PSOC™ Edge MCU

Programming Specification

About this document

Scope and purpose

This document provides the information necessary to acquire the PSOC™ Edge MCU family. It describes the communication protocol required for access by an external programmer, explains the acquisition algorithm, and gives a basic description of the physical connection. Pin locations and the electrical and timing specifications of the physical connection are not a part of this document: they can be found in the device datasheet. The algorithms described in the following sections are compatible with the entire PSOC™ Edge MCU family.

Intended audience

This document is intended for anyone who wants to program PSOC™ Edge MCU.

Abbreviations and definitions

Abbreviation	Definition
AHB	AMBA (advanced microcontroller bus architecture) high-performance bus, an Arm ® data transfer bus
DAP	Debug Access Port
JTAG	Joint Test Action Group
NVM	Non-Volatile Memory
RRAM	Resistive Random-Access Memory
S-AHB	Slave AHB Interface
SWD	Serial Wire debug
SWJ-DP	Serial Wire JTAG Debug Port
TAP	Test Access Port
XIP	eXecute In Place

Reference documents

This document should be read in conjunction with the following documents:

- Arm Debug Interface Architecture Specification ADIv6.0 (IHI0074)
- Arm® CoreSight™ System-on-Chip SoC-600 Technical Reference Manual (100806)
- Arm CoreSight Architecture Specification v3.0 (IHI0029)
- Armv8-M Architecture Reference Manual (DDI0553)
- Arm® Cortex®-M33 Processor Technical Reference Manual (100230)
- Arm® Cortex®-M55 Processor Technical Reference Manual (101051)
- PSoC™ Edge E84 MCU architecture reference manual (002-37464)
- Authenticated debug for PSOC™ Edge (AN239757)
- Getting started with PSOC™ Edge security (AN237849)

- Selecting and configuring memories for power and performance in PSOC™ Edge MCU (AN239774)

Table of contents

Table of contents

1 Introduction

This document provides the information necessary to acquire the PSOC™ Edge MCU family. It describes the communication protocol required for access by an external programmer, explains the acquisition algorithm, and gives a basic description of the physical connection. Pin locations and the electrical and timing specifications of the physical connection are not a part of this document: they can be found in the device datasheet. The algorithms described in the following sections are compatible with the entire PSOC™ Edge MCU family.

1.1 Programmer

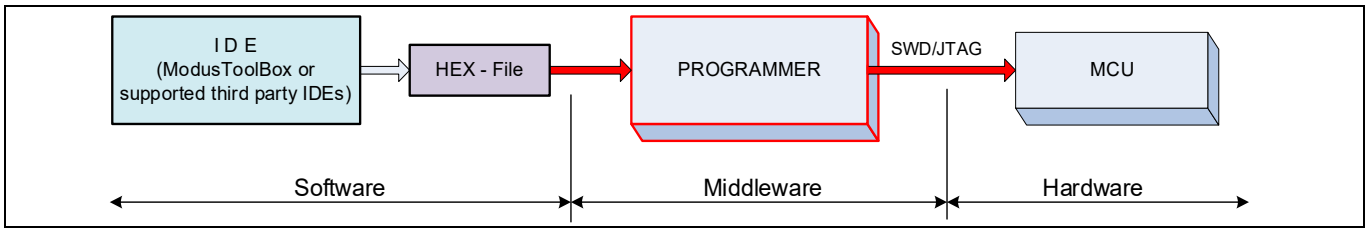

Figure 1 Programmer in Development Environment

In the manufacturing environment, the integrated development environment (IDE) block is absent because its main purpose is to produce a binary file (hex, elf, etc.). The programmer performs three functions:

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

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

In a software-centric structure, the programmer's hardware 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 binary 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 the functions of the programmer are implemented in one device, including storage for the binary file. Its main purpose is to act as a mass programmer in manufacturing.

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

1.2 PSOC™ Edge MCU family overview

The PSOC™ Edge MCU family is a Dual-CPU utilizing the Arm® Cortex®-M55 and Cortex®-M33 processor cores. This MCU family supports the Arm® SWJ-DP Interface for programming and debugging operations, using SWD or JTAG protocols.

The PSOC™ Edge MCU includes the internal RRAM memory for the secure applications and relies on the external flash memory chip connected via the high-speed QSPI/Octal SPI/HyperBus™ interface where the user application and data are stored. Upon reset, the application can either be copied to the system RAM or executed directly from the external flash thanks to the SMIF XIP (eXecute In Place) feature.

Introduction

The part can be programmed after it is installed in the system by way of the SWD or JTAG interface (in-system programming). External flash memory is programmed by means of Flash Loaders. Flash Loader is a small application which gets uploaded to the system RAM and executed. The debugger then passes the programming data to the Flash Loader, which then performs the actual flash programming.

This document does not describe the actual programming process; instead, it focuses on the specific device acquisition procedures required for flash programming. Many important topics are detailed in the appendices. Other device-specific information can be found in the device's datasheet or reference manual.

This document includes the following appendices:

Appendix A: Intel hex file format

Appendix B: Joint test action group (JTAG) protocol

Appendix C: Code example

2 Nonvolatile memory subsystem

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

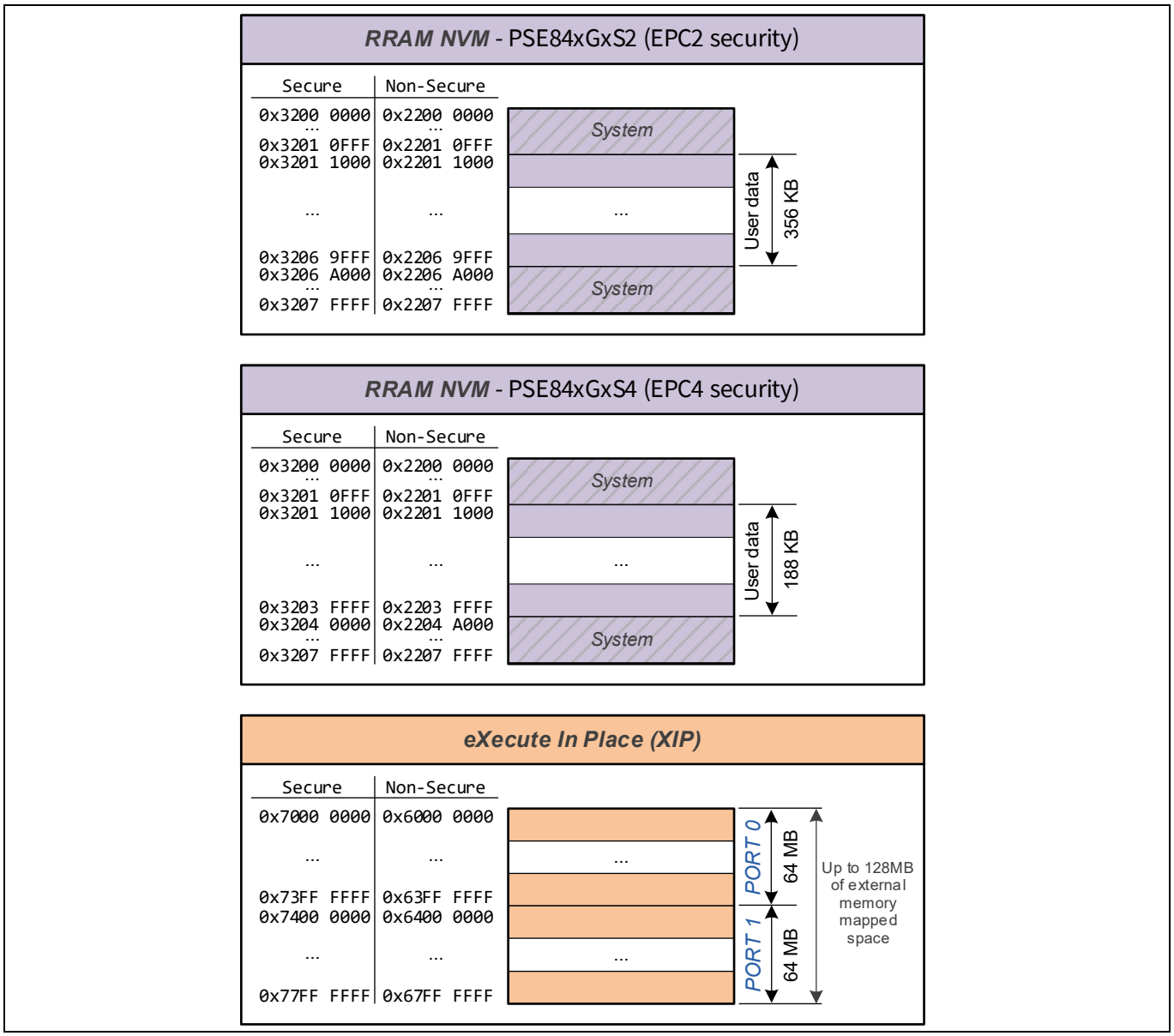

Figure 2 Nonvolatile subsystem

2.1 Resistive Random-Access Memory (RRAM)

PSOC™ Edge MCU family contains up to 512 KB of nonvolatile Resistive Random-Access Memory (RRAM).

The base address of the RRAM NVM in MCUs address space is 0x3200 0000 / 0x2200 0000 (Secure/Non-Secure S-AHB aliases). The programmable address range (user's area that can be safely used for the applications and custom data) may vary depending on the MCU security level:

- **0x3201 1000 - 0x3206 9FFF (0x2201 1000 - 0x2206 9FFF)** - 356 KB for PSE84xGxS2 devices (EPC2 security)
- **0x3201 1000 - 0x3203 FFFF (0x2201 1000 - 0x3203 FFFF)** - 188 KB for PSE84xGxS4 devices (EPC4 security)

While the default offset of the programmable area is 0x11000, users may free up to 28 KB of the additional space at the start of this area by replacing the extended boot image and setting the appropriate offset in the programming tools.

Refer to “Selecting and configuring memories for power and performance in PSOC™ Edge MCU” application notes for the detailed memory map and guidance on the applications and data storage selection.

Refer to “Getting started with PSOC™ Edge security” application notes for the details of replacing the extended boot image.

2.2 eXecute in Place (XIP)

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

Programming of the external flash memory devices via the SMIF IP block can be supported using a flash loader. A flash loader is an application compiled for a target CPU that implements programming algorithms and follows specific rules (framework) defined by a third-party IDE like **Keil µVision**, where **CMSIS-based flash loaders** are used. Such algorithms are loaded into target SRAM by programming software and executed from there for memory bank programming. Infineon provides support of such algorithms for 3rd party development tools like **Keil µVision** (MDK-ARM), **IAR Embedded Workbench** and **SEGGER J-Link Software and Documentation Pack**.

3 Hex file

The hexadecimal (hex) file describes the nonvolatile configuration of the project. It is the data source for the programmer, where the data sections must conform to the non-volatile memory subsystem, described in Section 2.

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

4 The protocol stack

This chapter explains the low-level details of the communication interface. **Figure 3** illustrates the stack of protocols involved in the programming process. The programmer must implement both hardware and software components.

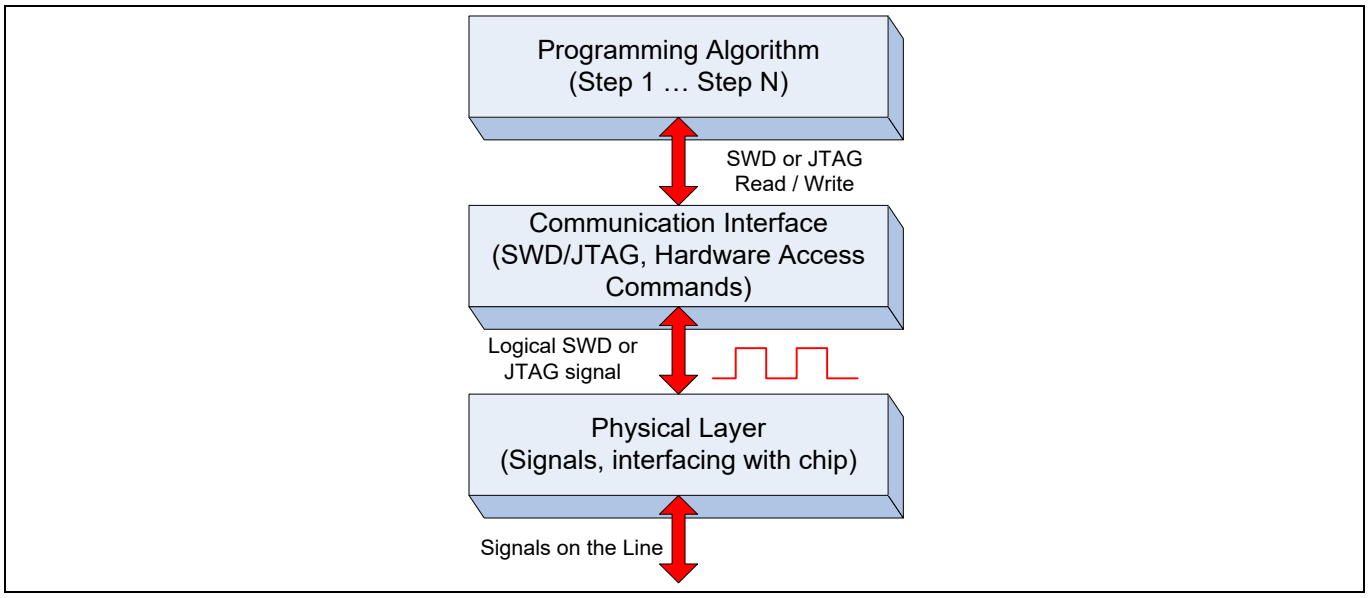

Figure 3 Programmer's protocol stack

- **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.
- **The 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.
- **The 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 PSOC™ Edge MCU. The serial wire debug (SWD) or the JTAG interface can be used as the communication protocol between the external device and the MCU.

4.2 Program and debug interface

The main purpose of PSOC™ Edge MCU program and debug interface is to support programming and debugging through the JTAG or SWD interface and to provide read and write access to all memory and registers in the system while debugging, including the Cortex®-M33 register banks when the core is running or halted.

The MCU implements a debug access port (DAP), which integrates SWJ-DP (serial wire/JTAG debug port) and complies with the Arm® specification "*Arm® Debug Interface Architecture Specification ADIv6.0 (IHI0074)*".

The debug physical port pins communicate with the DAP through the high-speed I/O matrix (HSIOM). The DAP communicates with the Cortex®-M33 and Cortex®-M55 CPU using the Arm®-specified advanced high-performance bus (AHB) interface. AHB is the systems interconnect protocol used inside the device, which facilitates memory and peripheral register access by the AHB master. The PSOC™ Edge MCU has several AHB masters, including the Arm® CM33, Arm® CM55 CPU cores, and DAP. The external host can effectively take control of the entire device through the DAP to perform programming and debugging operations.

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

- **The CM33-AP** located inside the CM33 core gives access to the CM33 internal debug components. The CM33-AP port also allows access to the rest of the system through the CM33 AHB master interfaces. This provides the debug host the same view as an application running on the CM33 core. Additionally, the CM33-AP port provides access to the debug components in the CM33 core through the external peripheral bus (EPB).
- **The CM55-AP** located inside the CM55 core gives access to the CM55 internal debug components. The CM55-AP port also allows access to the rest of the system through the CM55 AHB master interfaces. This provides the debug host the same view as an application running on the CM55 core. Additionally, the CM55-AP port provides access to the debug components in the CM55 core through the external peripheral bus (EPB).
- **The System-AP**, which gives access to the rest of the system. This allows access to the system ROM table, which is not intended to be reached any other way. The system ROM table provides the MCU ID.

4.2.1 DAP power domain

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

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

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

4.2.2 SWD/JTAG selection and DAP access

JTAG and SWD are mutually exclusive because of the Arm® SWJ-DP implementation and because they share pins. Therefore, an external programmer/debugger must be able to switch to the required protocol. The watcher circuit, implemented in SWJ-DP, detects a specific state switching sequences on SWDIOTMS and determines whether the JTAG, SWD, or DORMANT state is active. By default, JTAG operations are selected on power-on reset and therefore the JTAG protocol may be used from reset without sending a switching sequence. The debugger, however, may not know in advance the state of the debug logic when connecting to the target, so putting the debug interface in a DORMANT state first, and then switching to JTAG or SWD is recommended.For a more detailed description, see the "Switching between SWD and JTAG" section in "Arm Debug Interface Architecture Specification ADIv6.0 (IHI0074)".

The protocol stack

The DAP functionally is split into two control units:

- Debug port (DP) – Is responsible for the physical connection to the programmer or debugger.
- Access port (AP) – Provides the interface between the DAP module and one or more debug components (such as the Cortex®-M33 or Cortex®-M55 CPU).

For more information about the DP/AP access commands and registers, see the "The Access Port" chapter "Arm Debug Interface Architecture Specification ADIv6.0 (IHI0074)".

For information on the structure of the JTAG, see **Appendix B: Joint test action group (JTAG) protocol**

4.2.3 Physical layer

This section summarizes the hardware connection between the programmer and the PSOC™ Edge MCU for programming. **Figure 4** shows the generic connection between the MCU and the programmer. See **Table 1** for pins/signals description.

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

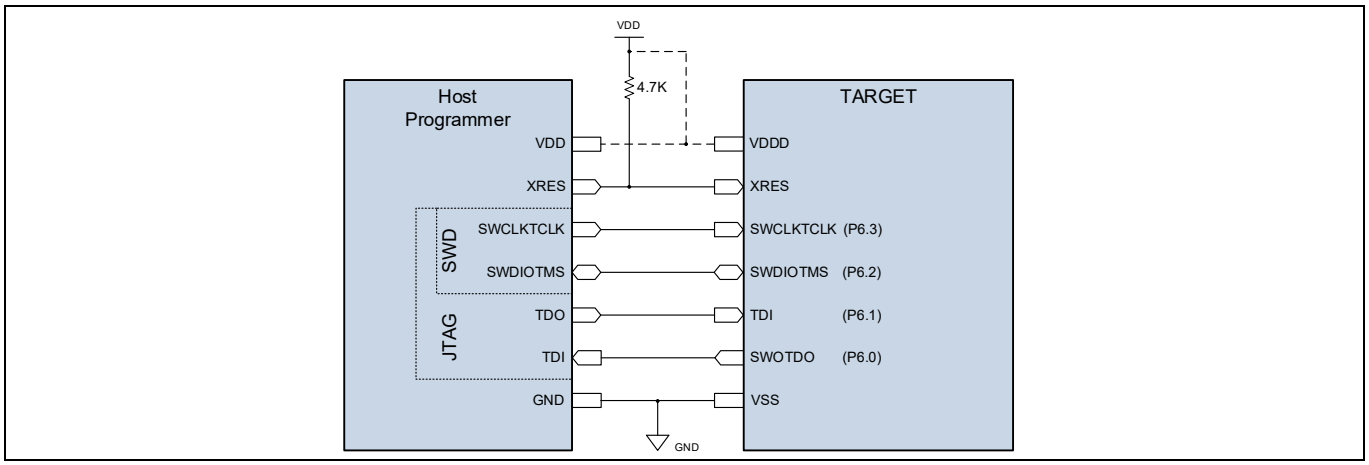

Figure 4 Connection schematic of the programmer

Table 1 Pins/signals

Pin	SWD		JTAG		Description
	Signal name	Mandatory	Signal name	Mandatory	
SWCLKTCLK	SWCLK (serial wire clock)	YES	TCLK (test clock)	YES	Data synchronization clock, driven by the host programmer/debugger. For SWD, the host should perform all read or write operations on the SWDIO line on the falling edge of SWDCK. The MCU performs read or write operations on SWDIO on the rising edge of SWDCK. For JTAG, the host writes to the TMS and TDI pins of the MCU on the falling edge of TCK and

Pin	SWD		JTAG		Description
	Signal name	Mandatory	Signal name	Mandatory	
					the MCU reads data on its TMS and TDI lines on the rising edge of TCK. MCU writes to its TDO line on the falling edge of TCK and the host reads from the TDO line of the MCU on the rising edge of TCK.
SWDIOTMS	SWDIO (serial wire data input/output)	YES	TMS (test mode select)	YES	SWDIO is a bidirectional data input/output signal. TMS is the JTAG test mode select signal, which is sampled at the rising edge of TCK to determine the next state.
SWOTDO	SWO (serial wire output)	NO	TDO (test data out)	YES	SWO signal (also known as TRACESWO) is required for serial wire viewer (SWV) and not required for SWD programming. It provides real-time data trace information from the MCU, via the SWO pin, while the CPU continues to run at full speed. Data trace via SWV is not available using the JTAG interface. TDO signal represents the data shifted out of the device's test or programming logic and is valid on the falling edge of TCK when the internal state machine is in the correct state.
TDI	-	-	TDI (Test Data In)	YES	TDI signal represents the data shifted into the device's test or programming logic. It is sampled at the rising edge of TCK.
XRES	XRES (External Reset)	NO [1]	XRES (Reset)	NO [1]	External reset active LOW signal. The XRES is not related to the ARM standard. It is used to reset the part as a first step in a programming flow.

[1] XRES pin is mandatory for "Reset" MCU acquisition mode, but not used for "power cycle" mode.

Pin	SWD		JTAG		Description
	Signal name	Mandatory	Signal name	Mandatory	
					Note: XRES 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)
VDD	VDD (voltage drain drain)	NO [1]	VDD (voltage drain drain)	NO [1]	Positive supply voltage. The MCU can be powered by external power supply or by programmer.

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

Reset mode: To start programming, the host toggles the XRES line and then sends SWD/JTAG commands (see **Hardware-specific subroutines** section). The power on the PSOC™ Edge MCU board can be supplied by the host or by an external power adapter (the VDD line can be optional).

- **Power cycle mode**: To start programming, the host powers on the MCU and then starts sending the SW/JTAG commands. The XRES line is not used.

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

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

[1] VDD pin is mandatory for "power cycle" MCU acquisition mode, where the programmer powers the MCU and external power is not applied. For "reset" acquisition mode, the source of power supplier does not matter, so the pin is optional.

5 Acquisition algorithm

This chapter describes in detail the acquisition flow of the PSOC™ Edge MCU. It starts with a high-level description of the algorithm and then describes every step using code examples. All code is based on upper-level subroutines composed of atomic SWJ instructions.

5.1 Code overview

The algorithms rely on a few low-level, hardware-specific subroutines which must be implemented by the user. This document provides only a short overview of these functions and does not specify the implementation details.

Hardware-specific backend functions for details.

