PMBus®
Power System Management Protocol Specification

Secure Device Application Profile

Revision 1.1
18 Sep 2025

www.powerSIG.org

Filename: PMBus_Specification_Secure_Device_App_Profile_1_1_20250918.docx
Last saved: 18 Sep 2025, 12:10

DISCLAIMER

This specification is provided “as is” with no warranties whatsoever, whether express, implied, or statutory, including but not limited to any warranty of merchantability, non-infringement, or fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification or sample.

In no event will any specification co-owner be liable to any other party for any loss of profits, loss of use, incidental, consequential, indirect, or special damages arising out of this specification, whether or not such party had advance notice of the possibility of such damages. Further, no warranty or representation is made or implied relative to freedom from infringement of any third party patents when practicing the specification.

Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owner’s benefit, without intent to infringe.

REVISION HISTORY

REV	DATE	DESCRIPTION	AUTHOR AND EDITOR
1.0	25 Aug 2025	First public release	Author: Robert W. Santucci, Intel Editor: Robert V. White Embedded Power Labs
1.1	18 Sep 2025	Public release, Revision 1.1	Author: Robert W. Santucci, Intel Editor: Robert V. White Embedded Power Labs

Table of Contents

Table of Figures

Table of Tables

1. Introduction

As new threats on the supply chain and at runtime emerge, PMBus® is integral to system security. Running tampered firmware and/or configurations on voltage regulators may expose the system to various attacks. As a cyber-physical system, where physical voltages are produced, voltage regulators present unique possibilities to an attacker. By varying voltage supplied to the processor, a malicious voltage regulator (VR) firmware (FW) could jolt the processor out of its normal routine, skipping security processes such as password-checking. Malicious voltages may also make random numbers used to encrypt data predictable, invalidating encryption. Incorrect VR configuration or execution of PMBus commands from aggressors may also cause physical damage by applying unexpected voltages, resulting in “denial of service” attacks.

To mitigate against these attacks, this profile recommends employing a 3-part solution:

- **Attestation**: PMBus target device, firmware, and configuration shall be attested using a pre-shared key to the PRoT at every boot cycle. The PRoT can also initiate this attestation request during runtime. The attestation process defined in this profile ensures the firmware and configuration bits of the PMBus target contain the expected non-volatile memory values for this application.
- **NVM Authentication:** This profile provides a mechanism for secure device firmware updates. This mechanism requires firmware to be digitally signed or measured by the system integrator and/or PMBus device manufacturer and verified prior to storing during boot/update cycles by the PRoT. If this firmware is securely transferred to the VR at every boot, this may enable inexpensive low-NVM devices. If applied upon request, it can enable PRoT driven updates of PMBus target device firmware or configuration.
- **Access Control:** This profile defines a method to send a PMBus command with structures to help verify both the integrity of the requestor and the contents of the packet. This helps prevent rejections from an unauthorized host on the bus.

The platform root-of-trust (PRoT), operating as PMBus host (controller), drives security functions. This application profile defines 4 different security profile levels targeting different security vs complexity tradeoff points. The PRoT approves system power-on once all VRs are attested. The PRoT also handles update requests from the outside including an initial public-key signature verification at the PRoT before attempting to update any PMBus target VRs. This system is illustrated in Figure 1-1.

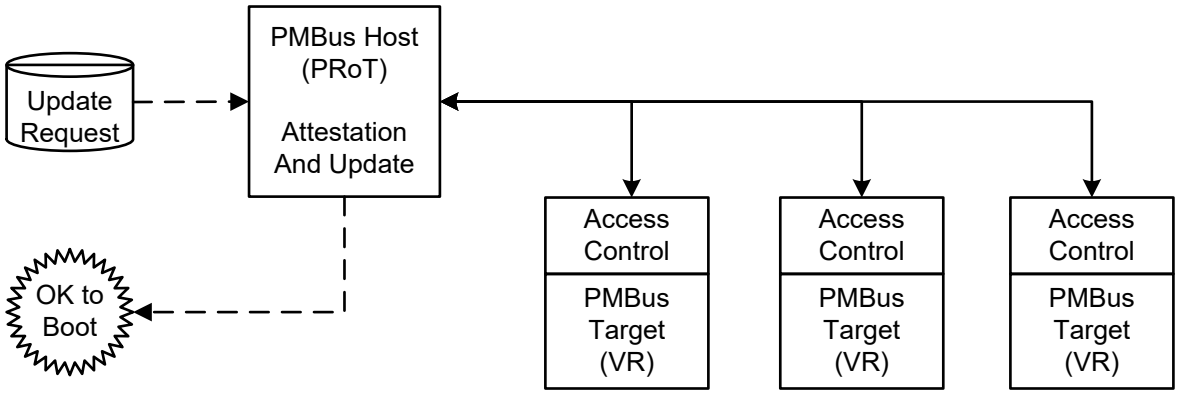

Figure 1-1. Platform Root of Trust and PMBus Targets

1.1 Specification Scope

1.1.1 Specification Structure

The PMBus specification is in four parts plus this secure device application profile.

