_TCLK rising edge (for example, 8-bit header data).
- Host Read Cycle – the target sends data to the SWJ_SWDIO_TMS line on the rising edge of SWJ_SWCLK_TCLK and the Host reads that data on the next SWJ_SWCLK_TCLK falling edge (for example, ACK phase (ACK[2:0]), Read Data (rdata[31:0])).
- The Host should not drive the SWJ_SWDIO_TMS line during the TrN phase. During the first TrN phase (½ cycle duration) of the SWD packet, the target starts driving the ACK data to the SWDIO line on the rising edge of SWJ_SWCLK_TCLK . The Host reads the data on the subsequent falling edge of SWJ_SWCLK_TCLK. The second TrN phase is 1.5 clock cycles as shown in this figure. Both the target and Host will not drive the line during the entire second TrN phase (indicated as ‘z’). The Host starts sending the Write data (wdata) on the next falling edge of SWDCK after the second TrN phase.

The SWD packet is transmitted in this sequence:

1. The start bit initiates a transfer; it is always logical 1.
2. The APnDP bit determines whether the transfer is AP access (indicated by 1), or DP access (indicated by 0).
3. The next bit is RnW, which is 1 for a Read from the MCU or 0 for a Write to the MCU.
4. The ADDR bits (A[3:2]) are register select bits for the access port or debug port. See Table 6 for the register definition.

5. 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 incorrect, the XMC5000 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.
6. The stop bit is always logic 0.
7. The park bit is always logic 1 and should be driven high by the host.
8. The ACK bits are a device-to-host response. The possible values are shown in Table 11. Note that ACK in the current SWD transfer reflects the status of the previous transfer. OK ACK means that the previous packet was successful. The 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 XMC5000 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 XMC5000 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 XMC5000 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 11 ACK response for SWD transfers

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

9. 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 XMC5000 MCU detects a parity error in the data sent by the host, it generates a FAULT ACK response in the next packet.
10. The 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 Table 11. According to the SWD protocol, both the host and the XMC5000 MCU use the TrN phase to change the drive modes on their respective SWDIO lines. During the first TrN phase after packet request, the XMC5000 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 XMC5000 MCU should drive the SWDIO line during the TrN phase, as indicated by ‘z’ in Table 11.
11. The address, ACK, and Read and Write data are always transmitted LSB first.
12. 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 SWJ_SWDIO_TMS kept high. To return to the idle state, the SWJ_SWDIO_TMS must be clocked low once.*

8 Appendix C: Joint test action group (JTAG) protocol

The XMC5000 JTAG interface complies with the IEEE 1149.1 specification and provides additional instructions. There are two TAPs in the chip. 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 Table 12.

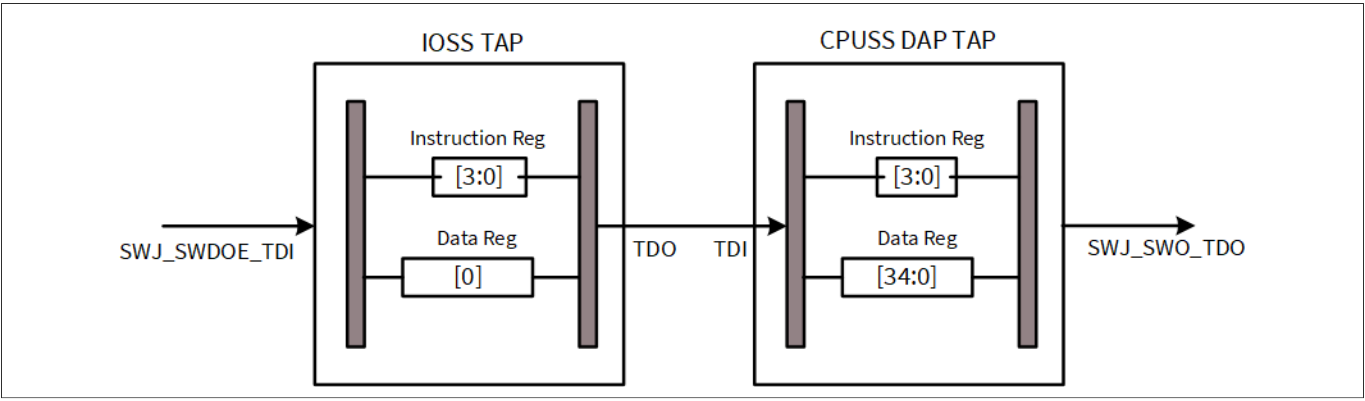


Figure 18 **IOSS/DAP TAP connection**

IOSS TAP is not used during device programming and must be put into BYPASS mode. This can be done by shifting the BYPASS instruction (0b1111) into the IOSS instruction register. The size of the Instruction Register in IOSS TAP is 4 bits. In Bypass mode, the data register is only 1-bit long with the contents of 0. Bypass mode is used to isolate the XMC5000 MCU TAP. See the example of the TAPs configuration in Figure 19.

CPUSS TAP consists of a 35-bit data register (called DP/AP access register) and a 4-bit instruction register. The important CPUSS TAP instructions to program the device through the JTAG are listed in Table 12.

Table 12 **XMC5000 CPUSS TAP JTAG instructions**

Bit Code [3:0]	Instruction	XMC5000 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.

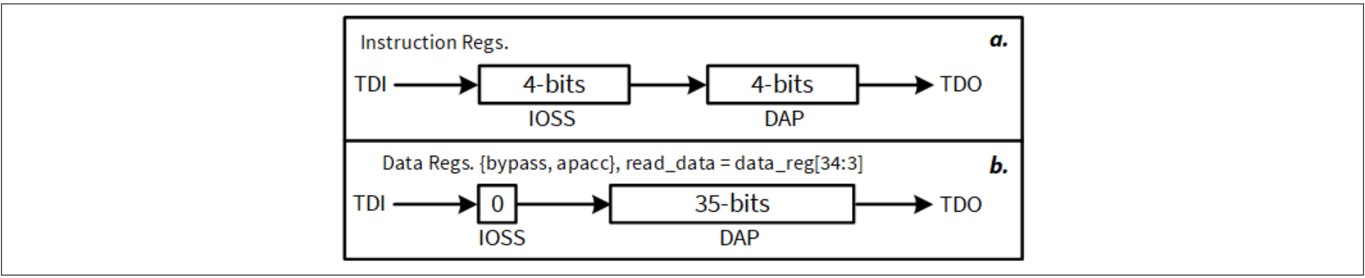


Figure 19 **IOSS/DAP TAP configuration examples**

1. Instructions registers combined. 8 bits in total.
2. Access the DAP's APACC registers for device debug and programming. IOSS TAP is in bypass mode. 36 bits in total.

Revision history

Version	Date	Description
**	2025-07-02	New document

Edition 2025-07-02

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-krn1751393449057