Subroutine	Description
`IsJTAG()`	Returns any non-zero value if the underlying transport is JTAG (zero for SWD)
`SetXRES(...)`	Controls the voltage supplied by the debug adapter to power the target MCU. This function is optional and should return an error if not implemented.
`SetVoltage(...)`	Controls the voltage supplied by the debug adapter to power the target MCU. This function is optional and should return an error if not implemented.
`SWJSequence(...)`	Generates the given bit sequence on the SWDIO/TMS pin, used for JTAG->SWD and SWD->JTAG switching
`ReadDAP(...)` `WriteDAP(...)`	Reads (or writes) data to the CoreSight registers
`SysSleepMs(...)`	Delays the execution by the given number of milliseconds
`SysGetTimeMs()`	Returns the number of milliseconds that have elapsed since some fixed time point in the past

See **Constants and static data used in code** for a detailed list of constants used by the subroutines.

The device acquisition flow includes many low-level operations that are used in most steps. The execution flow of these subroutines is straightforward, so only a summary table is provided here. The high-level and complex operations are explained in details in the following sections.

See **Appendix C: Code example** for a detailed description.

Subroutine	Description	Example
Memory access and polling functions		
`ReadAPv2(...)`	Reads MEM-AP register of the APv2 architecture	8.3.1
`WriteAPv2(...)`	Writes MEM-AP register of the APv2 architecture	8.3.2
`ReadMem(...)`	Reads a 32-bit value from the memory address provided	8.3.3
`WriteMem(...)`	Writes a 32-bit value to the memory address provided	8.3.4
`PollMem(...)`	Polls for the expected bit-field value in the given register	8.3.5
DAP security low-level functions		
`SecureAddr(...)`	Returns secure alias for a given address	8.4.1

Subroutine	Description	Example
`ReadAndInitSecure(…)`	Reads Current Domain Secure mode	8.4.2
ARM Core control and register access functions		
`ReadCoreReg(…)`	Reads the Arm® core register, special-purpose register, or floating-point register	8.5.1
`WriteCoreReg(…)`	Writes the Arm® core register, special-purpose register, or floating-point register	8.5.2
`EnableCPU()`	Enable CPU	8.5.3
`HaltCPU()`	Enables debug and halts the CPU using the DHCSR register	8.5.4
`ResumeCPU()`	Enables debug and resumes the CPU using the DHCSR register	8.5.5
DAP initialization functions		
`DAP_Handshake (…)`	Performs a handshake	8.6.1
`DAP_Init(…)`	Initializes the debug port	8.6.2
`DAP_HandshakeAndInit(…)`	Performs a handshake and initializes the debug port	8.6.3
`DAP_ScanAP`	Scans the Access Ports for the first available with CPU registers access	8.6.4
System reset		
`Reset(…)`	Resets the device using different methods	8.7.1
ROM boot status checking and polling		
`IsBootIdle(…)`	Check whether the device is in WFA (wait for action), IDLE or DEAD branches	8.8.1
`WaitForBootIdle(…)`	Waits for the device to be in IDLE or DEAD branches	8.8.2
Acquisition helper functions		
`GetVectorTableData(…)`	Gets the reset address and initial SP values from the application vector table	8.9.1
`SetPCandSPFromVectorTable(…)`	Sets the PC and SP by getting the values from the vector table	8.9.2
Acquisition functions		
`AcquireTestMode (…)`	Performs device acquisition in test mode	8.10.1
`AcquireVectorCatch (…)`	Performs target acquisition using Vector Catch	8.10.2
`Acquire (…)`	Performs a variety of chip acquisition attempts using different methods and reset types	8.10.3
Unlocking access to the CPU		
`WaitForWFAMode(…)`	Waits for the boot code to enter WFA mode; used in the unlock procedure using debug certificates	8.11.1
`AcquireInWFAMode(…)`	Acquires the device in WFA mode; used in the unlock procedure using debug certificates	8.11.2

Subroutine	Description	Example
`LoadDebugCert(…)`	Loads the debug certificate to the RAM; used in the unlock procedure using debug certificates	8.11.3
`StartWFARequest(…)`	Starts the WFA request; used in unlock procedure using debug certificates	8.11.4
`UnlockCPUAccess(…)`	Unlocks the access to the CPU using given debug certificate	8.11.5
`UnlockCPUAccessAndHalt(…)`	Resets the CPU and halts it at the first instruction using given debug Certificate	8.11.6

5.2 DAP initialization subroutines

The very first step required to initiate a connection between the debugger and the target MCU is to initialize the DAP port. This can be achieved by using the following subroutines:

5.2.1 DAP_Handshake

Waits for the debug interface to become enabled after a device reset. In the worst case, when the boot code performs a complex application HASH verification, the boot time may last up to 3000 ms and depends on the CPU clock used by the boot code. When the PowerCycle pre-reset type is used for acquisition, the timeout depends on the design schematic and must be longer.

See the code example provided in **DAP_Handshake**.

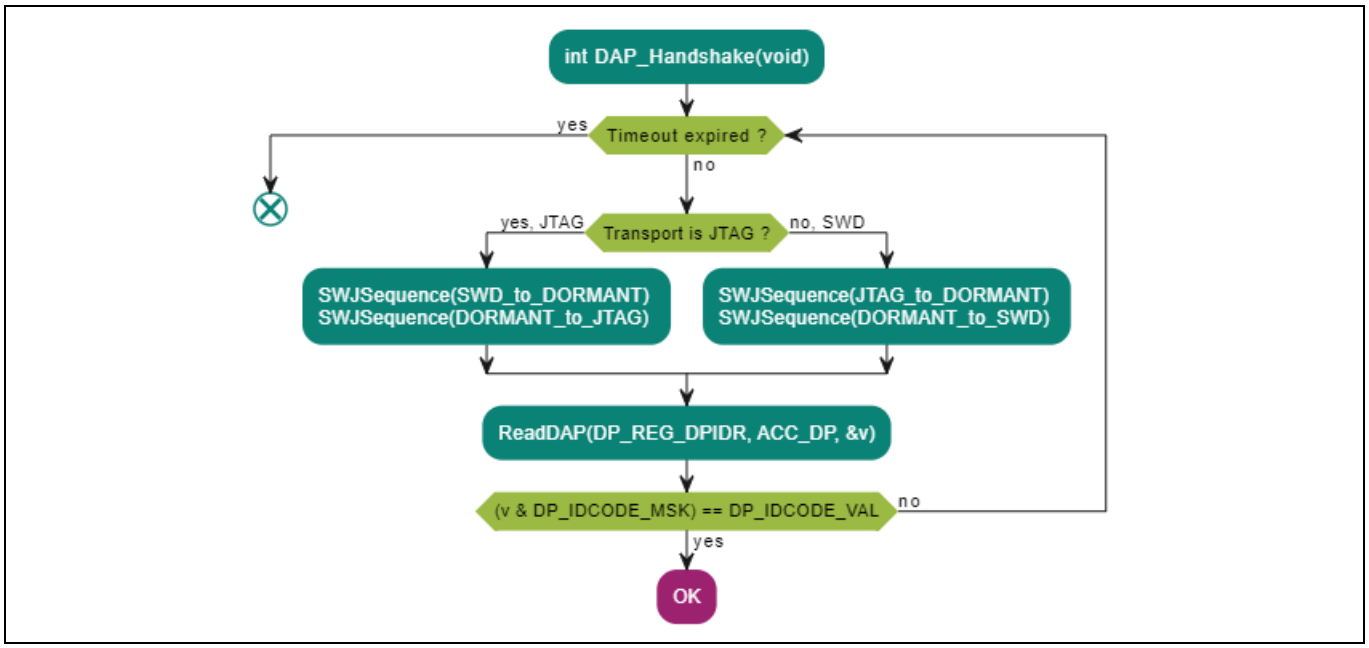

Figure 5 Flowchart of the DAP_Handshake subroutine

5.2.2 DAP_Init

Initializes the debug port for programing operations. DAP must be enabled and accessible at the moment this function is called.

Accepts access port number as the input:

Acquisition algorithm

- 0 – System AP
- 1 – CM33 AP
- 2 – CM55 AP

See the code example in **DAP_Init**.

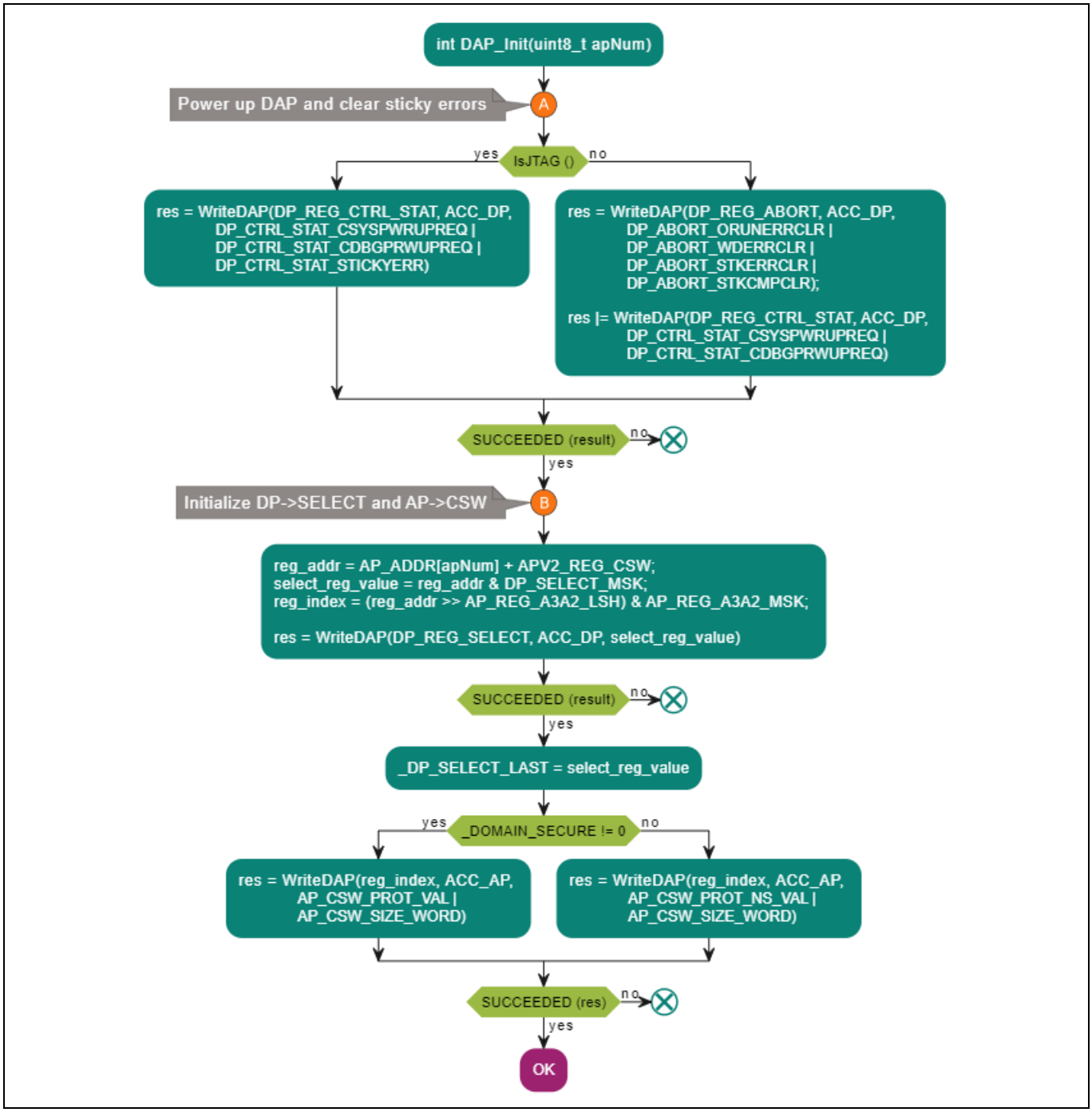

Figure 6 Flowchart of the DAP_Init subroutine

5.2.3 DAP_ScanAP

Scans the access ports first available with CPU registers access.

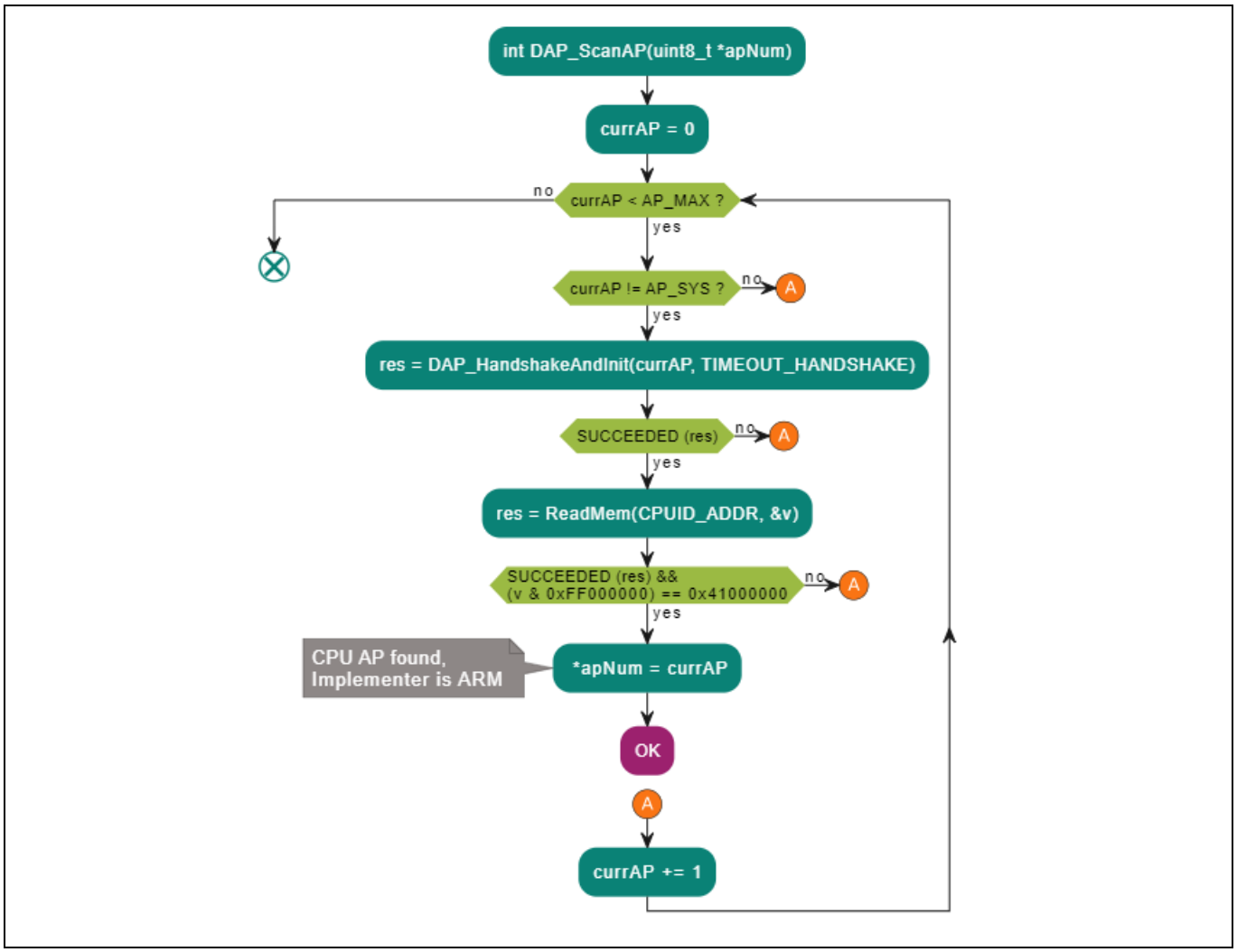

Figure 7 Flowchart of the DAP_ScanAP subroutine

See the code example in **DAP_ScanAP**.

5.2.4 Reset

The reset procedure is used during device acquisition; thus, it is very important for reset to reliably happen regardless of the state and hardware configuration of the target board. The most reliable reset type is hardware reset because it performs a full reset of the device including the reset of the retention registers which preserve their state between software resets.

The hardware reset signal can be routed to other (external to the MCU) peripherals on the target board. This ensures that all target systems start from the well-known state after reset.

Moreover, a hardware reset is required to perform the MCU acquisition in test mode, which is the recommended and most reliable acquisition method. It is strongly recommended to have the XRES pin properly routed to the debug connector.

If it is not possible for the debugger to perform a hardware reset for some reason (e.g., XRES signal not connected), the reset subroutine uses several strategies to ensure that the reset is successful. These include the following:

- Hardware reset by toggling the XRES pin

Acquisition algorithm

- Software reset by setting the RES_SOFT_CTL.TRIGGER_SOFT bit
- Software reset by setting the AIRCR.SYSRESETREQ bit
- Software reset by setting the AIRCR.VECTRESET bit
- Software reset by setting the DP->CTRL/STAT.CDBGRSTREQ bit

See the code example in **Reset**.