Part I includes the general requirements, defines the transport, electrical interface and timing requirements of hardwired signals for PMBus.

Part II defines the command language used with PMBus. For the context of this application profile, Part II contains the ACCESS_CONTROL command used to configure PMBus command accessibility. It also contains the PMBus commands for communicating with PMBus standard hardware security device.

Part III defines the transport, electrical interface, timing requirements and command language for PMBus 1.x

Part IV describes the standard hardware implementation and hashing standards used to secure PMBus devices. This includes the security memory structure, action requests, and related hardware processes enabling security through PMBus and this PMBus Secure Device Application Profile.

Finally, this security application profile motivates the need for security, defines different security levels and the requirements of each, and a software application programmable interface (API) to provide hardware abstraction during implementation.

Implementation of this application profile using the standard hardware defined in the Part IV specification is strongly recommended, but not mandatory to meet the requirements of this application profile. It is the responsibility of any vendor to ensure that any proprietary commands or direct memory accesses they define do not violate data control and security achieved in Part IV's reference interface and implementation. They must not leave memory vulnerable to being overwritten or reconfigured during a security action request. Data protection must be implemented in target hardware. it is not sufficient to use software to emulate security functions while not supporting it in the underlying hardware as rogue PMBus hosts may be on the bus.

1.1.2 What Is Included

This application profile defines profiles meeting several points upon the security strength versus cost curve to target various markets. It standardizes software application programming interface (API) calls to provide a level of hardware abstraction, to ensure greater commonality between PMBus device vendors for core security functions. Security level 0 is the least expensive to implement, security level 3 provides the highest strength for more capable devices.

1.1.2.1 Security Level 0

Security level 0 targets inexpensive devices where security is a secondary concern, adequately covered by securing the PMBus host. It provides requirements that make it possible to lock down access to critical PMBus commands. Security level 0 helps ensure the target is not used to launch a cyber-physical attack by locking access to key commands that can directly impact output voltage and/or modify self-preservation protection settings such as over/under current/voltage/temperature.

Security level 0 provides guidance on firmware updates, primarily on being able to lockout access to the NVM store or firmware update commands. Firmware updates are required to be protected by some passkey of given length and limited retries and

to have some basic integrity check. This integrity check can be a simplistic checksum designed to catch bit errors rather than intentionally inserted malicious code. Even at this level, firmware should contain some versioning mechanism to ensure old updates are not reapplied. This checksum does not provide protection against the update being replayed on other VRs.

This application profile level does not provide protection against supply chain attacks such as legitimate devices assembled with compromised firmware or counterfeit devices with additional capabilities.

1.1.2.2 Security Level 1

Security level 1 profile enhances the security of level 0 by providing the ability to securely validate a measurement of target firmware and configuration prior to enabling the VR output. This action is called attestation.

While attestation does not provide added strength preventing an attacker from applying a compromised firmware or configuration, it can detect its presence afterwards. This method can catch a supply chain attack by validating the expected firmware and configuration measurement at power-on matches the platform designer's expected value.

The attestation process requires that the PMBus host (PRoT) generates strong random numbers and that both the target and the host (PRoT) maintain the secrecy of a pre-shared key. The combination of the transmitted random number and the never-transmitted pre-shared-key (PSK) known to both the host (PRoT) and target provide protection against replay attacks where the target would reply with the firmware and configuration measurement value recorded an uncompromised system.

The protection for both access control and updates to firmware and configuration retain the properties of Security Level 0.

1.1.2.3 Security Level 2

Security Level 2 expands upon the capabilities in Security Level 1 by providing more rigorous and replay-attack protected integrity checks for firmware updates. In this methodology, an attestation calculation is applied to the candidate image by target and compared against an expected attestation result provided with the image. Only if the expected attestation matches the computed attestation result from the candidate image will the target accept the incoming image.