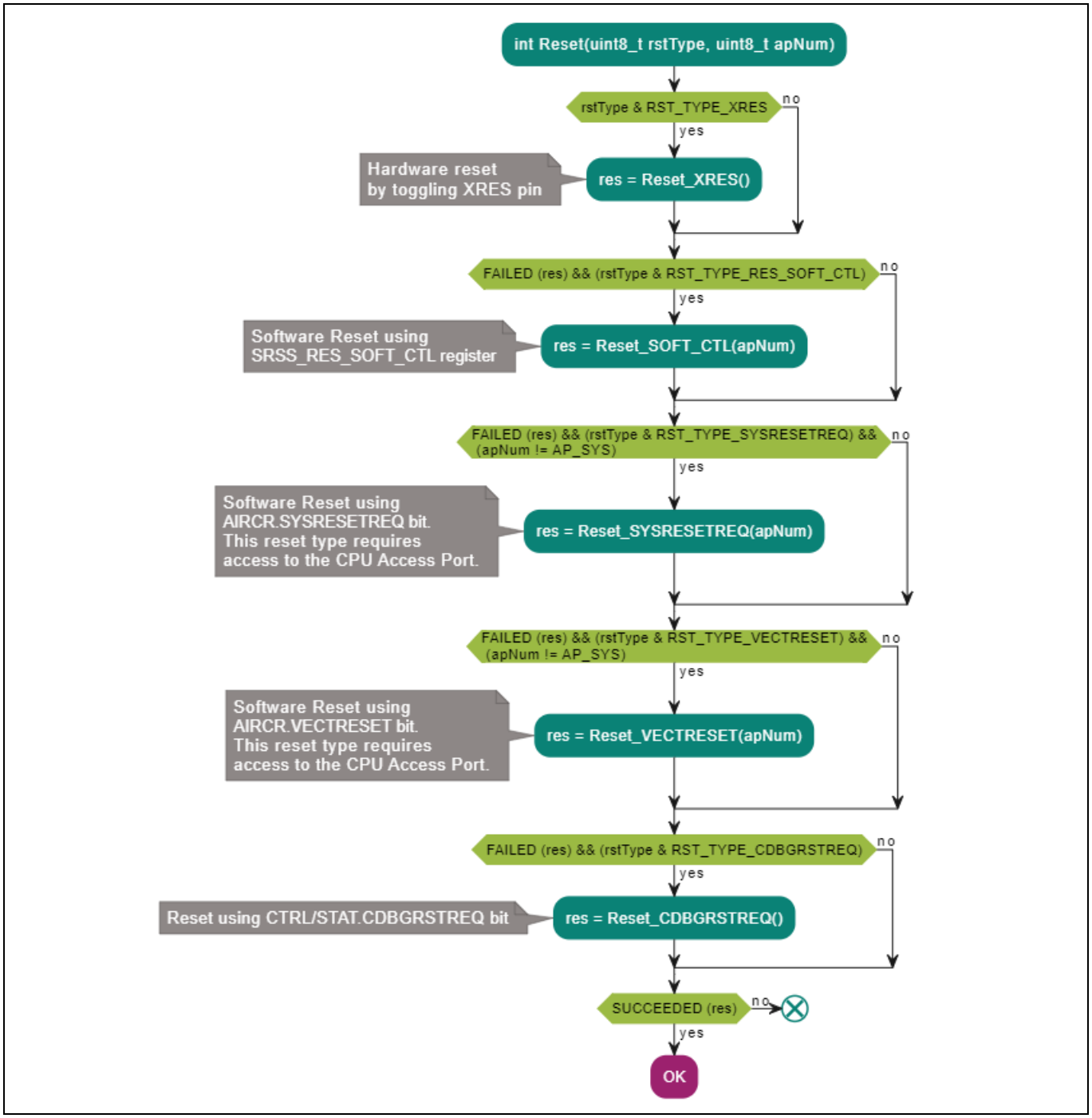

Figure 8 High-level flowchart of the reset procedure

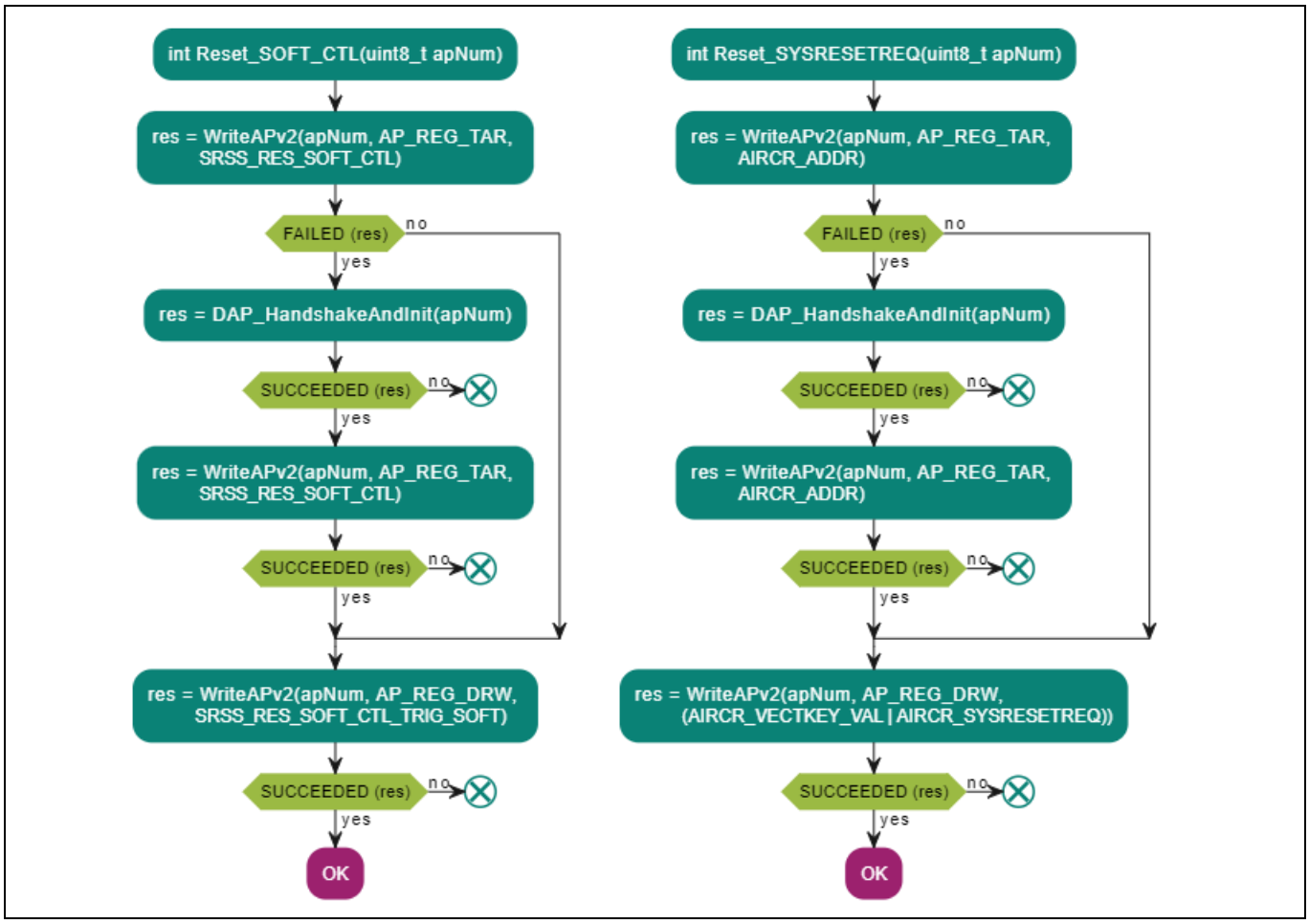

Figure 9 SOFT_RES_CTL and SYSRESETREQ reset

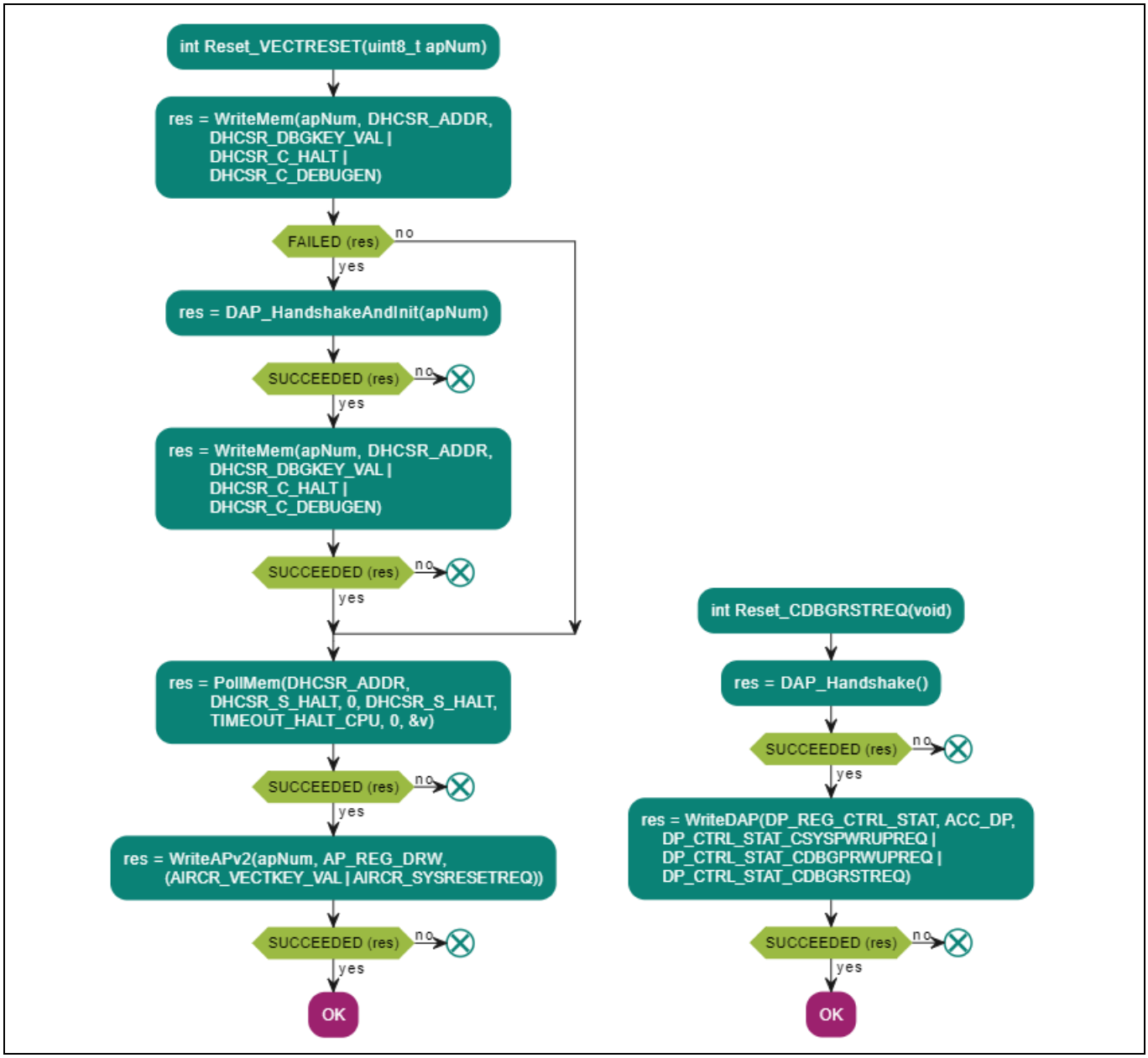

Figure 10 VECTRESET and CDBGRSTREQ reset

5.3 Acquire PSOC™ Edge MCU

The first step for the debugger before any programming or debugging actions is to acquire the device in a known good state and prevent execution of the user's code, which can put the MCU into a bad or corrupted state or repurpose the SWJ pins [1] (use them as GPIO) such that the external debugger will not be able to communicate with the device.

There are several different steps performed by the debugger sequentially for PSOC™ Edge MCU acquisition:

A. Check the boot IDLE state

B. Acquire in test mode with hardware reset

C. Acquire in test mode with software reset

D. Acquire with VectorCatch with software reset

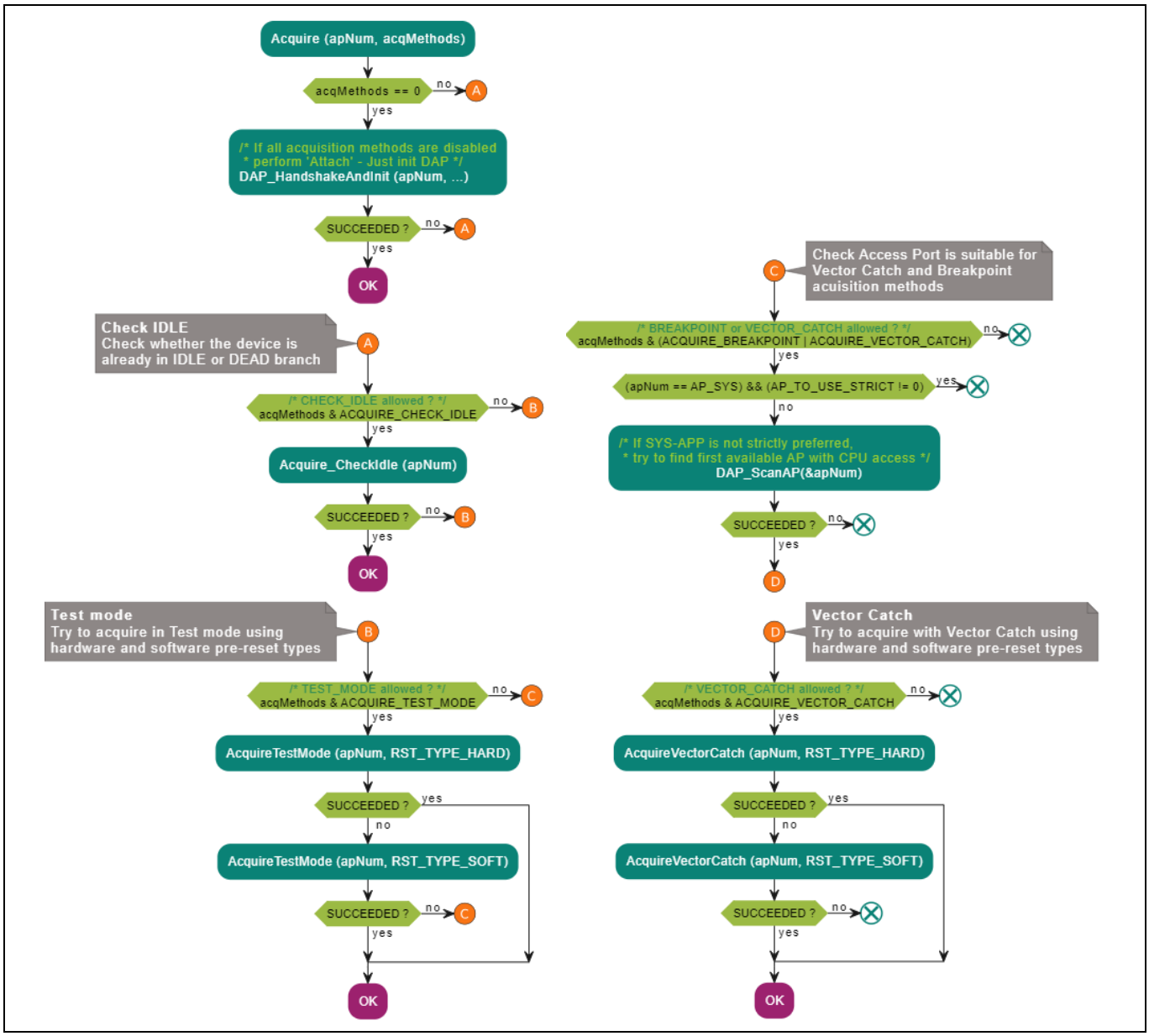

Figure 11 Top-level acquisition flowchart for PSOC™ Edge MCU

See the code example in **Acquire**.

5.3.1 Step 1 – check boot IDLE state

The IDLE state of the PSOC™ Edge MCU is the state when the ROM boot code did not launch the user's application, but is executing an endless loop in one of the following branches:

- IDLE branch. The boot code entered test mode or waits for the debugger for further actions in preproduction lifecycle stages such as SORT, NORMAL, or RMA.
- DEAD branch. Indicates recoverable failure occurred (invalid TOC object, for example).
- CORRUPTED branch. Indicates that the major system failure occurred (BIST failed, for example), the debugger has limited MCU access (via system access port only) and the programming is not possible.

IDLE and DEAD branches are sufficient MCU states for the debugger to perform further programming or debugging actions, so the additional device acquisition steps are not required.

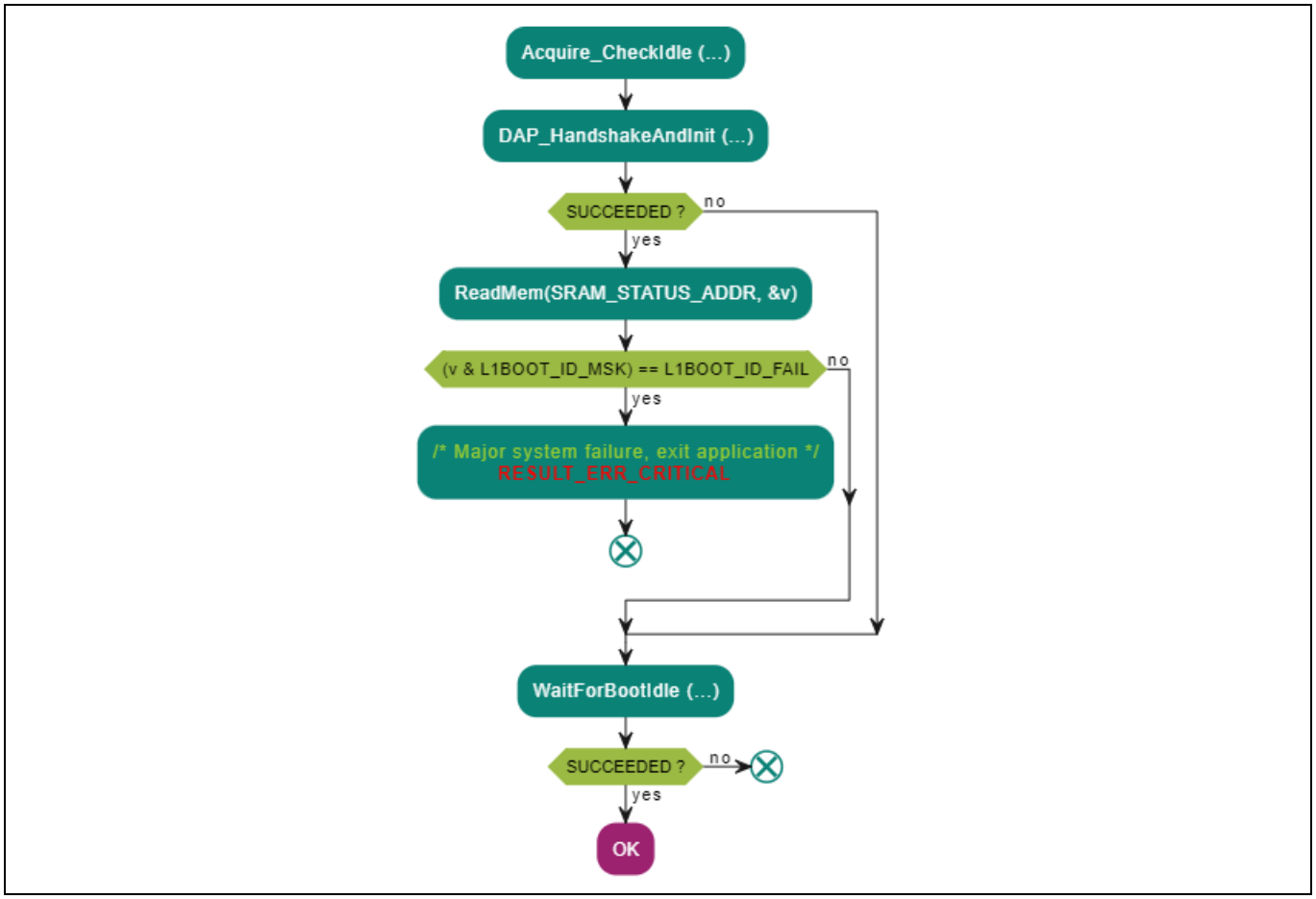

Figure 12 Flowchart for step 1 - check boot IDLE state

5.3.2 Step 2 – acquire in test mode

The test mode acquisition step has strict timing requirements that the host must meet to enter test mode successfully.

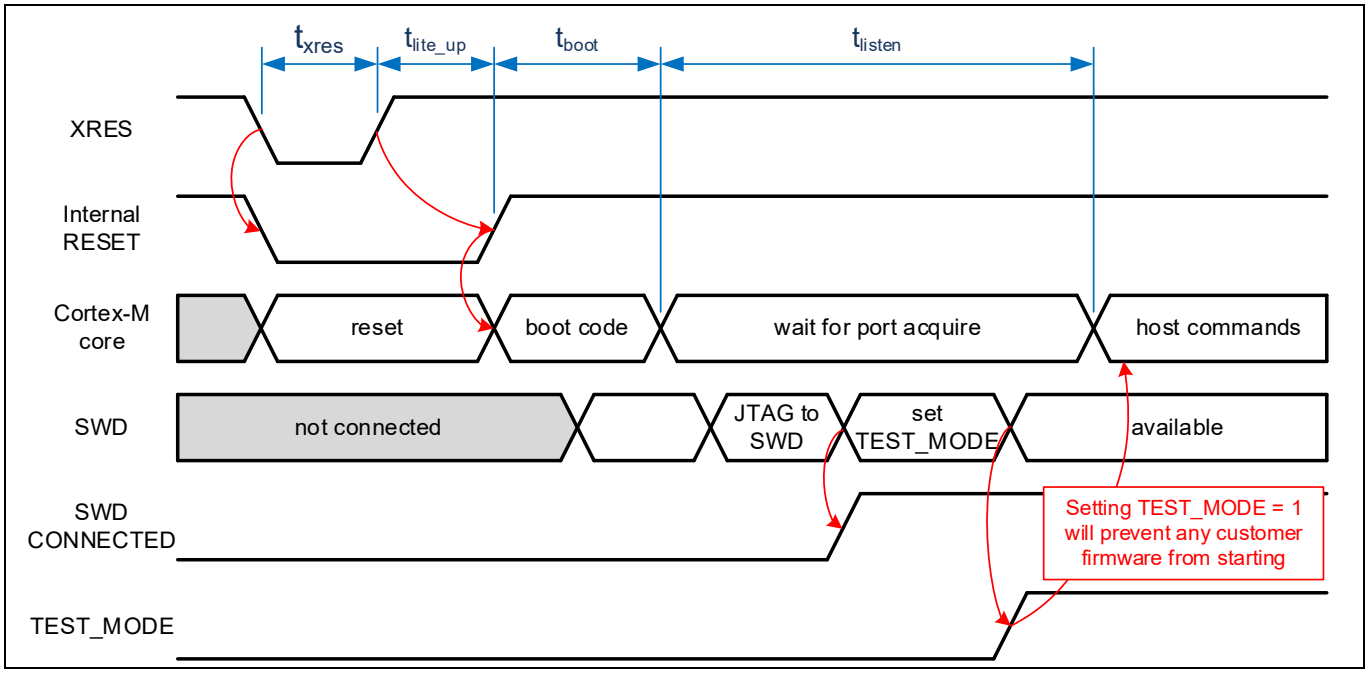

Figure 13 Timing diagram for entering test mode

This diagram details the chip's internal signals while entering test mode. Everything starts from toggling the XRES line (or applying power) so that the chip enters internal reset mode for t_{lite_up} period. After that, the system boot code starts execution. When completed, the CPU waits during a t_{listen} period for a special connection sequence on the SWJ port. If, during this time, the host sends the correct sequence of SWJ commands, the CPU enters test mode. Otherwise, it starts the execution of the user's code. Timing parameters may vary depending on the boot code execution flow (see **Table 2**). Therefore, the best way to enter test mode is to start sending an acquire sequence immediately after XRES is toggled (or power is supplied in power cycle mode). This sequence is sent iteratively until it succeeds (all SWJ transactions are ACKed and all conditions are met).

Table 2 Boot timing parameters

Parameter	Description	Min	Max	Units
t_{lite_up}	Time from reset release until the CPU begins executing the boot code	-	250	µs
t_{boot}	Time from when the boot code started execution until it opens SWJ lines and starts waiting for the TEST_MODE sequence. This time varies depending on the CPU clock, device lifecycle stage, etc.	0.7	5000	ms
t_{listen}	Amount of time the boot code waits and listens for the SWJ port initialization sequence before starting the application firmware execution. Note that the default duration of the listen window (t_{listen}) is 100 ms, but it may vary depending on the eFuse data.	0	100	ms

Figure 13 shows the test mode acquisition procedure. It is detailed in terms of the SWD transaction.

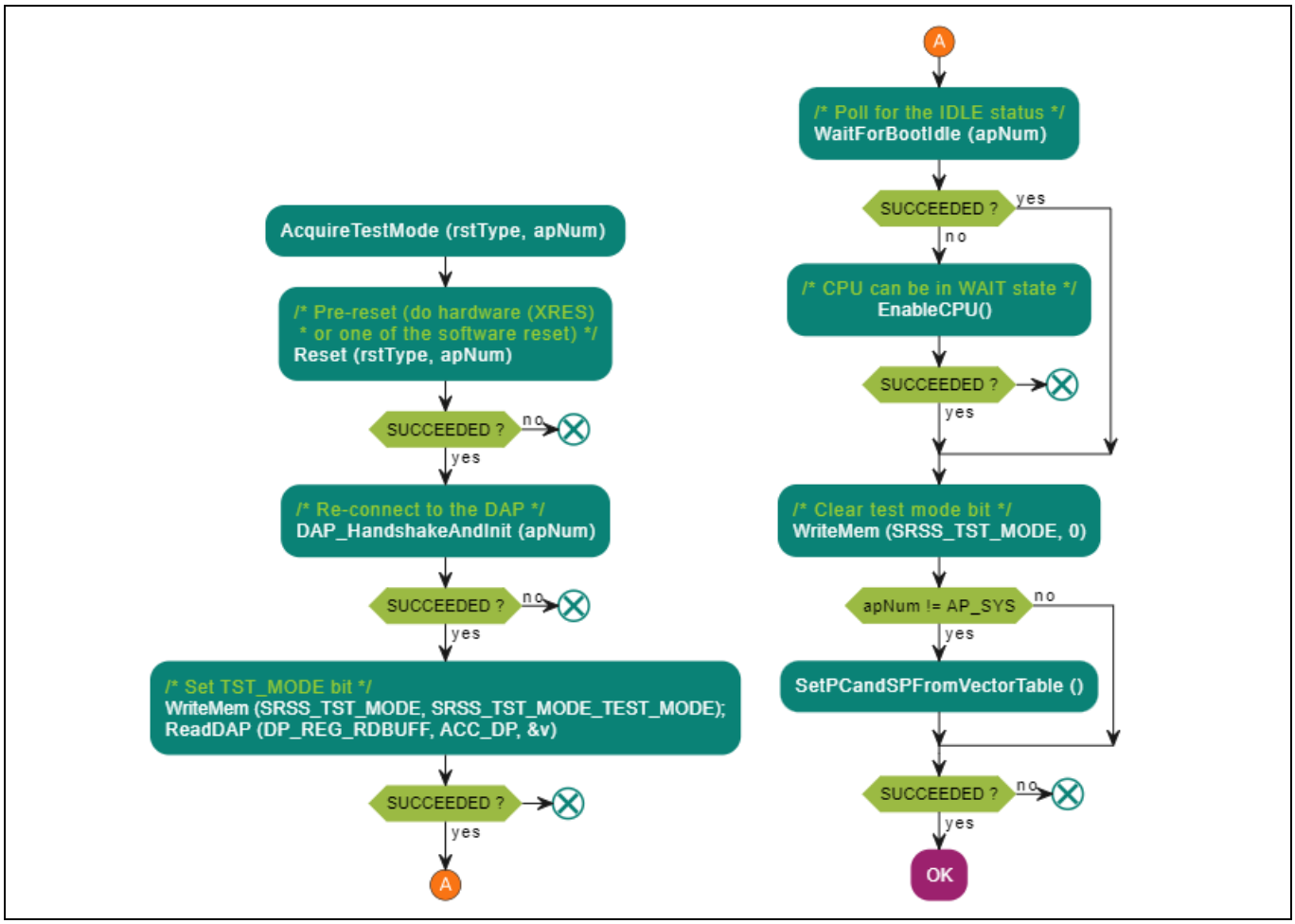

Figure 14 Flowchart for step 2 - acquire in test mode

See the code example in **AcquireTestMode**.