Knowledge of the secret PSK is needed by an agent computing the attestation result, or "message authentication code" (MAC). This is true both for the target device accepting the candidate image and the host (PRoT) sending both the candidate image and the expected MAC result to the target.

Management of the PSK is essential for this scheme.

1.1.2.4 Security Level 3

Security Level 3 extends the functionality of security level 2, but uses a public key cryptography, also known as asymmetric cryptography, to validate incoming firmware images on the PMBus target (VR). Using this method, the incoming firmware image is signed by the author using a private key. The candidate image verification is done using the public key stored in the VR. This security technique eases assurance of VR firmware authenticity by allowing for tightly controlling the private signing key, distributing only the public key. The cryptography algorithm recommended for public

key cryptography is NIST FIPS 204, “Module-Lattice-Based Digital Signature Standard”.

Beyond the public key cryptography-based firmware/configuration update method, this level also defines a method to attest the identity of the host sending PMBus commands. This method is very similar to the method used to attest the target starting in security level 1, but in this scenario:

- A sufficiently random nonce must be produced by the target device for the host.
- The host computes an expected MAC based on the command contents to be sent and the nonce
- The target computes the MAC from the nonce and incoming command.
- Target validates the MAC it calculated matches the expected MAC sent by the host. If they match the command is executed.

This technique enables defining an access-control permission where commands are allowed to run if the host can be attested. It enables additional options where the host can dynamically configure options such as if a target device alerts upon security action request completion or must be polled or issuing a reset command while the output is on.

1.1.2.5 Summary Level Summary

Table 1-1 summarizes the difference in features between the security levels. Note that the security level drives the methods and tools available for security. They do not specify the strength of the hashing or signing algorithms. Thus, designers will need to check that the algorithms utilized by the target device are sufficiently strong for their target application and timeframe.

Table 1-1. Security Profile Levels

Feature	**Security Profile Level 0**	**Security Profile Level 1**	**Security Profile Level 2**	**Security Profile Level 3**
Access control	Required	Required	Required	Required
VR attestation	Optional	Required	Required	Required
Target generated nonces, support for commands with attestation of host	Optional	Optional	Optional	Required
Firmware Updates – rows below pertain when updates are allowed				
Firmware & Configuration update: Signature verified by PRoT prior to communication with PMBus target (does not affect target hardware)	Recommended	Required	Required	Recommended (Signature is also verified by target)

Feature	Security Profile Level 0	Security Profile Level 1	Security Profile Level 2	Security Profile Level 3
Password-based write protection if firmware / config updates allowed	Required	Required	Prohibited	Prohibited
VR checksum verification of PRoT authenticated firmware and configuration by target device if updates allowed	Required	Required	Prohibited	Prohibited
Authentication via attestation-like MAC structure by target device if updates al	Optional	Optional	Required	Prohibited (Signature Verification)
Signature verification by target device	Optional	Optional	Optional	Required

Table 1-2. PMBus Secure Action Requests by Level

Action Request Code From [A03]	Action Request Name	Level 0 Profile	Level 1 Profile	Level 2 Profile	Level 3 Profile	Section
00h	Device attestation		R	R	R	8.3.1
01h	Sec Level 1 authenticated update			X	X	8.6.2
02h	Sec level 2 authenticated update		X	R	X	8.6.3
03h	Sec level 3 authenticated update		X	X	R	8.6.4
04h	PSK iterate request[1] Nonce can be PRoT or target generated.			R	R	8.2.3

Action Request Code From [A03]	Action Request Name	Level 0 Profile	Level 1 Profile	Level 2 Profile	Level 3 Profile	Section
05h	PSK_0 (Initial PSK) one-time write		R	R	R	8.2.1
06h	PSK lock with target-generated nonce[2,3]					8.2.4
07h	Power-On Reset while output off		R	R	R	8.6.5
08h	Firmware Fetch [2]					8.6.6
09h	Firmware Fetch Lockout[2,3]					8.5.7
0Ah	Power-On Reset Lockout[2,3]				R	8.5.5
0Bh	PMBus Status alert triggering[2,3]				R	8.5.4
0Ch	Power-On Reset (Anytime) [2,3]				R	8.5.6
0Dh	Page Plus R/W w/ host attested[2,3]				R	8.5.3
0Eh	Secured Access Control Action[2,3]				R	8.5.2
0Fh	Fetch Nonce				R	8.5.1
10h	PSK lock forever (NVM) with PRoT generated nonce[4]				R	8.2.4

R – Required Action Request for the listed Profile Level Compliance

X – Prohibited Action Request for the listed Profile Level Compliance (May be supported but must be disabled via NVM for compliance)

Note 1: These command scan use either a PRoT or target generated nonce

Note 2: These commands are strictly optional and driven by the needs of a particular product

Note 3: These commands require attestation of the controller identity. They must be executed following a “Fetch Nonce” command 0Fh. After their execution, the target must invalidate its generated nonce after each attempt of these commands. This is done using the steps of section 9

Note 4: Optional if coverage can be provided in NVM / firmware authenticated level 2 or 3 update, if firmware always locks out PSK iterate once shipped, or if PSK cannot be iterated. Otherwise, this feature is required.