5.3.3 Step 3 – acquire PSOC™ Edge MCU using vector catch

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

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

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

For a host programmer with any of these constraints, the modified acquire-chip sequence provided in this section does not require XRES/power supply toggling, and it does not have the test-mode timing requirements.

Acquisition algorithm

Only the SWJ protocol pins are used for programming. This modified sequence works only under the following conditions:

- The SWJ pins on the 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 the 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 states respectively).

Developers wanting to program devices using the modified sequence should be aware of these limitations. Devices coming from the factory satisfy both these conditions, and therefore can be programmed using the modified acquire sequence. However, if firmware that does not meet any of these conditions is programmed to the MCU, 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.

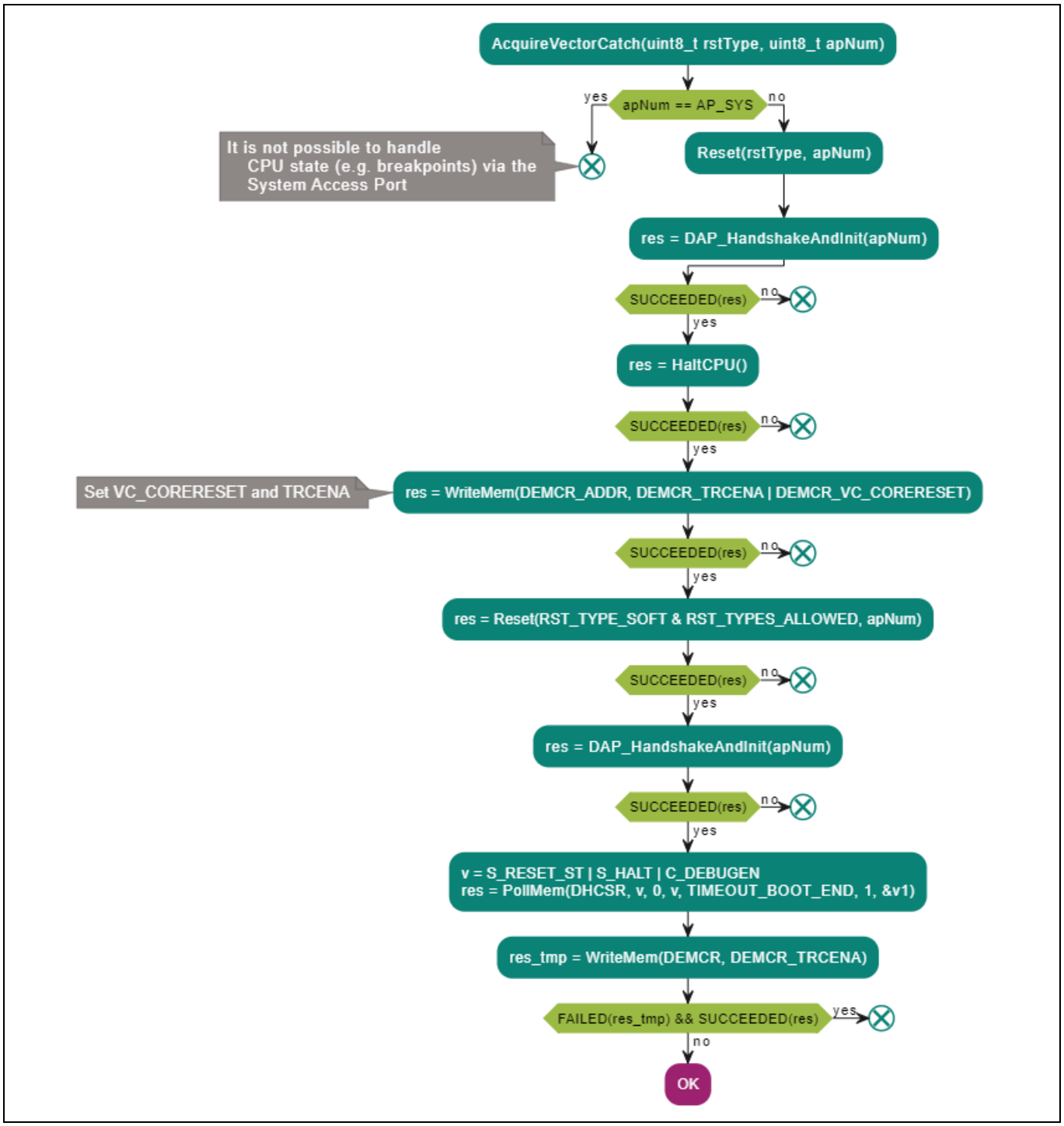

Figure 15 Acquire using vector catch

5.4 Unlock the access to the CPU (helper functions)

When the device is in SECURE lifecycle, the Cortex®-M33 or/and Cortex®-M55 access ports can be disabled by the access restrictions policy. The access port can be either temporarily or permanently disabled. If temporarily disabled, it can be re-enabled using the debug certificate.

Note that certificate cannot override the access port that is permanently disabled by access restrictions.

Acquisition algorithm

The BootROM verifies the certificate; if the verification successful, enables CM33-AP or/and CM55-AP as specified in the certificate.

Debug certificate can be placed in:

1. RAM: Sys-AP must be enabled to load the debug certificate directly into the RAM which is reserved by BootROM at the end of RAM macro 0. This flow can be used to debug L1 in PC1 and RAM applications in PC0. Flow details:
    a. The debugger sets the appropriate BOOT_DLM_CTL.REQUEST and issues a software reset by setting RES_SOFT_CTL.TRIGGER_SOFT = 1.
    b. The BootROM detects BOOT_DLM_CTL.REQUEST and sets appropriate BOOT_DLM_STATUS, and goes to IDLE loop and waiting for BOOT_DLM_CTL.WFA to be cleared by the debugger. The Sys-AP is enabled to allow debugger to upload the certificate.
    c. The debugger loads the debug token into the RAM using Sys-AP.
    d. The debugger writes the address where the debug token was loaded to BOOT_DLM_CTL2
    e. The debugger sets BOOT_DLM_CTL.WFA = 0 to release the BootROM so that it can verify the debug certificate.
    f. If verification is successful, the appropriate APs are enabled.
2. External memory. This specification does not cover this option.

5.4.1 WaitForWFAMode

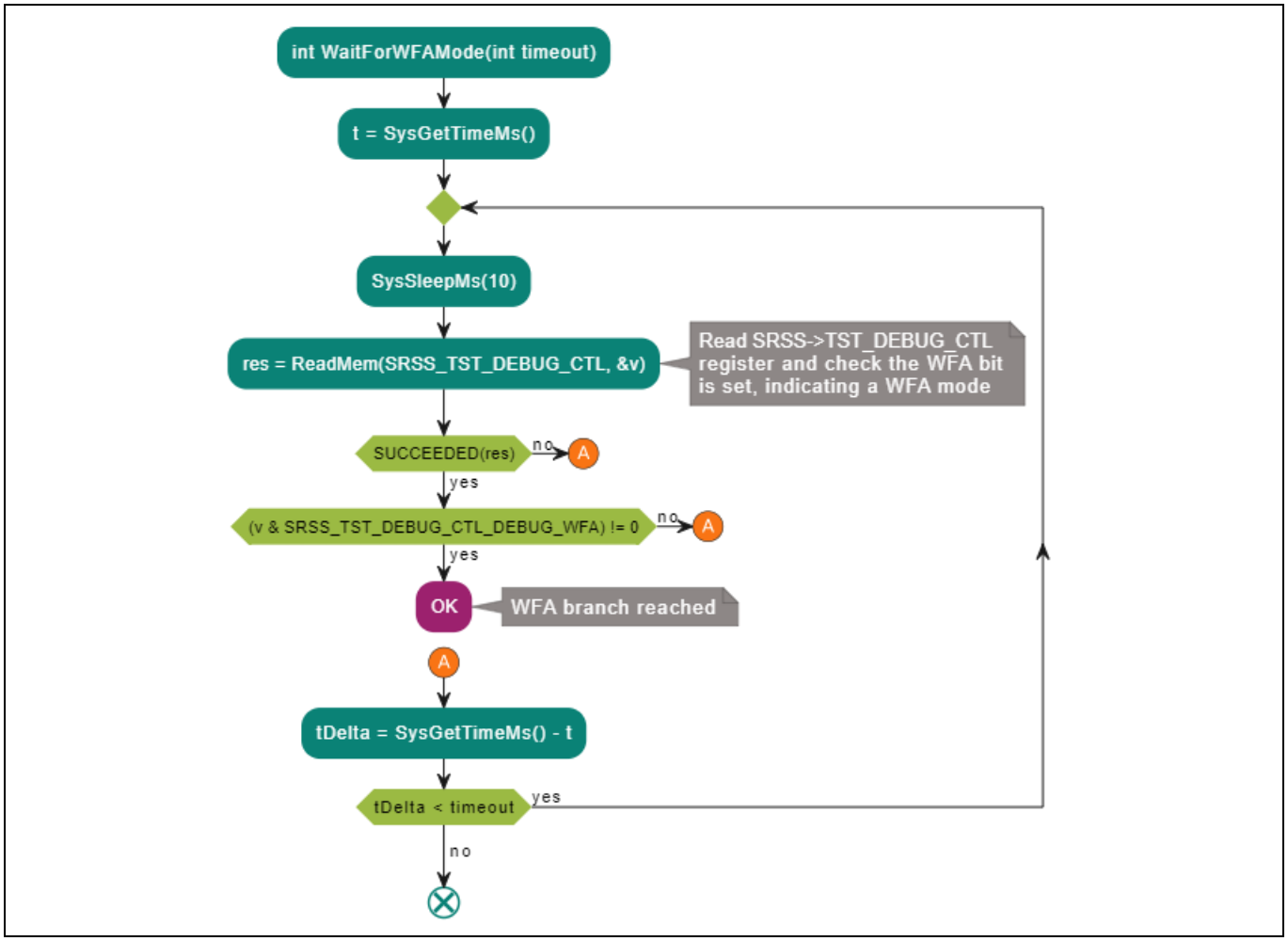

Figure 16 Wait for the device to enter WFA mode

This is a helper function used in the **AcquireInWFAMode** subroutine. It polls the SRSS_BOOT_DLM_CTL register waiting for the BootROM to enter WFA mode. In WFA mode, the BootROM is spinning in the IDLE loop waiting for the debug certificate to be loaded by the debugger.

See the code example in **WaitForWFAMode.**

5.4.2 AcquireInWFAMode

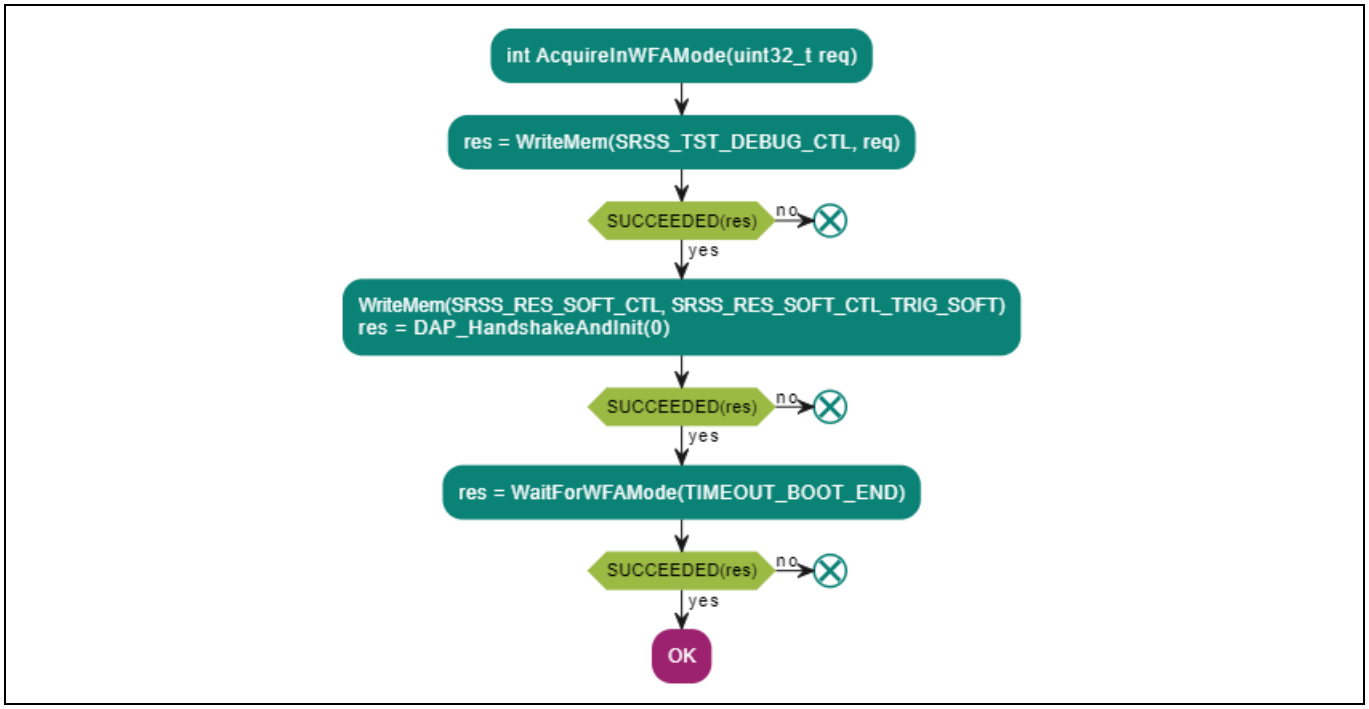

Figure 17 Acquire the device and wait for WFA mode

This function acquires the device in WFA mode and waits for the BootROM to be ready to accept the debug certificate. The debugger sets the appropriate TST_BOOT_DLM.REQUEST and issues a software reset by setting RES_SOFT_CTL.TRIGGER_SOFT = 1.

See the code example in **AcquireInWFAMode**.

5.4.3 LoadDebugCert

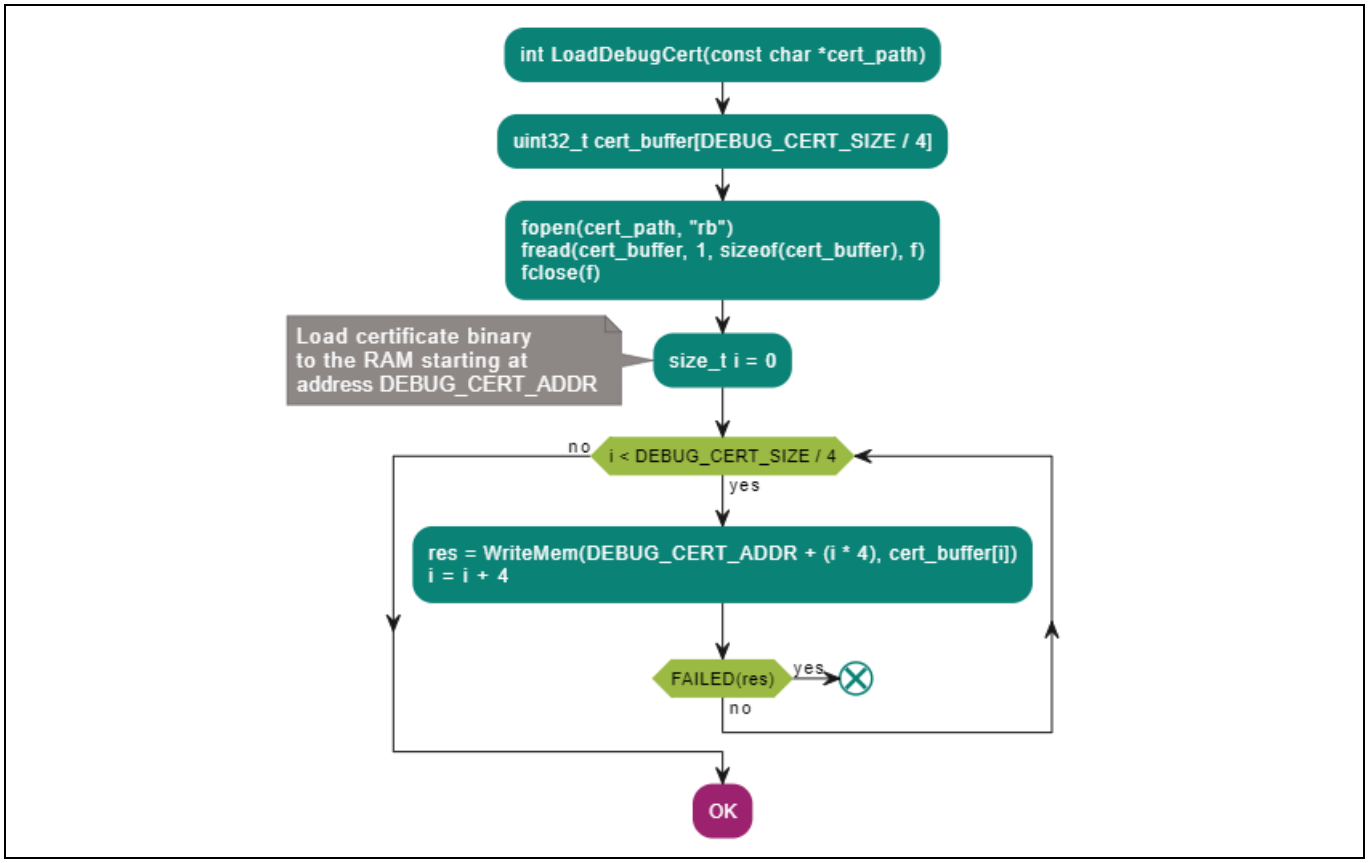

Figure 18 Loads the debug certificate to the RAM

Reads the debug certificate and loads its contents to the RAM.

See the code example in **LoadDebugCert**.

5.4.4 StartWFARequest

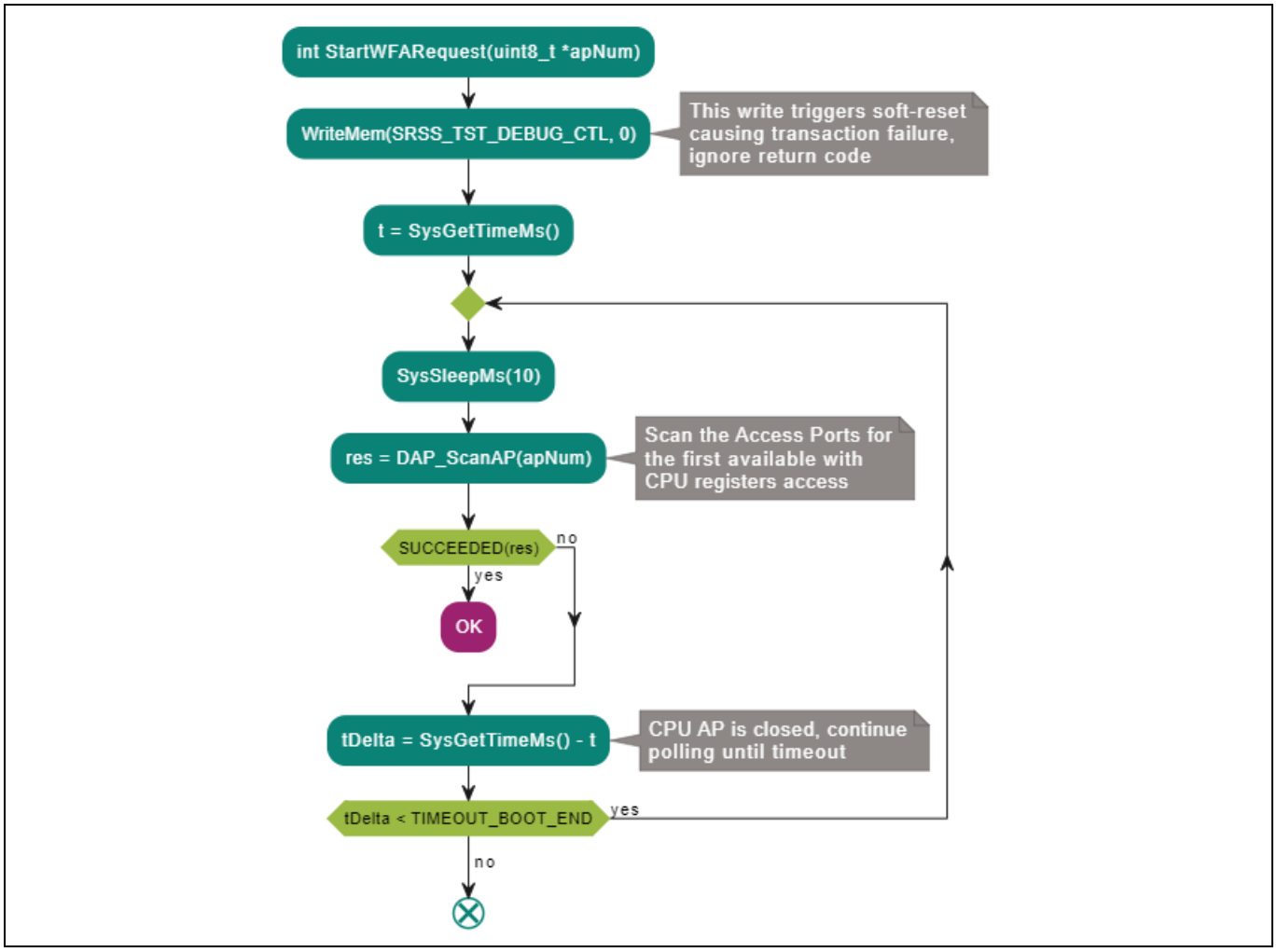

Figure 19 Executes the WFA request

Executes the WFA request by clearing the SRSS_TST_DEBUG_CTL register and waits for the CM33 access port to be opened by the BootROM.

See the code example in **StartWFARequest**.

5.5 Unlock the access to the CPU using the debug certificate

5.5.1 UnlockCPUAccess

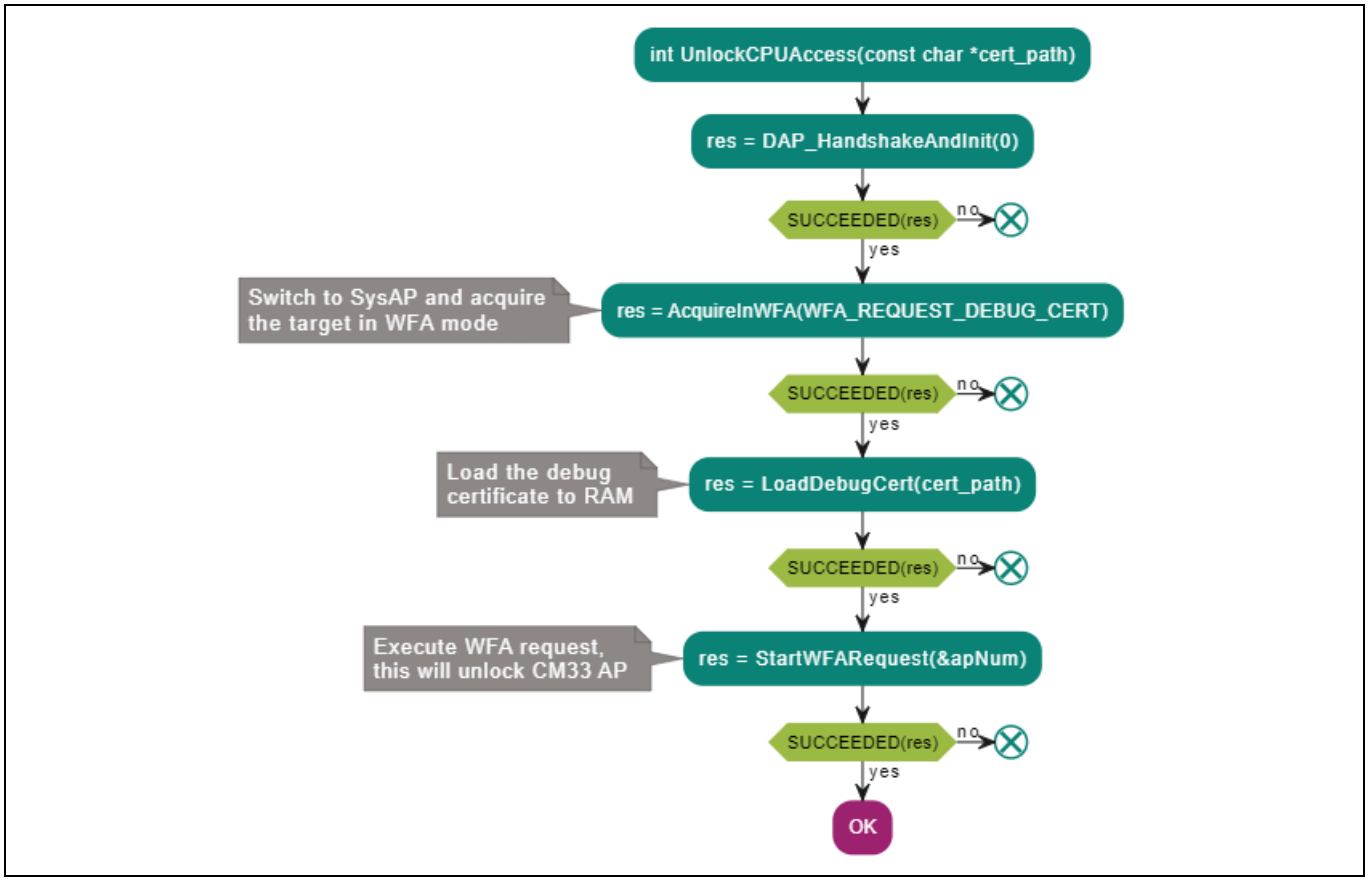

Figure 20 Unlocks the access to the CM33 access port

This function performs whole unlock sequence described in Section **5.4** . After running this function, the device is reset and CM33 access port is opened by the BootROM. The CPU is not halted after the reset, so it starts to execute the application code. This is sufficient for the debugger to attach to the running target and observe its state.

See the code example in **UnlockCPUAccess**.

5.5.2 UnlockCPUAccessAndHalt

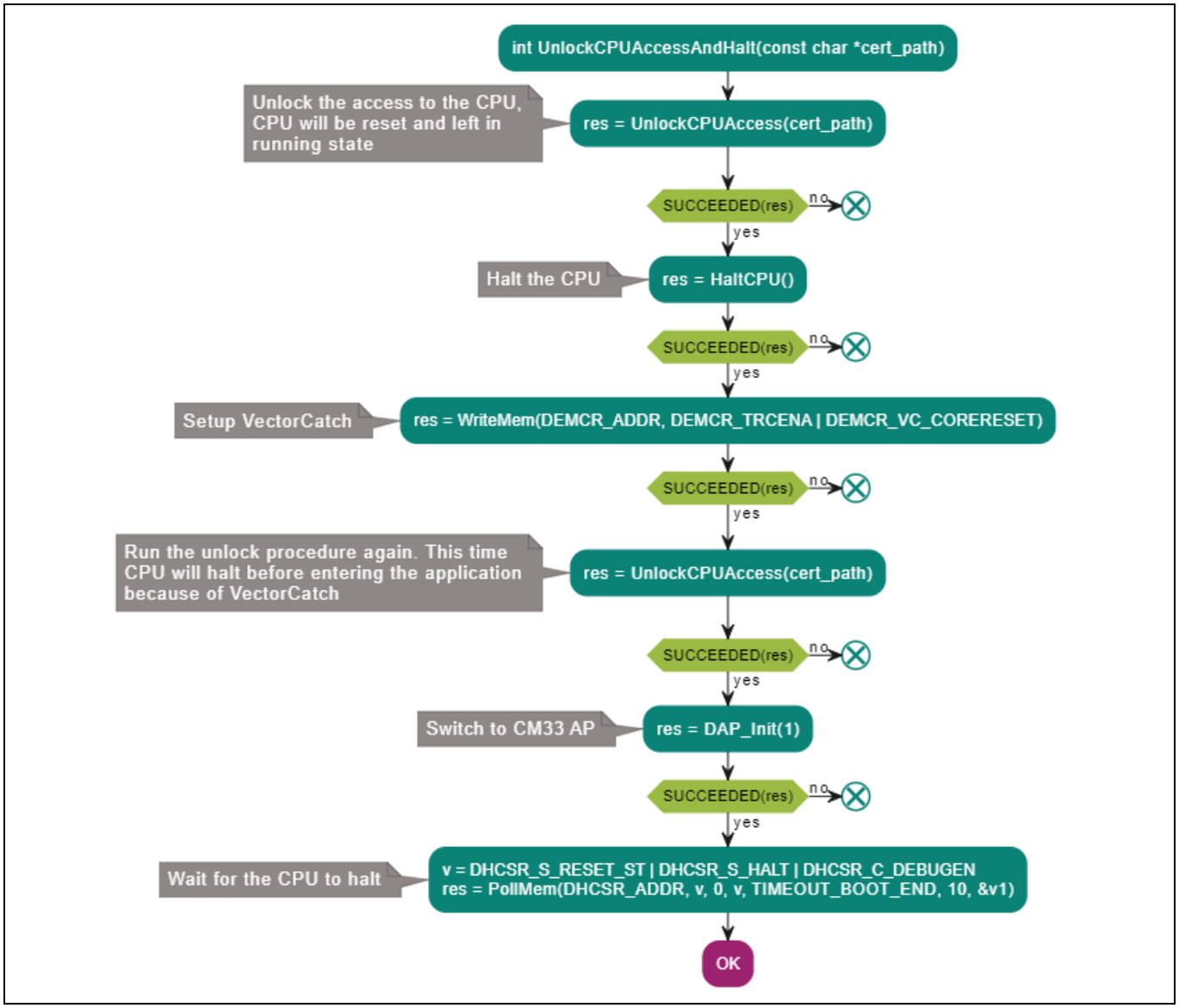

Figure 21 Unlocks the access to the CM33 access port and halts the CPU

This function is similar to **UnlockCPUAccess** but performs whole unlock sequence twice. This allows to setup vector catch in between **UnlockCPUAccess** invocations and halt the CPU before it starts to execute the application code. This allows to launch the debug session from the very beginning of the application.

See the code example in **UnlockCPUAccessAndHalt**.

6 Appendix A: Intel hex file format

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

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

Figure 22 Hex file record structure

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

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

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

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

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

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

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

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

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

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

7 Appendix B: Joint test action group (JTAG) protocol

The PSOC™ Edge MCU JTAG interface complies with the IEEE 1149.1 specification and provides additional instructions. There are two TAPs in the silicon. One is in the IOSS for boundary scan and the other is in the CPUSS DAP (IDCODE 0x4BA07477), which is used for device debug and programming. The two TAPs are connected in series, where the TDO of the IOSS TAP is connected to the TDI of the DAP TAP. This is illustrated in **Figure 23**.

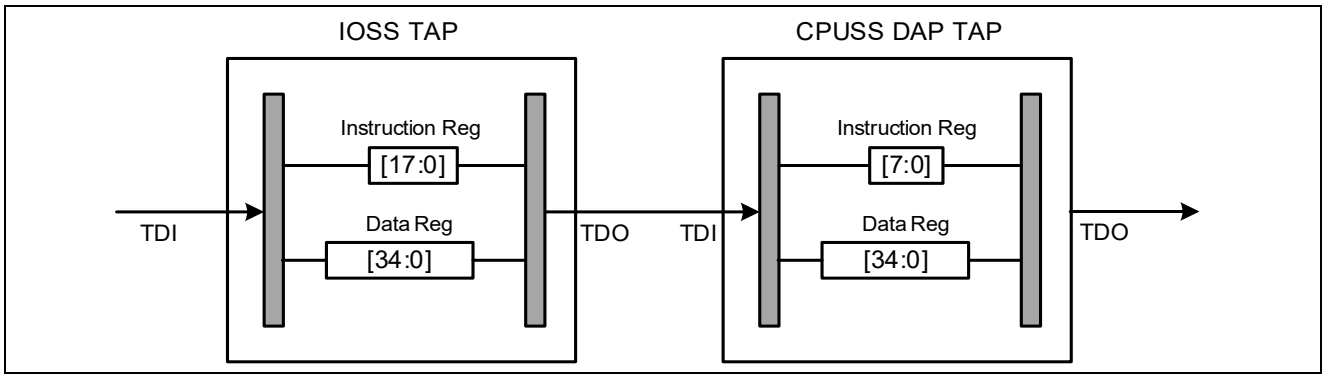

Figure 23 IOSS/DAP TAP connection

Each TAP consists of a 35-bit data register (called DP/AP access register). The size of the instruction register is 4-bits for DAP TAP and 18-bits for IOSS TAP. The important instructions to program the device through JTAG are listed in **Table 3**.

Table 3 PSOC™ Edge MCU JTAG instructions

Bit Code [3:0]	Instruction	MCU 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

If an instruction that is not applicable is shifted into a TAP, the TAP goes into bypass mode. In bypass mode, the data register is only 1 bit long with the contents of 0. The bypass mode is used to isolate the MCU TAP. For example, if targeting the IOSS TAP, the DAP TAP is put in bypass mode by shifting in the BYPASS instruction into its instruction register and if targeting the DAP TAP, the IOSS TAP will be placed in bypass mode. See the examples of TAPs configuration in **Figure 24**.

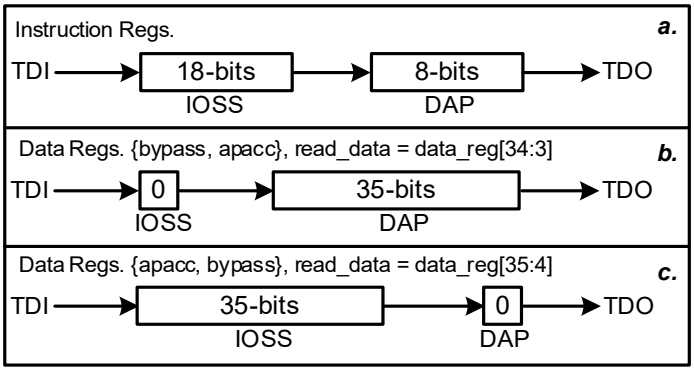

Figure 24 IOSS/DAP TAP configuration examples

a. Instruction registers combined. 26 bits total.

b. Access the DAP's APACC registers for device debug and programming. IOSS TAP in bypass mode. 36 bits total.

c. Access the IOSS APACC registers for enabling test modes. DAP TAP in bypass mode. 36 bits total.

8 Appendix C: Code example

8.1 Hardware-specific backend functions

The following code example is written in hardware-independent way so that it relies on a few backend functions which must be implemented by the user. Implementation of these functions will be different across different debug adapters and different operating systems.

This code expects that the following functions are available during linking:

8.1.1 extern int IsJTAG(void);

```
/*****************************************************************************
 * Returns any non-zero value if underlying transport is JTAG (zero for SWD).
 */
extern int IsJTAG(void);
```

8.1.2 extern int SetXRES(state);

```
/*****************************************************************************
 * Controls the logic level on XRES (nSRST) pin.
 *
 * Parameters:
 *   state - value for the XRES pin (zero -> logic low, non-zero -> logic high)
 *
 * Return value:
 *   zero     - O.K.
 *   non-zero - Error
 */
extern int SetXRES(int state);
```

8.1.3 extern int SetVoltage(voltage);

```
/*****************************************************************************
 * Controls the voltage supplied by the debug adapter to power the target MCU.
 * This function is optional and should return Error if not implemented.
 *
 * Parameters:
 *   voltage - output voltage, in millivolts
 *
 * Return value:
 *   zero     - O.K.
 *   non-zero - Error
 */
extern int SetVoltage(uint32_t voltage);
```

8.1.4 extern int SWJSequence(out_bits, num_bits);

```
/*****************************************************************************
 * Generates given bit sequence on the SWDIO/TMS pin, used for JTAG->SWD and SWD->JTAG switching.
 *
 * Parameters:
 *   out_bits - pointer to buffer containing sequence bit data, LSB first
 *   num_bits - number of bits in sequence
 *
 * Return value:
 *   zero     - O.K.
 *   non-zero - Error
 */
extern int SWJSequence(const uint8_t* out_bits, size_t num_bits);
```

8.1.5 extern int Read/WriteDAP(reg, ap_n_dp, value);

```
/*****************************************************************************
 * Reads (or Writes) data to CoreSight registers.
 *
 * Parameters:
```

```
 *   reg - register address
 *     For Read operation this parameter should take one of the following values:
 *     - DP_REG_DPIDR     0x00
 *     - DP_REG_CTRL_STAT 0x04
 *     - DP_REG_SELECT    0x08
 *     - DP_REG_RDBUFF    0x0C
 *     - AP_REG_CSW       0x00
 *     - AP_REG_TAR       0x04
 *     - AP_REG_DRW       0x0C
 *     For Write operation this parameter should take one of the following values:
 *     - DP_REG_ABORT     0x00
 *     - DP_REG_CTRL_STAT 0x04
 *     - DP_REG_SELECT    0x08
 *     - DP_REG_RDBUFF    0x0C
 *     - AP_REG_CSW       0x00
 *     - AP_REG_TAR       0x04
 *     - AP_REG_DRW       0x0C
 *   ap_n_dp - true for AP registers, false for DP registers
 *   value - value to write (or pointer to the variable where read result will be stored)
 *
 * Return value:
 *   zero     - O.K.
 *   non-zero - Error
 */
extern int ReadDAP(uint8_t reg, uint8_t ap_n_dp, uint32_t* value);
extern int WriteDAP(uint8_t reg, uint8_t ap_n_dp, uint32_t value);
```

8.1.6 extern void SysSleepMs(uint32_t msec);

```
/*****************************************************************************
 * Delays execution by the given ammount of milliseconds
 *
 * Parameters:
 *   msec - delay time in milliseconds
 *
 * Return value:
 *   none
 */
extern void SysSleepMs(uint32_t msec);
```

8.1.7 extern int SysGetTimeMs(void);

```
/*****************************************************************************
 * Returns the number of milliseconds that have elapsed since some fixed time point in the past.
 *
 * Parameters:
 *   none
 *
 * Return value:
 *   number of milliseconds
 */
int SysGetTimeMs(void);
```

8.2 Constants and static data used in code

8.2.1 Application common constants and definitions

```
/* --- Error checking --- */
#define RESULT_OK              (0)   /* Function return result: O.K. */
#define RESULT_ERR             (-1)  /* Function return result: Error */
#define RESULT_ERR_CRITICAL    (-15) /* Function return result: Critical Error */
#define SUCCEEDED(result)      ((result) >= (RESULT_OK))
#define FAILED(result)         ((result) <  (RESULT_OK))
#define BREAK_IF_FAILED(result) if (FAILED(result)) { break; }

/* --- Target acquisition methods --- */
#define ACQUIRE_CHECK_IDLE     (1 << 0) /* Initial check whether boot code is already in IDLE or DEAD branch */
#define ACQUIRE_TEST_MODE      (1 << 1) /* Test mode (TM) acquisition (recommended) */
#define ACQUIRE_VECTOR_CATCH   (1 << 2) /* Vector Catch */

/* --- MCU reset types --- */
```

```
#define RST_TYPE_XRES          (1 << 0) /* Hardware reset (XRES) */
#define RST_TYPE_POWER         (1 << 1) /* Hardware reset (Power Cycle) */
#define RST_TYPE_RES_SOFT_CTL  (1 << 2) /* Software reset (RES_SOFT_CTL.TRIGGER_SOFT) */
#define RST_TYPE_SYSRESETREQ   (1 << 3) /* Software reset (AIRCR.SYSRESETREQ) */
#define RST_TYPE_VECTRESET     (1 << 4) /* Software reset (AIRCR.VECTRESET) */
#define RST_TYPE_CDBGRSTREQ    (1 << 5) /* Software reset (DP->CTRL/STAT.CDBGRSTREQ) */
#define RST_TYPE_SOFT          (RST_TYPE_RES_SOFT_CTL | RST_TYPE_SYSRESETREQ | RST_TYPE_VECTRESET |
RST_TYPE_CDBGRSTREQ)
#define RST_TYPE_HARD          (RST_TYPE_XRES | RST_TYPE_POWER)
#define RST_TYPE_ANY           (RST_TYPE_HARD | RST_TYPE_SOFT)

/* --- Misc. --- */
#define ERR_ADDR_MSK           0xF0000000 /* Error mask for SP and PC values*/
#define LOOP_CODE              0xE7FEE7FE /* Endless loop */
```

8.2.2 MCU-specific constants and definitions

```
/* --- Suitable acquisition methods and reset types --- */
#define ACQUIRE_METHODS_ALLOWED (    \
            ACQUIRE_CHECK_IDLE |     \
            ACQUIRE_TEST_MODE |      \
            ACQUIRE_VECTOR_CATCH)
#define RST_TYPES_ALLOWED (          \
            RST_TYPE_XRES |          \
            RST_TYPE_RES_SOFT_CTL |  \
            RST_TYPE_SYSRESETREQ |   \
            RST_TYPE_VECTRESET |     \
            RST_TYPE_CDBGRSTREQ)

/* --- Access Ports --- */
#define AP_SYS                  0 /* AP[0] System Access Port */
#define AP_CM33                 1 /* AP[1] Cortex-M33 Access Port */
#define AP_CM55                 2 /* AP[2] Cortex-M55 Access Port */
#define AP_MAX                  3 /* Maximum number of Access Ports for scanning algorithm */
#define AP_TO_USE               1 /* Preferred Access Port (AP[1] - CM33 Core is used by default in this script) */
#define AP_TO_USE_STRICT        0 /* "0" - Can use any available AP if needed; "1" - Strict AP usage to preferred
only */
static const uint32_t AP_ADDR[] = { /* Array of AP adresses: */
                                0xF0000000, /* Address of System Access Port - AP[0] */
                                0xF0002000, /* Address of Cortex-M33 Access Port - AP[1] */
                                0xF0006000  /* Address of Cortex-M55 Access Port - AP[2] */
};

/* --- AP/DP registers --- */
#define DP_IDCODE_MSK           0xF0000FFF /* DP IDCODE 0x4C013477 for SWD or 0x4BA07477 for JTAG */
#define DP_IDCODE_VAL           0x40000477
                                           /* AP->CSW.Prot (bits[30:24]): */
#define AP_CSW_PROT_VAL         (0x0B<<24) /*   0x0B000000: Bus access protection control for Secure access */
#define AP_CSW_PROT_NS_VAL      (0x4B<<24) /*   0x4B000000: Bus access protection control for Non Secure access. */
#define AP_CSW_PROT_MSK         (0x4F<<24) /*   0x4F000000: Bus access protection control mask. */
#define AP_CSW_SIZE_WORD        ( 2 << 0 ) /* AP->CSW.Size: Size of access <- Word (32-bits)
                                            * AP->CSW typical write value: 0x4B000002 */
#define DP_SELECT_MSK           0xFFFFFFF0 /* Mask for bits[31:4] of DP->SELECT register */
#define AP_REG_A3A2_MSK         (3u << 2u) /* Mask for Bits[3:2] of the AP register address */
#define APV2_REG_CSW            0xD00      /* Offset of AP->CSW register */
#define APV2_REG_TAR            0xD04      /* Offset of AP->TAR register */
#define APV2_REG_DRW            0xD0C      /* Offset of AP->DRW register */

/* --- Target memory mapping --- */
#define SRAM_NS_BASE            0x24000000
#define SRAM_S_BASE             0x34000000
#define SRAM_SIZE               0x00010000
#define SRAM_DBG_ADDR    (SRAM_NS_BASE + 0x8000) /* Address in SRAM for the debug messages and status. */
#define SRAM_LOOP_ADDR   (SRAM_NS_BASE + 0x8004) /* Address in SRAM for infinite loop. Safe option is to avoid
                                                  * bottom addresses that might be used by the boot code       */
#define SRAM_STATUS_ADDR (SRAM_NS_BASE + 0x0000) /* Address in SRAM, where boot code or application stores the */
                                                  * status word */

/* --- Target-specific registers and definitions --- */
#define CPUSS_CM33_CTL          0x42260000 /* MXCM33_CM33_CTL */
#define CPUSS_CM33_S_VT_BASE    0x42261000 /* MXCM33_CM33_S_VECTOR_TABLE_BASE */
#define CPUSS_CM33_NS_VT_BASE   0x42261004 /* MXCM33_CM33_NS_VECTOR_TABLE_BASE */
#define CPUSS_CM55_CTL          0x44160000 /* MXCM55_CM55_CTL */
#define CPUSS_CM55_S_VT_BASE    0x44161000 /* MXCM55_CM55_S_VECTOR_TABLE_BASE: CM55 secure vector table base */
```

```
#define CPUSS_CM55_NS_VT_BASE    0x44161004 /* MXCM55_CM55_NS_VECTOR_TABLE_BASE: CM55 non-secure vector table base
*/
#define MSK_CPUSS_CMx_CTL_CPU_WAIT 0x00000010
#define SRSS_TST_MODE            0x42400400 /* SRSS->TST_MODE: Test Mode Control Register */
#define SRSS_TST_MODE_TEST_MODE  ( 1 << 31) /* SRSS->TST_MODE.TEST_MODE:
                                             * 1 - Indicates the chip is in test mode. 0 - Normal operation mode */
#define SRSS_BOOT_DLM_CTL        0x42400404 /* SRSS->TST_DEBUG_CTL: Debug Control Register */
#define SRSS_BOOT_DLM_CTL_DEBUG_WFA (1<<31) /* SRSS->TST_DEBUG_CTL.DEBUG_WFA: Wait for Action.
                                             * Set by BootROM when it waits for application or debug certificate to
                                             * be loaded into the RAM. The bit must be cleared to continue BootROM
                                             * operation. */
#define SRSS_BOOT_DLM_STATUS     0x4240040C /* SRSS->BOOT_DLM_STATUS: Debug Status Register */
#define SRSS_RES_SOFT_CTL        0x42400410 /* SRSS->RES_SOFT_CTL: Soft Reset Trigger Register */
#define SRSS_RES_SOFT_CTL_TRIG_SOFT (1 <<0) /* SRSS->RES_SOFT_CTL.TRIGGER_SOFT: Triggers a soft reset.
                                             * The reset clears this bit. */

/* --- Boot code status --- */
#define L1BOOT_ID_MSK            0xFF000000 /* Mask for MODULE_ID in status word */
#define L1BOOT_ID_SUCCESS        0xAA000000 /* The module IDs for BootROM in case of success */
#define L1BOOT_ID_FAIL           0xEE000000 /* The module IDs for BootROM in case of fail */
#define L1BOOT_STATUS_MSK        0x00FFFFFF /* Mask for RESULT_CODE (status) in status word */
#define L1BOOT_IDLE_BRANCH_REACHED 0x0000B5F8 /* Result code indicating BootROM reached IDLE branch */

/* --- Timings --- */
#define TIMEOUT_HANDSHAKE        2500 /* Maximum possible boot time until the debug interface is enabled */
#define TIMEOUT_HANDSHAKE_SMALL  5    /* Small timeout for the handshake when performed not after reset */
#define TIMEOUT_LISTEN_WND       200  /* Timeout for Listen window duration (100ms max) */
#define TIMEOUT_HALT_CPU         10   /* Timeout for CPU halt/unhalt actions */

/* --- Debug Certificate --- */
#define DEBUG_CERT_SIZE          808
#define DEBUG_CERT_ADDR          0x2000FC00
#define WFA_REQUEST_DEBUG_CERT   2

uint32_t _DOMAIN_SECURE; /* The state of DSCSR->CDS bit (Current Domain Secure) */
uint32_t _DP_SELECT_LAST = 0xFFFFFFFF; /* Last value written to DP.SELECT */
```