1.1.2.6 Application Programming Interface

This document presents a list of standardized functions used to implement the required functionality. Device vendors are required to provide these APIs along with their hardware implementation. These functions, validated by the supplier and implemented to a common set of parameters, help software implementors port code from one supplier to another more rapidly. They allow hardware implementors some variances which can be abstracted away under the software driver.

If implementation for the standard hardware function of part IV becomes widespread, then a future class-level driver supporting many target (VR) models may be feasible.

1.1.3 Threat Matrix

Table 1-3 outlines potential attacks and if they are mitigated by the security profiles outlined in this document.

Table 1-3. Threat Matrix for PMBus Secure Device

#	Attack	Mitigated by Secure Device Specification
1	**Supply chain attack** Authorized third-party vendor replaces a genuine PMBus target device (VR) with a counterfeit/rogue device	**Level 1 and Higher** Rogue device’s PSK will not match PRoT PSK. Attestation result equivalence comparison by PRoT will fail. PRoT shall halt boot process.
2	**Man-in-the-middle (MITM)** Attacker attempts to bypass attestation on a compromised device by: (1) eavesdropping on the attestation response from the good target and (2) replaying that response to future attestation requests.	**Level 1 and Higher** Integrity of attestation traffic is protected by ephemeral key (MAC key) and not replay-able.

#	Attack	Mitigated by Secure Device Specification
3	**Compromised Agent on Bus** Attacker attempts to send unauthorized PMBus commands to target device from Controller capable device on Bus.	**Level 0, Level 1 & Level 2** ACCESS_CONTROL Never Again / Write Once bits limit write access to commands **Level 3** Authenticated PMBus Write w/ Target Generated Nonce allows update to commands protected by Never Again / Write Once.
4	**Firmware Update attack** Unauthorized update of the device FW with rogue FW by a malicious user	**Level 0 & Level 1 – PRoT Protected** Integrity of FW and configuration is protected by digital signature verified by the PRoT before request gets relayed to the PMBus device. **Level 2 – Device Protected with Attestation-like Authentication** **Level 3 – Device Protected with Signature Verification**
5	**Confused Deputy FW attack** Unauthorized FW update to PMBus target device by any compromised PMBus device on the same bus.	**Level 0 & Level 1, minimal:** Access Control limits write capability, Simple password & checksum only for writable commands. **Level 2, partial:** Protected up to the level where PSK may be compromised (PRoT must still be trusted to protect the PSK). **Level 3, full if manufacturer's private signing key remains secure (see attack #7):** PMBus target device itself attests the firmware. Secure so long as the signing private key is secured.
6	**Replay Attack (PRoT DRBG)** At end-user site, compromise the digital random bit generator (DRBG) used by PRoT such that the same "nonce" (number once) is returned every time instead of a unique random number. Eavesdrop on traffic between PRoT and good VR to observe MAC-value returned from attestation process. Replace PMBus device with compromised device. Replay the attestation result of the good VR	**No (Attestation)** If PRoT "nonce" is constant, attestation messages are replay-able, allowing counterfeit / compromised target devices to bypass Attestation. **Level 2** Only Same FW Update can be replayed with the same Target Address and Nonce. Revision tag increment requirement prevents reversion to older FW configuration. **Level 3** Only FW Updates with valid signatures can be replayed. Revision tag increment requirement prevents reversion to older FW configuration.