8.2.3 Standard ARM constants and definitions

```
/* --- Debug Access Port (DAP) --- */
#define ACC_DP                          (0) /* APnDP for DP access */
#define ACC_AP                          (1) /* APnDP for AP access */
#define DP_ABORT_ORUNERRCLR      ( 1 <<  4) /* DP->ABORT.ORUNERRCLR : Clears CTRL/STAT.STICKYORUN */
#define DP_ABORT_WDERRCLR        ( 1 <<  3) /* DP->ABORT.WDERRCLR   : Clears CTRL/STAT.WDATAERR   */
#define DP_ABORT_STKERRCLR       ( 1 <<  2) /* DP->ABORT.STKERRCLR  : Clears CTRL/STAT.STICKYERR  */
#define DP_ABORT_STKCMPCLR       ( 1 <<  1) /* DP->ABORT.STKCMPCLR  : Clears CTRL/STAT.STICKYERR  */
                                            /* DP->ABORT typical write value: 0x0000001E          */
#define AP_SELECT_APSEL_RSH            (24) /* AP->SELECT.APSEL (bits[31:24], 0xFF000000): Selects an AP */
#define DP_CTRL_STAT_CSYSPWRUPREQ (1 << 30) /* DP->CTRL/STAT.CSYSPWRUPREQ : System powerup request       */
#define DP_CTRL_STAT_CDBGPRWUPREQ (1 << 28) /* DP->CTRL/STAT.CDBGPRWUPREQ : Debug powerup request        */
#define DP_CTRL_STAT_CDBGRSTREQ  ( 1 << 26) /* DP->CTRL/STAT.CDBGRSTREQ   : Debug reset request          */
#define DP_CTRL_STAT_STICKYERR   ( 1 <<  5) /* DP->CTRL/STAT.STICKYERR    : Error in AP transaction      */
#define DP_CTRL_STAT_STICKYCMP   ( 1 <<  4) /* DP->CTRL/STAT.STICKYCMP    : Match on a pushed operations */
#define DP_CTRL_STAT_STICKYORUN  ( 1 <<  1) /* DP->CTRL/STAT.STICKYORUN   : Overrun detection            */

/* --- System Control Block (SCB) --- */
#define CPUID_ADDR               0xE000ED00 /* SCB->CPUID Base Register                                    */
#define VTOR_ADDR                0xE000ED08 /* SCB->VTOR Vector Table Offset Register                      */
#define AIRCR_ADDR               0xE000ED0C /* SCB->AIRCR: Application Interrupt and Reset Control Register */
#define AIRCR_VECTKEY_VAL      (0x05FA<<16) /* SCB->AIRCR.VECTKEY     : Vector Key.                        */
#define AIRCR_SYSRESETREQ        ( 1 <<  2) /* SCB->AIRCR.SYSRESETREQ : System Reset Request               */
#define AIRCR_VECTRESET          ( 1 <<  0) /* SCB->AIRCR.VECTRESET   : Core Reset Request                 */

/* --- Debug Control Block (DCB) --- */
#define DHCSR_ADDR               0xE000EDF0 /* DCB->DHCSR: Debug Halting Control and Status Register */
#define DHCSR_DBGKEY_VAL     (0xA05F << 16) /* DCB->DHCSR.DBGKEY : Must write 0xA05F to DBGKEY to enable write
                                             * accesses to bits[15:0] */
#define DHCSR_S_RESET_ST         ( 1 << 25) /* DCB->DHCSR.S_RESET_ST: Reset sticky status. Indicates whether the PE
                                             * has been reset since the last read of the DHCSR.   */
#define DHCSR_S_SLEEP            ( 1 << 18) /* DCB->DHCSR.S_SLEEP   : Is processor sleeping       */
#define DHCSR_S_HALT             ( 1 << 17) /* DCB->DHCSR.S_HALT    : Is processor in Debug state */
#define DHCSR_C_HALT             ( 1 <<  1) /* DCB->DHCSR.C_HALT    : Processor halt bit          */
#define DHCSR_C_DEBUGEN          ( 1 <<  0) /* DCB->DHCSR.C_DEBUGEN : Halting debug enable bit    */
                                            /* (DBGKEY|C_HALT|C_DEBUGEN = 0xA05F0003) */
```

Appendix C: Code example

```
#define DCRSR_ADDR              0xE000EDF4 /* DCB->DCRSR: Debug Core Register Selector Register */
#define DCRSR_REGWnR            ( 1 << 16) /* DCB->DCRSR.REGWnR: Specifies the access type for the transfer
                                            * ('0' - Read, '1' - Write)*/
#define DCRSR_REGSEL_MSK        0x0000007F /* DCB->DCRSR.REGSEL: Specifies the ARM core register, special-purpose
                                            * register, or Floating-point extension register */
#define DCRSR_REGSEL_xPSR             0x10 /* DCB->DCRSR.REGSEL = xPSR */
#define DCRSR_REGSEL_MSP              0x11 /* DCB->DCRSR.REGSEL = Main stack pointer, MSP */
#define DCRSR_REGSEL_PC               0x0F /* DCB->DCRSR.REGSEL = PC / DebugReturnAddress */
#define DCRDR_ADDR              0xE000EDF8 /* DCB->DCRDR: Debug Core Register Data Register */
#define DEMCR_ADDR              0xE000EDFC /* DCB->DEMCR: Debug Exception and Monitor Control Register */
#define DEMCR_VC_CORERESET      ( 1 <<  0) /* DCB->DEMCR.VC_CORERESET: Reset Vector Catch.
                                            * Halt running system if Core reset occurs. */
#define DEMCR_TRCENA            ( 1 << 24) /* DCB->DEMCR.TRCENA: Global enable for all DWT and ITM features */
#define xPSR_T                  ( 1 << 24) /* xPSR.T: Thumb bit */
#define DSCSR_ADDR              0xE000EE08 /* DCB->DSCSR: Debug Security Control and Status Register */
#define DSCSR_CDS               ( 1 << 16) /* DCB->DSCSR.CDS: Current Domain Secure */

/*******************************************************************************
 * SWJ state switching sequences
 ******************************************************************************/

/* SWD to DORMANT - standard ARM command to switch SWJ-DP from SWD to dormant state:
 * 1) Send at least 50 SWCLKTCK cycles with SWDIOTMS HIGH. This sequence ensures that the SWD
 *    interface is in the reset state. The target only detects the SWD-to-DS sequence when it is
 *    in the reset state. Note: Fifty-six cycles will be used here to align subsequent data.
 * 2) Send the 16-bit SWD-to-DS select sequence on SWDIOTMS. This sequence can be represented as either:
 *    - 0x3DC7 transmitted MSB first.
 *    - 0xE3BC transmitted LSB first. */
static const uint8_t bSWD_to_DORMANT_len = 9 * 8;
static const uint8_t bSWD_to_DORMANT[] = {
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
  0xBC, 0xE3
};

/* JTAG to DORMANT - standard ARM command to switch SWJ-DP from JTAG to dormant state:
 * 1) Send at least five SWCLKTCK cycles with SWDIOTMS HIGH. This sequence places the JTAG TAP state
 *    machine into the Test-Logic-Reset state, and selects the IDCODE instruction.
 *    Note: Eight cycles will be used here to align subsequent data.
 * 2) Send the recommended 31-bit JTAG-to-DS select sequence on SWDIOTMS.
 *    This sequence can be represented as either:
 *    - 0x2EEEEEE6 transmitted MSB first, that is, starting from bit 30.
 *    - 0x33BBBBBA transmitted LSB first. */
static const uint8_t bJTAG_to_DORMANT_len = 5 * 8;
static const uint8_t bJTAG_to_DORMANT[] = {
  0xFF,
  0xBA, 0xBB, 0xBB, 0x33
};

/* DORMANT to SWD - standard ARM command to switch SWJ-DP from dormant state to SWD:
 * 1) Send at least eight SWCLKTCK cycles with SWDIOTMS HIGH. This sequence ensures that the target is
 *    not in the middle of detecting a Selection Alert sequence. The target is permitted to detect the
 *    Selection Alert sequence even if this 8-cycle sequence is not present.
 * 2) Send the 128-bit Selection Alert sequence on SWDIOTMS. This sequence can be represented as either:
 *    - 0x49CF9046 A9B4A161 97F5BBC7 45703D98 transmitted MSB first.
 *    - 0x19BC0EA2 E3DDAFE9 86852D95 6209F392 transmitted LSB first.
 * 3) Send four SWCLKTCK cycles with SWDIOTMS LOW.
 * 4) Send 16-bit Arm CoreSight SW-DP activation code sequence on SWDIOTMS.
 *    This sequence can be represented as either:
 *    - 0x58 transmitted MSB first.
 *    - 0x1A transmitted LSB first.
 * 5) Send a sequence to place the target into a known state - at least 50 SWCLKTCK cycles with SWDIOTMS HIGH.
 *    This sequence ensures that the SWD interface is in the line reset state.
 * 6) Send at least 2 idle with SWDIOTMS LOW */
static const uint8_t bDORMANT_to_SWD_len = 25 * 8;
static const uint8_t bDORMANT_to_SWD[] = {
  0xFF,
  0x92, 0xF3, 0x09, 0x62,
  0x95, 0x2D, 0x85, 0x86,
  0xE9, 0xAF, 0xDD, 0xE3,
  0xA2, 0x0E, 0xBC, 0x19,
  0xA0, 0xF1,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
  0x3F
};
```

```
/* DORMANT to JTAG - standard ARM command to switch SWJ-DP from dormant state to JTAG:
 * 1) Send at least eight SWCLKTCK cycles with SWDIOTMS HIGH. This sequence ensures that the target is
 *    not in the middle of detecting a Selection Alert sequence. The target is permitted to detect the
 *    Selection Alert sequence even if this 8-cycle sequence is not present.
 * 2) Send the 128-bit Selection Alert sequence on SWDIOTMS.
 *    This sequence can be represented as either:
 *    - 0x49CF9046 A9B4A161 97F5BBC7 45703D98 transmitted MSB first.
 *    - 0x19BC0EA2 E3DDAFE9 86852D95 6209F392 transmitted LSB first.
 * 3) Send four SWCLKTCK cycles with SWDIOTMS LOW.
 * 4) Send 16-bit Arm CoreSight SW-DP activation code sequence on SWDIOTMS.
 *    This sequence can be represented as either
 *    - 0x50 transmitted MSB first.
 *    - 0x0A transmitted LSB first.
 * 5) Send a sequence to place the target into a known state: four SWCLKTCK cycles with SWDIOTMS LOW
 *    to ensure that the TAP state machine is in the Run-Test/Idle state. Then at least five SWCLKTCK
 *    cycles with SWDIOTMS HIGH to ensure that the TAP state machine is in the Test-Logic/Reset state */
static const uint8_t bDORMANT_to_JTAG_len = 20 * 8;
static const uint8_t bDORMANT_to_JTAG[] = {
  0xFF,
  0x92, 0xF3, 0x09, 0x62,
  0x95, 0x2D, 0x85, 0x86,
  0xE9, 0xAF, 0xDD, 0xE3,
  0xA2, 0x0E, 0xBC, 0x19,
  0xA0, 0x00,
  0xFF
};
```

8.3 Memory access and polling functions

8.3.1 ReadAPv2

```
/*****************************************************************
* Reads MEM-AP register of the APv2 architecture (CoreSight SoC-600)
*
* Return value
*   0  SUCCEEDED
*   1  FAILED
*/
static int ReadAPv2(uint8_t apNum, uint32_t regOffset, uint32_t *value) {
  int result;
  uint32_t reg_addr;         /* Effective AP's register address */
  uint32_t select_reg_value; /* DP->SELECT value for access to AP's register with given offset */
  uint32_t reg_index;        /* Bits[3:2] of the address, that are used to select a specific register in a bank,
                              * are provided with APACC transactions */
                             /* Used as RegIndex in ReadDAP/WriteDAP functions */
  reg_addr         = AP_ADDR[apNum] + regOffset;
  select_reg_value = reg_addr & DP_SELECT_MSK;
  reg_index        = reg_addr & AP_REG_A3A2_MSK;
  /* Update DP->SELECT value if needed */
  if (select_reg_value != _DP_SELECT_LAST) {
    result = WriteDAP(DP_REG_SELECT, ACC_DP, select_reg_value);
    if (SUCCEEDED(result)) {
      _DP_SELECT_LAST = select_reg_value;
    }
  } else {
    result = RESULT_OK;
  }
  /* Read AP register value */
  if (SUCCEEDED(result)) {
    result = ReadDAP(reg_index, ACC_AP, value);
  }

  return result;
}
```

8.3.2 WriteAPv2

```
/*****************************************************************
* Writes MEM-AP register of the APv2 architecture (CoreSight SoC-600)
*
* Return value
*   0  SUCCEEDED
```

```
*   1  FAILED
*/
static int WriteAPv2(uint8_t apNum, uint32_t regOffset, uint32_t value) {
  int result;
  uint32_t reg_addr;         /* Effective AP's register address */
  uint32_t select_reg_value; /* DP->SELECT value for access to AP's register with given offset */
  uint32_t reg_index;        /* Bits[3:2] of the address, that are used to select a specific register in a bank,
                              * are provided with APACC transactions */
                             /* Used as RegIndex in ReadDAP/WriteDAP functions */
  reg_addr         = AP_ADDR[apNum] + regOffset;
  select_reg_value = reg_addr & DP_SELECT_MSK;
  reg_index        = reg_addr & AP_REG_A3A2_MSK;
  /* Update DP->SELECT value if needed */
  if (select_reg_value != _DP_SELECT_LAST) {
    result = WriteDAP(DP_REG_SELECT, ACC_DP, select_reg_value);
    if (SUCCEEDED(result)) {
      _DP_SELECT_LAST = select_reg_value;
    }
  } else {
    result = RESULT_OK;
  }
  /* Read AP register value */
  if (SUCCEEDED(result)) {
    result = WriteDAP(reg_index, ACC_AP, value);
  }

  return result;
}
```

8.3.3 ReadMem

```
/*****************************************************************************
 *  Reads 32-bit value from provided memory address.
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int ReadMem(uint8_t apNum, uint32_t address, uint32_t *value) {
  int result;
  /* AP.TAR <- address */
  result = WriteAPv2(apNum, APV2_REG_TAR, address);
  if (SUCCEEDED(result)) {
    /* AP.DRW -> value */
    result = ReadAPv2(apNum, APV2_REG_DRW, value);
  }

  return result;
}
```

8.3.4 WriteMem

```
/*****************************************************************************
 *  Writes uint32_t value to provided memory address.
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int WriteMem(uint8_t apNum, uint32_t address, uint32_t value) {
  int result;
  /* AP.TAR <- address */
  result = WriteAPv2(apNum, APV2_REG_TAR, address);
  if (SUCCEEDED(result)) {
    /* AP.DRW <- value */
    result = WriteAPv2(apNum, APV2_REG_DRW, value);
  }

  return result;
}
```

8.3.5 PollMem

```
/*****************************************************************************
 *  Polls for the expected bit-field value in given register
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error/Timeout
 */
static int PollMem(uint32_t regAddr, uint32_t fieldMsk, uint32_t rsh, uint32_t expectedValue, int timeout,
                   uint32_t sleepBetweenPolling, uint32_t *regValue) {
  int result;
  int t;
  int tDelta;
  tDelta = -1;
  result = RESULT_ERR;

  t = SysGetTimeMs();
  do {
    result = ReadMem(AP_TO_USE, regAddr, regValue);
    BREAK_IF_FAILED(result);
    if (((*regValue & fieldMsk) >> rsh) == expectedValue) {
      result = RESULT_OK;
      break;
    }
    /* Sleep between polling - let the CPU do its job and avoid too much garbage on SWD */
    if ((sleepBetweenPolling > 0) && (tDelta >= 0 /* not first iteration*/)) {
      SysSleepMs(sleepBetweenPolling);
    }
    tDelta = SysGetTimeMs() - t;
  } while (tDelta < timeout);

  return result;
}
```

8.4 DAP security low-level functions

8.4.1 SecureAddr

```
/*********************************************************************
* Returns secure alias for a given address (sets bit[28])
*
*/
uint32_t SecureAddr(uint32_t address) {
  if (_DOMAIN_SECURE) { address |= (1 << 28); }
  else (_DOMAIN_SECURE) { address &= (~(1 << 28)); }
  return address;
}
```

8.4.2 ReadAndInitSecure

```
/*********************************************************************
* Reads "Debug Security Control and Status Register" (DSCSR)
* and checks "Current Domain Secure" (CDS) bit
*
* Return value
*   >= 0  O.K.
*   <  0  Error
*/
int ReadAndInitSecure (void) {
  int result;
  uint32_t v;
  _DOMAIN_SECURE = 0;

  result = ReadMem(AP_TO_USE, DSCSR_ADDR, &v);
  if (SUCCEEDED(result)) {
    _DOMAIN_SECURE = v & DSCSR_CDS;
  }
  else {
    printf("Warning, DSCSR register is unaccesible. Assumed Non-secure CDS");
  }
```

```
  if (_DOMAIN_SECURE != 0) {
    printf("Current domain secure state: Secure");
  }
  else {
    printf("Current domain secure state: Non-secure");
  }

  /* Read current CSW value */
  result = ReadAPv2(AP_TO_USE, APV2_REG_CSW, &v);
  if (SUCCEEDED(result)) {
      /* Clean PROT bits */
      v &= (~AP_CSW_PROT_MSK);
      /* Apply proper PROT value */
      if (_DOMAIN_SECURE) {
          v |= (AP_CSW_PROT_VAL & AP_CSW_PROT_MSK);
      }
      else {
          v |= (AP_CSW_PROT_NS_VAL & AP_CSW_PROT_MSK);
      }
      /* Write back to CSW */
      result = WriteAPv2(AP_TO_USE, APV2_REG_CSW, v);
  }
  return result;

}
```

8.5 ARM Core control and register access functions

8.5.1 ReadCoreReg

```
/*****************************************************************
* Reads ARM core register, special-purpose register, or Floating-point extension register
* CPU must be halted for this operation
*
* Return value
*   >= 0  O.K.
*   <  0  Error
*/
int ReadCoreReg(uint32_t regsel, uint32_t* value) {

  /* DCRSR (0xE000EDF4) <- (REGWnR == read) | REGSEL */
  int result = WriteMem(AP_TO_USE, DCRSR_ADDR, (regsel & DCRSR_REGSEL_MSK));
  if (SUCCEEDED(result)) {
    /* DCRDR (0xE000EDF8) -> value */
    result = ReadMem(AP_TO_USE, DCRDR_ADDR, value);
  }

 return result;
}
```

8.5.2 WriteCoreReg

```
/*****************************************************************
* Writes ARM core register, special-purpose register, or Floating-point extension register
* CPU must be halted for this operation
*
* Return value
*   >= 0  O.K.
*   <  0  Error
*/
int WriteCoreReg(uint32_t regsel, uint32_t value) {

  /* DCRDR (0xE000EDF8) <- value */
  int result = WriteMem(AP_TO_USE, DCRDR_ADDR, value);
  if (SUCCEEDED(result)) {
    /* DCRSR (0xE000EDF4) <- (REGWnR == write) | REGSEL */
    result = WriteMem(AP_TO_USE, DCRSR_ADDR, (DCRSR_REGWnR | (regsel & DCRSR_REGSEL_MSK)));
  }

  return result;
}
```

8.5.3 EnableCPU

```
/*****************************************************************
*  Enable CPU
*
*  Return value
*    >= 0  O.K. (CPU is enabled)
*    <  0  Error/Timeout
*/
int EnableCPU(void) {
  LOG_ENTRY();
  int result;
  uint32_t CPU_CTL_ADDR;
  uint32_t vtbl_addr;
  uint32_t entry_addr;
  uint32_t sp;
  uint32_t v;
  uint32_t v1;

  /* Set registers addresses, depending on the current CPU */
  if (AP_TO_USE == AP_CM33) {
    CPU_CTL_ADDR = SecureAddr(CPUSS_CM33_CTL);
  } else if (AP_TO_USE == AP_CM55) {
    CPU_CTL_ADDR = SecureAddr(CPUSS_CM55_CTL);
  } else {
    LOG_EXIT(RESULT_ERR);
    return RESULT_ERR;
  }
  fprintf(stderr, "CPU_CTL_ADDR = 0x%08" PRIx32 "\n", CPU_CTL_ADDR);
  /* Check CPU_WAIT bit over CM33 AP */
  result = ReadMem(AP_CM33, CPU_CTL_ADDR, &v);
  if (SUCCEEDED(result)) {
    if ((v & MSK_CPUSS_CMx_CTL_CPU_WAIT) != 0) {
      fprintf(stderr, "CPU is in WAIT state after the reset, resuming...\n");

      /* Check VTOR and set it to some *safe* place if not set by the boot */
      result = ReadMem(AP_TO_USE, VTOR_ADDR, &vtbl_addr);
      if (SUCCEEDED(result)) {
        fprintf(stderr, "VTOR: 0x%08" PRIx32 "\n", vtbl_addr);
        if ((vtbl_addr >= SecureAddr(SRAM_NS_BASE)) && (vtbl_addr < SecureAddr(SRAM_NS_BASE + SRAM_SIZE))) {
        } else {
          fprintf(stderr, "      replacing with 0x%08" PRIx32 "\n", SecureAddr(SRAM_NS_BASE));
          vtbl_addr = SecureAddr(SRAM_NS_BASE);
          result    = WriteMem(AP_TO_USE, VTOR_ADDR, vtbl_addr);
        }
      }

      if (FAILED(result)) {
        LOG_EXIT(result);
        return result;
      }

      /* Check reset handler and set it to some *safe* place if not set by the boot */
      result = ReadMem(AP_TO_USE, vtbl_addr + 4, &entry_addr);
      if (SUCCEEDED(result)) {
        fprintf(stderr, "Reset handler: 0x%08" PRIx32 "\n", entry_addr);
        if ((entry_addr >= SecureAddr(SRAM_NS_BASE)) && (entry_addr < SecureAddr(SRAM_NS_BASE + SRAM_SIZE)) &&
            (entry_addr != vtbl_addr)) {
          /* No modification of the entry_addr required */
        } else {
          /* Use secure address. Force LSB to 1 to avoid LOCKUP due to the THUMB bit not being set. */
          entry_addr = SecureAddr(SRAM_LOOP_ADDR) | 1;
          fprintf(stderr, "      replacing with 0x%08" PRIx32 "\n", entry_addr);
          result = WriteMem(AP_TO_USE, vtbl_addr + 4, entry_addr);
        }
      }

      if (FAILED(result)) {
        LOG_EXIT(result);
        return result;
      }

      /* Check reset handler code */
      result = ReadMem(AP_TO_USE, entry_addr & 0xFFFFFFFC, &v);
```

```
  if (SUCCEEDED(result)) {
    fprintf(stderr, "Reset handler code: 0x%08" PRIx32 "\n", v);
    if (v == 0) {
      fprintf(stderr, "      replacing with infinite loop 0x%08" PRIx32 "\n", LOOP_CODE);
      result = WriteMem(AP_TO_USE, entry_addr & 0xFFFFFFFC, LOOP_CODE);
    }
  }

  if (FAILED(result)) {
    LOG_EXIT(result);
    return result;
  }

  /* Check SP */
  result = ReadMem(AP_TO_USE, vtbl_addr, &sp);
  if (SUCCEEDED(result)) {
    fprintf(stderr, "SP by VTOR = 0x%08" PRIx32 "\n", sp);
    if ((sp >= SecureAddr(SRAM_NS_BASE)) && (sp < SecureAddr(SRAM_NS_BASE + SRAM_SIZE)) && (sp != vtbl_addr)) {
      /* No modification of the entry_addr required */
    } else {
      sp = SecureAddr(SRAM_DBG_ADDR);
      fprintf(stderr, "      replacing with 0x%08" PRIx32 "\n", sp);
      result = WriteMem(AP_TO_USE, vtbl_addr, sp);
    }
  }

  /* Check DEMCR against DEMCR_TRCENA and DEMCR_VC_CORERESET bits */
  if (SUCCEEDED(result)) {
    result = ReadMem(AP_TO_USE, DEMCR_ADDR, &v);
    if (SUCCEEDED(result)) {
      if ((v & (DEMCR_TRCENA | DEMCR_VC_CORERESET)) != (DEMCR_TRCENA | DEMCR_VC_CORERESET)) {
        v |= DEMCR_TRCENA | DEMCR_VC_CORERESET;
        result = WriteMem(AP_TO_USE, DEMCR_ADDR, v);
      }
    }
  }

  /* Write DEBUG_ENABLED bit to DHCSR */
  if (SUCCEEDED(result)) {
    result = WriteMem(AP_TO_USE, DHCSR_ADDR, DHCSR_DBGKEY_VAL | DHCSR_C_DEBUGEN);
  }

  /* Clear CPU_WAIT bit */
  if (SUCCEEDED(result)) {
    fprintf(stderr, "Clearing CPU_WAIT bit\n");
    /* Access to CPU_CTL_ADDR over AP_CM33 only */
    result = WriteMem(AP_CM33, SecureAddr(CPU_CTL_ADDR), 0);
  }

  /* Wait for core halt */
  if (SUCCEEDED(result)) {
    v      = DHCSR_S_RESET_ST;
    result = PollMem(DHCSR_ADDR, v, 0, v, TIMEOUT_LISTEN_WND, 1, &v1);
    if (SUCCEEDED(result)) {
      v      = DHCSR_S_HALT | DHCSR_C_DEBUGEN;
      result = PollMem(DHCSR_ADDR, v, 0, v, TIMEOUT_LISTEN_WND, 1, &v1);
    }
  }

  /* Read xPSR register, set the thumb bit, and restore modified value to xPSR register */
  if (SUCCEEDED(result)) {
    result = ReadCoreReg(DCRSR_REGSEL_xPSR, &v);
    if (SUCCEEDED(result)) {
      result = WriteCoreReg(DCRSR_REGSEL_xPSR, (v | xPSR_T));
    }
  }

  /* Set PC and SP*/
  if (SUCCEEDED(result)) {
    result = WriteCoreReg(DCRSR_REGSEL_PC, entry_addr);
  }
  if (SUCCEEDED(result)) {
    result = WriteCoreReg(DCRSR_REGSEL_MSP, sp);
  }
```

```
      /* Run infinite loop in SRAM */
      if (SUCCEEDED(result)) {
        result = WriteMem(AP_TO_USE, DHCSR_ADDR, DHCSR_DBGKEY_VAL | DHCSR_C_DEBUGEN);
      }

    } else {
      fprintf(stderr, "CPU is not in WAIT state...\n");
    }
  } else {
    fprintf(stderr, "The CPU_WAIT bit cannot be checked due to CPU_CTL_ADDR is inaccessible.\n");
  }

  LOG_EXIT(result);
  return result;
}
```

8.5.4 HaltCPU

```
/*****************************************************************************
 *  Enables debug and halts the CPU using the DHCSR register
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int HaltCPU(void) {
  int result;
  uint32_t v;

  /* Enable debug, and halt the CPU using the DHCSR register: 0xE000EDF0 <- 0xA05F0003 */
  result = WriteMem(AP_TO_USE, DHCSR_ADDR, DHCSR_DBGKEY_VAL | DHCSR_C_HALT | DHCSR_C_DEBUGEN);

  /* Check S_HALT bit [17] in DHCSR register (@0xE000EDF0) */
  if (SUCCEEDED(result)) {
    result = PollMem(DHCSR_ADDR, DHCSR_S_HALT, 0, DHCSR_S_HALT, TIMEOUT_HALT_CPU, 0, &v);
  }

  return result;
}
```

8.5.5 ResumeCPU

```
/*****************************************************************************
 *  Enables debug and resumes the CPU using the DHCSR register
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int ResumeCPU(void) {
  int result;
  uint32_t v;

  /* Resume CPU (keeping debug enabled) using the DHCSR register: 0xE000EDF0 <- 0xA05F0001 */
  result = WriteMem(AP_TO_USE, DHCSR_ADDR, DHCSR_DBGKEY_VAL | DHCSR_C_DEBUGEN);

  /* Check S_HALT (bit[17] in DHCSR register @0xE000EDF0) is cleared */
  if (SUCCEEDED(result)) {
    result = PollMem(DHCSR_ADDR, DHCSR_S_HALT, 0, 0, TIMEOUT_HALT_CPU, 0, &v);
  }

  return result;
}
```

8.6 DAP initialization functions

8.6.1 DAP_Handshake

```
/*****************************************************************************
 * Handshake: wait for debug interface becomes enabled after device reset. In the worst case, when
 * the boot code performs application HASH verification, boot time is around 2000ms and depends on
```

```
 * CPU clock used by boot code. For PowerCycle, timeout depends on the design schematic and must be
 * longer.
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int DAP_Handshake(uint32_t timeout) {
  uint32_t v;
  int tDelta;
  int t      = SysGetTimeMs();
  int result = RESULT_ERR;

  do {
    if (IsJTAG()) {
      /* If the interface was left in SWD by a previous session,
       * try switching to JTAG once over the dormant state. */
      SWJSequence(&bSWD_to_DORMANT[0], bSWD_to_DORMANT_len);
      SWJSequence(&bDORMANT_to_JTAG[0], bDORMANT_to_JTAG_len);
    } else {
      /* Switch to SWD over the dormant state */
      SWJSequence(&bJTAG_to_DORMANT[0], bJTAG_to_DORMANT_len);
      SWJSequence(&bDORMANT_to_SWD[0], bDORMANT_to_SWD_len);
    }
    v = 0;
    ReadDAP(DP_REG_DPIDR, ACC_DP, &v);
    /* DAP is responsive if we can read IDCODE */
    if ((v & DP_IDCODE_MSK) == DP_IDCODE_VAL) {
      _DP_SELECT_LAST = 0; /* Zeroing last used value of DP->SELECT */
      result = RESULT_OK;
      break;
    }
    tDelta = SysGetTimeMs() - t;
  } while (tDelta < timeout); /* Timeout reached? */

  return result;
}
```

8.6.2 DAP_Init

```
/*****************************************************************************
 *  Initialize the Debug Port for programing operations. Accepts Access Port number as input:
 *   0 - System AP; 1 - CM33 AP.
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int DAP_Init(uint8_t apNum) {
  LOG_ENTRY();
  int result;
  uint32_t reg_addr;         /* Effective AP's reg addr */
  uint32_t select_reg_value; /* DP->SELECT value for acces to AP's register with given offset */
  uint32_t reg_index;        /* Bits[3:2] of the address, that are used to select a specific register in a bank.
                              * Used as RegIndex in ReadDAP/WriteDAP functions */

  /* Power up DAP using DP.CTRL/STAT: [30]:CSYSPWRUPREQ, [28]:CDBGPWRUPREQ
   * And clear sticky errors:
   * - SWD:  Using AP.ABORT register
   * - JTAG: Using DP.CTRL/STAT: [5]:STICKYERR, [4]:STICKYCMP, [1]:STICKYORUN
   *   For JTAG, sticky error bits are read-write enabled and writing '1' to these bits clears associated sticky
errors.
   *   For SWD, these bits are read-only and to clean the sticky errors, you should write to appropriate bits of
   * DP.ABORT register */
  if (IsJTAG()) { /* JTAG */
    result = WriteDAP(DP_REG_CTRL_STAT, ACC_DP,
                      DP_CTRL_STAT_CSYSPWRUPREQ | DP_CTRL_STAT_CDBGPRWUPREQ | DP_CTRL_STAT_STICKYERR /* 0x50000020
*/);
  } else { /* SWD */
    result =
        WriteDAP(DP_REG_ABORT, ACC_DP,
                 DP_ABORT_ORUNERRCLR | DP_ABORT_WDERRCLR | DP_ABORT_STKERRCLR | DP_ABORT_STKCMPCLR /* 0x0000001E
*/);
    if (SUCCEEDED(result)) {
```

```
      result = WriteDAP(DP_REG_CTRL_STAT, ACC_DP, DP_CTRL_STAT_CSYSPWRUPREQ | DP_CTRL_STAT_CDBGPRWUPREQ);  //
0x50000000
    }
  }

  /* Initialize DP->SELECT and AP->CSW */
  if (SUCCEEDED(result)) {
    /* Get effective address of CSW register */
    reg_addr         = AP_ADDR[apNum] + APV2_REG_CSW;
    select_reg_value = reg_addr & DP_SELECT_MSK;
    reg_index        = reg_addr & AP_REG_A3A2_MSK;

    result = WriteDAP(DP_REG_SELECT, ACC_DP, select_reg_value); /* DP->SELECT <- select_reg_value */
    if (SUCCEEDED(result)) {
      _DP_SELECT_LAST = select_reg_value; /* Update last used value of DP->SELECT */

      /* Set CSW (DbgSwEnable=0, Prot=0x0B, SPIDEN=0, Mode=0x0, TrInProg=0, DeviceEn=0, AddrInc=Auto-increment off,
       * Size=Word (32 bits)) */
      /* Note: Set Prot bits in DAP CSW register, because of no access to CPU registers via M33 or M55 AP without
these
       * bits */
      if (_DOMAIN_SECURE != 0) {
        result = WriteDAP(reg_index, ACC_AP, AP_CSW_PROT_VAL | AP_CSW_SIZE_WORD); /* 0x0B000002 */
      } else {
        result = WriteDAP(reg_index, ACC_AP, AP_CSW_PROT_NS_VAL | AP_CSW_SIZE_WORD); /* 0x4B000002 */
      }
    }
  }

  LOG_EXIT(result);
  return result;
}
```

8.6.3 DAP_HandshakeAndInit

```
/*****************************************************************************
 *  Performs Handshake and Initializes the Debug Port Accepts Access Port number as input:
 *  0 – System AP; 1 – CM33 AP.
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int DAP_HandshakeAndInit(uint8_t apNum, uint32_t timeout) {
  int result;

  result = DAP_Handshake(timeout);
  if (SUCCEEDED(result)) {
    result = DAP_Init(apNum);
    if (SUCCEEDED(result)) {}
  }

  return result;
}
```

8.6.4 DAP_ScanAP

```
/*****************************************************************************
 *  Scans the Access Ports for the first available with CPU registers access.
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
int DAP_ScanAP(uint8_t *apNum) {
  uint32_t v;
  uint8_t currAP;
  int result;

  /* Try all possible Access Ports */
  currAP = AP_SYS;
  while (currAP < AP_MAX) {
    if (currAP != AP_SYS) {
```

```
      /* Initializes DAP and selects Access Port with provided number */
      result = DAP_HandshakeAndInit(currAP, TIMEOUT_HANDSHAKE);
      if (SUCCEEDED(result)) {
        /* Try to read CPUID register @0xE000ED00 */
        result = ReadMem(currAP, CPUID_ADDR, &v);
        /* If the CPUID Implementer is ARM, the Access Port is correct (we have access to the ARM
         * registers) */
        if (SUCCEEDED(result) && ((v & 0xFF000000) == 0x41000000)) {
          *apNum = currAP;
          LOG_EXIT(result);
          return RESULT_OK;
        }
      }
    }
    currAP += 1;
  }

  return RESULT_ERR;
}
```

8.7 System reset

8.7.1 Reset

```
/*******************************************************************************
 *  Resets the device using either of:
 *    1. Hardware reset by toggling XRES pin
 *    2. Software reset by setting the RES_SOFT_CTL.TRIGGER_SOFT bit
 *    3. Software reset by setting the AIRCR.SYSRESETREQ bit
 *    4. Software reset by setting the AIRCR.VECTRESET bit
 *    5. Software reset by setting the DP->CTRL/STAT.CDBGRSTREQ bit
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int Reset(uint8_t rstType, uint8_t apNum) {
  LOG_ENTRY();

  uint32_t v;
  int result;
  result = RESULT_ERR;

  /* Attempt to reset the device with different methods
   * Note1: do not check OK/WAIT/FAULT ACKs for the data write phase since the target immediately
   * reboots.
   * Note2: caller code needs to do Handshake and DAP Init after reset or in case of failure */

  /* 1. Hardware reset by toggling XRES pin */
  if ((rstType & RST_TYPE_XRES) != 0) {
    SetXRES(0);     /* nRESET == LOW */
    SysSleepMs(50); /* Make sure that device recognizes the reset */
    SetXRES(1);     /* nRESET == HIGH */
    result = RESULT_OK;
  }

  /* 2. Software reset by setting the RES_SOFT_CTL.TRIGGER_SOFT bit:
   * This type of software reset can work via SYS-AP, so it is more preferable vs. SYSRESETREQ */
  if (FAILED(result) && ((rstType & RST_TYPE_RES_SOFT_CTL) != 0)) {
    /* AP.TAR <- @(SRSS->RES_SOFT_CTL) */
    result = WriteAPv2(apNum, APV2_REG_TAR, SecureAddr(SRSS_RES_SOFT_CTL));
    if (FAILED(result)) {
      result = DAP_HandshakeAndInit(apNum, TIMEOUT_HANDSHAKE);
      if (SUCCEEDED(result)) {
        result = WriteAPv2(apNum, APV2_REG_TAR, SRSS_RES_SOFT_CTL);
      }
    }
    /* AP.DRW <- TRIGGER_SOFT bit */
    if (SUCCEEDED(result)) {
      WriteAPv2(apNum, APV2_REG_DRW, SRSS_RES_SOFT_CTL_TRIG_SOFT);
    }
  }
```

```
  /* 3. Software reset by setting the AIRCR.SYSRESETREQ bit */
  if (FAILED(result) && ((rstType & RST_TYPE_SYSRESETREQ) != 0) && (apNum != AP_SYS)) {
    /* AP.TAR <- @(AIRCR 0xE000ED0C) */
    result = WriteAPv2(apNum, APV2_REG_TAR, AIRCR_ADDR);
    if (FAILED(result)) {
      result = DAP_HandshakeAndInit(apNum, TIMEOUT_HANDSHAKE);
      if (SUCCEEDED(result)) {
        result = WriteAPv2(apNum, APV2_REG_TAR, AIRCR_ADDR);
      }
    }
    /* AP.DRW <- 0x05FA0004 */
    if (SUCCEEDED(result)) {
      WriteAPv2(apNum, APV2_REG_DRW, (AIRCR_VECTKEY_VAL | AIRCR_SYSRESETREQ));
    }
  }

  /* 4. Software reset by setting the AIRCR.VECTRESET bit */
  if (FAILED(result) && ((rstType & RST_TYPE_VECTRESET) != 0) && (apNum != AP_SYS)) {
    /* A debugger must halt the processor before using VECTRESET, otherwise the effect is unpredictable */
    /* Enable debug, and halt the CPU using the DHCSR register: 0xE000EDF0 <- 0xA05F0003 */
    result = WriteMem(apNum, DHCSR_ADDR, DHCSR_DBGKEY_VAL | DHCSR_C_HALT | DHCSR_C_DEBUGEN);
    if (FAILED(result)) {
      result = DAP_HandshakeAndInit(apNum, TIMEOUT_HANDSHAKE);
      if (SUCCEEDED(result)) {
        result = WriteMem(apNum, DHCSR_ADDR, DHCSR_DBGKEY_VAL | DHCSR_C_HALT | DHCSR_C_DEBUGEN);
      }
    }
    if (SUCCEEDED(result)) {
      result = PollMem(DHCSR_ADDR, DHCSR_S_HALT, 0, DHCSR_S_HALT, TIMEOUT_HALT_CPU, 0, &v);
      if (SUCCEEDED(result)) {
        /* AIRCR 0xE000ED0C <- 0x05FA0001 */
        result = WriteMem(apNum, AIRCR_ADDR, AIRCR_VECTKEY_VAL | AIRCR_VECTRESET);
      }
    }
  }

  /* 5. Software reset by setting the DP->CTRL/STAT.CDBGRSTREQ bit.
   * In worst case, if standard software reset via SYSRESETREQ failed, it may mean that the
   * firmware did very bad things disabling the debug pins or AHB_AP access (anything behind the
   * DAP). However, if we still can access the DAP registers, the last thing we could try is to
   * reset the target via DP->CTRL/STAT.CDBGRSTREQ.*/
  if (FAILED(result) && ((rstType & RST_TYPE_CDBGRSTREQ) != 0)) {
    result = DAP_Handshake(TIMEOUT_HANDSHAKE);
    if (SUCCEEDED(result)) {
      WriteDAP(DP_REG_CTRL_STAT, ACC_DP,
               DP_CTRL_STAT_CSYSPWRUPREQ | DP_CTRL_STAT_CDBGPRWUPREQ | DP_CTRL_STAT_CDBGRSTREQ);
    }
  }

  LOG_EXIT(result);
  return result;
}
```

8.8 ROM boot status checking and polling

8.8.1 IsBootIdle

```
/*****************************************************************************
 * Check if device is in WFA (Wait For Action), IDLE or DEAD branches, what is sufficient condition
 * for programming
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int IsBootIdle(uint8_t apNum, uint32_t *stopPolling) {
  int result;
  uint32_t v;
  *stopPolling = 0;

  /* Read SRSS->TST_DEBUG_CTL register and check the WFA bit is set, indicating a special mode with
   * additional restrictions for debugger. Normal programming/debugging is not possible - need reset/acquire */
  result = ReadMem(apNum, SecureAddr(SRSS_BOOT_DLM_CTL), &v);
```

```
  if (SUCCEEDED(result)) {
    if ((v & SRSS_BOOT_DLM_CTL_DEBUG_WFA) != 0) {
      *stopPolling = 1; /* WFA branch, no sense to continue polling - need to do reset/acquire */
      result = RESULT_ERR;
    }
  }

  /* Check the status reported by boot code in RAM.
   * Both, DEAD and IDLE branches are sufficient for programming */
  if (SUCCEEDED(result)) {
    result = ReadMem(apNum, SecureAddr(SRAM_STATUS_ADDR), &v);
    if (SUCCEEDED(result)) {
      if ((v & L1BOOT_ID_MSK) == L1BOOT_ID_SUCCESS) {
        if ((v & L1BOOT_STATUS_MSK) != L1BOOT_IDLE_BRANCH_REACHED) {
          result = RESULT_ERR; /* Not IDLE branch */
        }
      }
      else {
        if ((v & L1BOOT_ID_MSK) != L1BOOT_ID_FAIL) {
          result = RESULT_ERR; /* No status word */
        }
      }
    }
  }

  return result;
}
```

8.8.2 WaitForBootIdle

```
/*******************************************************************************
 *  Waits for the device to be in IDLE or DEAD branches
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int WaitForBootIdle(uint8_t apNum, int timeout) {
  int result;
  int t;
  int tDelta;
  uint32_t stopPolling;
  tDelta = -1;

  result = IsBootIdle(apNum, &stopPolling);
  if (FAILED(result) && (result != RESULT_ERR_CRITICAL) && (stopPolling == 0)) {
    t = SysGetTimeMs();
    do {
      /* Sleep between polling - let the target do its job and avoid too much garbage on SWD */
      SysSleepMs(10);
      result = IsBootIdle(apNum, &stopPolling);
      if (SUCCEEDED(result) || (result == RESULT_ERR_CRITICAL) || (stopPolling != 0)) {
        /* No sense to wait if target is in CORRUPTED state (result == RESULT_ERR_CRITICAL) or in
         * WFA branch or when the application is already launched (stopPolling != 0) */
        break;
      }
      tDelta = SysGetTimeMs() - t;
    } while (tDelta < timeout);
  }

  return result;
}
```

8.9 Acquisition helper functions

8.9.1 GetVectorTableData

```
/*******************************************************************************
 *  Gets Reset Address and Initial SP values from application Vector Table
 *
 *  Return value
 *    >= 0  O.K.
```

```
 *    <  0  Error
 */
static int GetVectorTableData(uint32_t *resetAddress, uint32_t *sp) {
  int result;
  uint32_t v;
  uint32_t vtBase;

  *resetAddress = 0;
  *sp           = 0;

  /* Check Vector Table base address for Cortex core.
   * Note: Zero in Vector Table base register or in reset address (reset handler + 4) likely
   * indicates that the target is in preproduction state, so the ROM boot code debugging is enabled. */
  result = ReadMem(AP_TO_USE, VTOR_ADDR, &vtBase);
  if (SUCCEEDED(result) {
    if (vtBase == 0) {
      vtBase = SecureAddr(SRAM_S_BASE);
    }

    if (SUCCEEDED(result) && ((vtBase & ERR_ADDR_MSK) != ERR_ADDR_MSK)) {
      /* Get Reset Address from Vector Table */
      result = ReadMem(AP_TO_USE, vtBase + 4, &v);
      if (SUCCEEDED(result) && ((v & ERR_ADDR_MSK) != ERR_ADDR_MSK)) {
        *resetAddress = v;

        /* Get Initial SP value from Vector Table */
        result = ReadMem(AP_TO_USE, vtBase, &v);
        if (SUCCEEDED(result) && (v != 0) && ((v & ERR_ADDR_MSK) != ERR_ADDR_MSK)) {
          *sp = v;
        } else {
          result = RESULT_ERR;
        }
      } else {
        result = RESULT_ERR;
      }
    } else {
      result = RESULT_ERR;
    }
  }
  return result;
}
```

8.9.2 SetPCandSPFromVectorTable

```
/*******************************************************************************
 *  Sets PC and SP getting the values from Vector Table
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int SetPCandSPFromVectorTable(void) {
  int result;
  uint32_t v;
  uint32_t pc;
  uint32_t sp;

  /* Get PC and SP for the application in flash */
  result = GetVectorTableData(&pc, &sp);
  if (SUCCEEDED(result) && ((pc & ERR_ADDR_MSK) != ERR_ADDR_MSK) && ((sp & ERR_ADDR_MSK) != ERR_ADDR_MSK)) {
    /* Set PC */
    result = WriteCoreReg(DCRSR_REGSEL_PC, pc);
    if (SUCCEEDED(result)) {
      /* Set MSP */
      result = WriteCoreReg(DCRSR_REGSEL_MSP, sp);
      if (SUCCEEDED(result)) {
        /* Read xPSR register, set the thumb bit, and restore modified value to xPSR register */
        result = ReadCoreReg(DCRSR_REGSEL_xPSR, &v);
        if (SUCCEEDED(result)) {
          result = WriteCoreReg(DCRSR_REGSEL_xPSR, (v | xPSR_T));
        }
      }
    }
  } else {
```

```
    result = RESULT_ERR;
  }

  return result;
}
```

8.10 Acquisition functions

8.10.1 AcquireTestMode

```
/*******************************************************************************
 *  Performs device acquisition in test mode:
 *    1. Pre-reset (do hardware (XRES) or one of the software reset) and connect to the DAP
 *    2. Set TEST_MODE bit in TST_MODE SRSS register
 *    3. Poll for the IDLE status set by boot code in RAM
 *    4. Prepares target for debug
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int AcquireTestMode(uint8_t rstType, uint8_t apNum) {
  LOG_ENTRY();

  int result;
  uint32_t v;

  /* 1. Pre-reset (do hardware (XRES) or one of the software reset) and connect to the DAP
   * ------------------------------------------------------------------------------
   * a. Do hardware (XRES) or one of the software reset. It is critical for Test Mode acquisition,
   * so stop in case of failure.
   * b. Handshake (wait for debug interface to become enabled after device reset), initialize the
   * Debug Port, and select the appropriate Access Port (AP) */
  result = Reset(rstType, apNum);
  if (SUCCEEDED(result)) {
    result = DAP_HandshakeAndInit(apNum, TIMEOUT_HANDSHAKE);
  }

  /* 2. Set TEST_MODE bit in TST_MODE SRSS register
   * ------------------------------------------------------------------------------*/
  if (SUCCEEDED(result)) {
    result = WriteMem(apNum, SecureAddr(SRSS_TST_MODE), SRSS_TST_MODE_TEST_MODE);
    /* Read RDBUFF to make sure that the last AP write actually happens as SW-DP may buffer/delay it
     * until next DAP access*/
    ReadDAP(DP_REG_RDBUFF, ACC_DP, &v);
  }

  /* 3. Poll for the IDLE status set by boot code and check PC points to address in ROM*/
  if (SUCCEEDED(result)) {
    result = WaitForBootIdle(apNum, TIMEOUT_LISTEN_WND);
    if (FAILED(result)) {
      /* CPU can be in WAIT state, so it needs to be enabled */
      result = EnableCPU();
    }
  }

  /* 4. Prepares target for debug
   * ------------------------------------------------------------------------------
   * a. Clear TEST_MODE bit in SRSS->TST_MODE.TEST_MODE register
   * b. Set SP and PC values from the vector table. Needs to be done after Test mode acquisition to
   * withdraw target from IDLE loop Otherwise, such commands as "go" or "step" will not work after
   * acquisition */
  if (SUCCEEDED(result)) {
    result = WriteMem(apNum, SecureAddr(SRSS_TST_MODE), 0);
    if (SUCCEEDED(result) && (apNum != AP_SYS)) {
      SetPCandSPFromVectorTable();
    }
  }

  LOG_EXIT(result);
  return result;
}
```

8.10.2 AcquireVectorCatch

```
/*****************************************************************************
 *  Performs target acquisition using Vector Catch:
 *   1. Pre-reset (do hardware (XRES) or one of the software reset) and connect to the DAP
 *   2. Halt CPU, set DEMCR->VC_CORERESET, and issue software reset
 *   3. Connect to the DAP and check CPU is halted
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
static int AcquireVectorCatch(uint8_t rstType, uint8_t apNum) {
  LOG_ENTRY();

  int result;
  int resultTmp;
  uint32_t v;
  uint32_t v1;

  /* 1. Pre-reset and connect
   * ---------------------------------------------------------------------------
   * a. Pre-reset (do hardware (XRES) or one of the software reset) and connect to the DAP
   *    Pre-reset is not critical for the Vector Catch acquisition,
   *    so do not check for the result and do not stop if it is failed
   * b. Handshake (wait for debug interface to become enabled after device reset),
   *    initialize the Debug Port and select appropriate Access Port (AP) with the CPU access
   * c. Update Current Domain Secure */
  if (apNum == AP_SYS) {
    /* It is not possible to handle CPU state (e.g. breakpoints) via the System Access Port */
    result = RESULT_ERR;
  } else {
    Reset(rstType, apNum);
    result = DAP_HandshakeAndInit(apNum, TIMEOUT_HANDSHAKE);
    if (SUCCEEDED(result)) {
      /* Update Current Domain Secure */
      result = ReadAndInitSecure();
    }
  }

  /* 2. Halt CPU, set DEMCR->VC_CORERESET and issue software reset
   * ---------------------------------------------------------------------------
   * a. Enable debug and halt CPU as quickly as possible right after Reset+Handshake+InitDAP
   *    It is not mandatory to do this quickly, but there is a good chance to stop
   *    in Listen window or at least prevent user application from doing too much "bad" stuff
   * b. Set VC_CORERESET and TRCENA bits in DEMCR register
   * c. Issue software reset*/
  if (SUCCEEDED(result)) {
    result = HaltCPU();
    if (SUCCEEDED(result)) {
      /* Set VC_CORERESET and TRCENA: DEMCR (0xE000EDFC) = 0x01000001 */
      result = WriteMem(apNum, DEMCR_ADDR, DEMCR_TRCENA | DEMCR_VC_CORERESET);
      if (SUCCEEDED(result)) {
        result = Reset(rstType, apNum);
      }
    }
  }

  /* 3. Connect to the DAP and check CPU is halted
   * ---------------------------------------------------------------------------
   * a. Handshake and initialize the Debug Port
   * b. Verify reset indeed occurred and CPU is halted in debug mode
   * c. Verify CPU is halted and in debug mode.
   *    It must be verified in separate step after the reset confirmation to avoid raise conditions.
   * d. Clear VC_CORERESET, but leave TRCENA bit enabled. Do it even in failed scenario */
  if (SUCCEEDED(result)) {
    result = DAP_HandshakeAndInit(apNum, TIMEOUT_HANDSHAKE);
    if (SUCCEEDED(result)) {
      v      = DHCSR_S_RESET_ST;
      result = PollMem(DHCSR_ADDR, v, 0, v, TIMEOUT_LISTEN_WND, 1, &v1);
      if (SUCCEEDED(result)) {
        v      = DHCSR_S_HALT | DHCSR_C_DEBUGEN;
        result = PollMem(DHCSR_ADDR, v, 0, v, TIMEOUT_LISTEN_WND, 1, &v1);
      }
```

```
    }
    resultTmp = WriteMem(apNum, DEMCR_ADDR, DEMCR_TRCENA);  // DEMCR (0xE000EDFC) <- TRCENA
    if (FAILED(resultTmp) && SUCCEEDED(result)) {
      result = RESULT_ERR;
    }
  }

  if (SUCCEEDED(result)) {
    /* CPU can be in WAIT state, so it needs to be enabled */
    result = EnableCPU();
  }

  LOG_EXIT(result);
  return result;
}
```

8.10.3 Acquire

```
/*****************************************************************************
 * Performs a variety of chip acquisition attempts:
 * 1. Check if the device is already in IDLE or DEAD branches
 * 2. Try to acquire in Test mode (TM). This is recommended and the only 100% reliable method. But
 *    it will not work if debugger cannot meet timing requirements or Listen window is disabled.
 * 3. Try to acquire using the Vector Catch
 * Acquisition methods may be invoked twice - with hardware (XRES) and software pre-reset.
 *
 * ! Note that XRES connection is strongly required for the hardware reset. Otherwise, neither of
 * the above methods will work if the firmware does "bad" things such as:
 *   - Repurposes the debug pins (intentionally or unintentionally)
 *   - Disables/Protects access ports and the Listen window is turned off or too short
 *   - Intentionally or unintentionally corrupts values in MMIO registers and the Listen window is
 * turned off or too short In this case, there is no way for the debugger to establish even basic
 * communication with the target
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
int Acquire(uint8_t *apNum) {
  LOG_ENTRY();

  int result;
  uint32_t v;
  uint8_t v1;
  uint8_t acqMethods;
  result     = RESULT_ERR;
  acqMethods = ACQUIRE_METHODS_ALLOWED;

  /* SysAP should always use Secure Access  */
  if (*apNum == 0)
    _DOMAIN_SECURE = 1;

  /* --- (0) Attach --- */
  if (acqMethods == 0) {
    /* Just init DAP if all acquisition methods are disabled */
    result = DAP_HandshakeAndInit(*apNum, TIMEOUT_HANDSHAKE);
  }

  /* --- (1) Check IDLE --- */
  /* Check whether the device is already in IDLE or DEAD branch, what is sufficient condition for programming,
   * so Reset/Acquire is not needed. */
  if ((acqMethods & ACQUIRE_CHECK_IDLE) != 0) {
    result = DAP_HandshakeAndInit(*apNum, TIMEOUT_HANDSHAKE_SMALL);
    if (SUCCEEDED(result)) {
      /* L1BOOT_ID_FAIL in SRAM_STATUS_ADDR indicates that boot code reached the
       * CORRUPTED branch - major system failure occurred (e.g. BIST failed). Debugger has limited
       * MCU access (via System Access Port only), programming/debugging is not possible */
      result = ReadMem(*apNum, SecureAddr(SRAM_STATUS_ADDR), &v);
      if (!SUCCEEDED(result) || (v & L1BOOT_ID_MSK) == L1BOOT_ID_FAIL) {
        result = RESULT_ERR_CRITICAL;
      }
    }
    if (SUCCEEDED(result)) {
      result = WaitForBootIdle(*apNum, TIMEOUT_LISTEN_WND);
```

```
    }
  }

  /* 2. Try to acquire in Test mode (TM) */
  if (FAILED(result) && (result != RESULT_ERR_CRITICAL) && ((acqMethods & ACQUIRE_TEST_MODE) != 0)) {
    result = AcquireTestMode(RST_TYPE_HARD & RST_TYPES_ALLOWED, *apNum);
    if (FAILED(result) && (result != RESULT_ERR_CRITICAL)) {
      /* If the acquisition failed for some reason (e.g. XRES is not connected), try to acquire in
       * Test mode using software reset This should work if there is no valid user application so
       * the ROM boot code is in WFA state or if the Listen window is wide enough and running
       * application did not disabled or corrupted the debug infrastructure */
      result = AcquireTestMode(RST_TYPE_SOFT & RST_TYPES_ALLOWED, *apNum);
    }
  }

  /* Try to acquire using the Vector Catch or the alternate sequence (breakpoint in RAM) */
  if (FAILED(result) && (result != RESULT_ERR_CRITICAL) &&
      ((acqMethods & (ACQUIRE_BREAKPOINT | ACQUIRE_VECTOR_CATCH)) != 0)) {
    result = RESULT_OK;

    /* If SYS-APP is not strictly preferred, try to find first available AP with CPU access */
    if (*apNum == AP_SYS) {
      if (AP_TO_USE_STRICT != 0) {
        result = RESULT_ERR;
      } else {
        result = DAP_ScanAP(&v1);
        if (SUCCEEDED(result)) {
          *apNum = v1;
        }
      }
    }

    if (SUCCEEDED(result)) {
      result = RESULT_ERR;

      /* 3. Try to acquire using the Vector Catch */
      if ((acqMethods & ACQUIRE_VECTOR_CATCH) != 0) {
        result = AcquireVectorCatch(RST_TYPE_HARD & RST_TYPES_ALLOWED, *apNum);
        if (FAILED(result)) {
          result = AcquireVectorCatch(RST_TYPE_SOFT & RST_TYPES_ALLOWED, *apNum);
        }
      }
    }
  }

  LOG_EXIT(result);
  return result;
}
```

8.11 Unlocking access to the CPU

8.11.1 WaitForWFAMode

```
/*****************************************************************************
 * Polls the target (with given timeout) waiting for WFA mode to be entered
 *
 * Return value
 *   >= 0  O.K.
 *   <  0  Error
 */
static int WaitForWFAMode(uint8_t apNum, int timeout) {
  int result = RESULT_ERR;
  int t;
  int tDelta;
  uint32_t v;

  t = SysGetTimeMs();
  do {
    /* Sleep between polling - let the target do its job and avoid too much garbage on SWD */
    SysSleepMs(10);

    /* Read SRSS->BOOT_DLM_CTL register and check the WFA bit is set, indicating a WFA mode */
    result = ReadMem(apNum, SRSS_BOOT_DLM_CTL, &v);
```

```
    if (SUCCEEDED(result)) {
      if ((v & SRSS_BOOT_DLM_CTL_DEBUG_WFA) != 0) {
        /* WFA branch reached */
        result = RESULT_OK;
        break;
      }
    }

    tDelta = SysGetTimeMs() - t;
  } while (tDelta < timeout);

  return result;
}
```

8.11.2 AcquireInWFAMode

```
/*******************************************************************************
 * Acquires the target in WFA mode using specified request code
 *
 * Return value
 *   >= 0  O.K.
 *   <  0  Error
 */
static int AcquireInWFAMode(uint8_t apNum, uint32_t req) {
  int result;

  result = WriteMem(apNum, SRSS_BOOT_DLM_CTL, req);
  if (SUCCEEDED(result)) {
    /* This write triggers soft-reset causing transaction failure, ignore the error */
    WriteMem(apNum, SRSS_RES_SOFT_CTL, SRSS_RES_SOFT_CTL_TRIG_SOFT);

    result = DAP_HandshakeAndInit(0, TIMEOUT_HANDSHAKE);
    if (SUCCEEDED(result)) {
      result = WaitForWFAMode(apNum, TIMEOUT_HANDSHAKE);
    }
  }
  return result;
}
```

8.11.3 LoadDebugCert

```
/*******************************************************************************
 * Reads the debug certificate and loads it to the RAM
 *
 * Return value
 *   >= 0  O.K.
 *   <  0  Error
 */
static int LoadDebugCert(uint8_t apNum, const char *cert_path) {
  int result       = RESULT_OK;
  size_t num_reads = 0;
  uint32_t cert_buffer[DEBUG_CERT_SIZE / 4];

  if (cert_path == NULL)
    cert_path = "debug_cert/debug_cert_oem.bin";

  struct stat s;
  result = stat(cert_path, &s);
  if (result) {
    log_write("Failed to read the Debug Certificate '%s'", cert_path);
    result = RESULT_ERR_CRITICAL;
  }

  if (SUCCEEDED(result) && s.st_size != DEBUG_CERT_SIZE) {
    log_write("Debug Certificate size mismatch");
    result = RESULT_ERR_CRITICAL;
  }

  if (SUCCEEDED(result)) {
    FILE *f = fopen(cert_path, "rb");
    if (f) {
      num_reads = fread(cert_buffer, 1, sizeof(cert_buffer), f);
```

```
      if (num_reads != sizeof(cert_buffer))
        result = RESULT_ERR_CRITICAL;

      fclose(f);

      if (SUCCEEDED(result)) {
        for (uint32_t i = 0; i < DEBUG_CERT_SIZE / 4; i++) {
          result = WriteMem(apNum, DEBUG_CERT_ADDR + (i * 4), cert_buffer[i]);
          if (FAILED(result)) {
            break;
          }
        }
      }
    }
  }

  return result;
}
```

8.11.4 StartWFARequest

```
/*****************************************************************************
 * Executes WFA request. Target must be acquired in FWA mode before
 * calling this function.
 *
 * Return value
 *   >= 0  O.K.
 *   <  0  Error
 */
static int StartWFARequest(uint8_t *apNum) {
  int result;
  int t;
  int tDelta;

  /* This write triggers soft-reset causing transaction failure, ignore the error */
  WriteMem(*apNum, SRSS_BOOT_DLM_CTL, 0);

  t = SysGetTimeMs();
  do {
    /* Sleep between polling - let the target do its job and avoid too much garbage on SWD */
    SysSleepMs(10);

    /* Scan the Access Ports for the first available with CPU registers access */
    result = DAP_ScanAP(apNum);
    if (SUCCEEDED(result))
      break;

    /* CPU AP is closed, continue polling */
    tDelta = SysGetTimeMs() - t;
  } while (tDelta < TIMEOUT_HANDSHAKE);

  return result;
}
```

8.11.5 UnlockCPUAccess

```
/*****************************************************************************
 * Unlocks the access to the CPU using given debug certificate.
 * CPU is left running after calling this function. The following
 * steps are performed:
 * 1. Target is acquired in WFA mode with request #2
 * 2. Debug certificate is loaded into RAM
 * 3. WFA request #2 is executed, this will enable CM33 AP
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
int UnlockCPUAccess(uint8_t apNum, const char *cert_path) {
  int result = DAP_HandshakeAndInit(apNum, TIMEOUT_HANDSHAKE);
  if (SUCCEEDED(result))
    result = AcquireInWFAMode(apNum, WFA_REQUEST_DEBUG_CERT);
```

```
  if (SUCCEEDED(result))
    result = LoadDebugCert(apNum, cert_path);

  if (SUCCEEDED(result))
    result = StartWFARequest(&apNum);

  return result;
}
```

8.11.6 UnlockCPUAccessAndHalt

```
/*****************************************************************************
 * Resets the CPU and halts it at the first instruction using given
 * Debug Certificate. This function is require only to launch the
 * debug session from the beginning of code execution.
 *
 *  Return value
 *    >= 0  O.K.
 *    <  0  Error
 */
int UnlockCPUAccessAndHalt(uint8_t apNum, const char *cert_path) {
  int result;
  uint32_t v1;
  uint32_t v;

  result = UnlockCPUAccess(apNum, cert_path);

  if (SUCCEEDED(result))
    result = HaltCPU();

  if (SUCCEEDED(result))
    result = WriteMem(apNum, DEMCR_ADDR, DEMCR_TRCENA | DEMCR_VC_CORERESET);

  if (SUCCEEDED(result))
    result = UnlockCPUAccess(apNum, cert_path);

  if (SUCCEEDED(result))
    result = DAP_Init(apNum);

  if (SUCCEEDED(result)) {
    v      = DHCSR_S_RESET_ST | DHCSR_S_HALT | DHCSR_C_DEBUGEN;
    result = PollMem(DHCSR_ADDR /* 0xE000EDF0 */, v, 0, v, TIMEOUT_HANDSHAKE, 10, &v1);
  }

  return result;
}
```

Revision history

Date	Version	Description
2025-09-12	*F	Initial public release

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

Edition 2025-09-12
Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to **www.cypress.com/support**

Document reference
002-37778 Rev. *F

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

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

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

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

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

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

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