#	Attack	Mitigated by Secure Device Specification
7	**Key Compromise (PSK) attack** At end-user site, attacker compromises PSK on either PRoT or PMBus target, then replaces VR FW with rogue FW.	**Level 1 & 2 No.** If PSK is leaked, an attacker can generate valid attestation messages and in level 2 FW updates. **Level 3** Attestation not protected, but FW Updates of valid devices protected by Update Signature Requirement. ACCESS_CONTROL Never Again bit(s) may still limit Write Capability to critical commands.
8	**Key Compromise (PSK) attack** Manufacturer or system integrator's private key for signing VR FW is compromised. Attacker signs rogue VR FW and loads it on to the VR.	**No.** The rogue FW can be loaded/updated on the VR as a genuine FW
8	**Firmware downgrade attack** Attacker requests to replace PMBus target firmware/configuration with an older validly signed firmware/configuration image. This older image may have contained an exploitable vulnerability.	**Level 1 and Higher** This standard embeds a version tag into the firmware that must be greater than or equal to present value at each firmware update for the update to be permitted.
9	**Open-drain bus pull-down attack** Attacker attempts to alter packets driven by the PMBus controller by selectively pulling down some bits from 0 to 1 during the transaction.	**Partial, hardware dependent.** Detectable by a Bus controller that monitors the state of the bus as it is driving. Will present itself similarly to I2C bus arbitration issue. Bus controller may abort the packet and may halt any aggressors attempt to continue the aborted transaction by holding the bus low through the reset time. An alternate mitigation would be to drive a bad PEC intentionally with a 1->0 substitution made by the controller in the PEC once alteration was detected.

1.1.4 Where Is It Applied

The requirements are placed upon the set of the PMBus target devices (typically voltage regulators) and the host devices that drive them.

When a PMBus device (such as a multi-loop VR controller or PMIC) implements multiple outputs or sensors within a single device, the authentication and attestation commands shall be applied similarly to the method by which firmware updates are applied. If all firmware and configuration updates are programmed via a single page in a multi-output PMBus target, then the attestation and authentication commands for all outputs can be implemented via the same shared page. It can be implemented on

only a single page. If the firmware and configurations are completely independent on each target (VR or sensor page) within the PMBus target, then the attestation and authentication must be replicated for each independent function.

1.1.5 What Is Not Included In The PMBus Specification

The PMBus specification is not a definition or specification of:

- A particular power conversion device or family of power conversion devices.
- A specification of any individual or family of integrated circuits.

This specification does not address direct unit to unit communication such as analog current sharing, real-time analog or digital voltage tracking, and switching frequency clock signals.

1.2 Specification Changes Since The Last Revision

A summary of the changes between this revision and the previous revision are shown in Table 12-1 at the end of this document.

1.3 Where To Send Feedback And Comments

Please send all comments by email to: securityfb@smiforum.org

2. Related Documents

2.1 Scope

If the requirements of this specification and any of the reference documents are in conflict, this specification shall have precedence unless otherwise stated.

Referenced documents apply only to the extent that they are referenced.

The latest version and all amendments of the referenced documents at the time the device is released to manufacturing apply.

2.2 Applicable Documents

Applicable documents include information that is, by extension, part of this specification.

[A01] System Management Bus (SMBus) Specification
http://smbus.org/specs/SMBus_3_1_20180319.pdf

[A02] PMBus™ Power System Management Protocol, Part II, Command Language
https://pmbus.org/current-specifications/

[A03] PMBus™ Power System Management Protocol, Part IV, Security
https://pmbus.org/current-specifications/

[A04] NIST FIPS 180-4: Secure Hash Standard
http://dx.doi.org/10.6028/NIST.FIPS.180-4

[A05] NIST FIPS PUB 198-1: The Keyed-Hash Message Authentication Code (HMAC)
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.198-1.pdf

[A06] NIST FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions
http://dx.doi.org/10.6028/NIST.FIPS.202

[A07] NIST FIPS 204: Module-Lattice-Based Digital Signature Standard
https://doi.org/10.6028/NIST.FIPS.204

[A08] NIST SP 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash.
https://doi.org/10.6028/NIST.SP.800-185

[A09] NIST SP 800-108 Rev 1: Recommendation for Key Derivation Using Pseudorandom Functions
https://doi.org/10.6028/NIST.SP.800-108r1

[A10] NIST SP 800-90A Rev 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators.
https://doi.org/10.6028/NIST.SP.800-90Ar1

[A11] NIST SP800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation.
https://doi.org/10.6028/NIST.SP.800-90B

[A12] NIST SP800-90C: Recommendation for Random Bit Generator (RBG) Constructions (3rd Draft)
https://doi.org/10.6028/NIST.SP.800-90C.3pd

[A13] PMBus Power System Management Protocol, Part I, General Requirements, Transport And Electrical Interface, Revision 1.5

[A14] PMBus Power System Management Protocol, Part II, Command Language, Revision 1.5

[A15] SMBus System Management Protocol, Revision 3.3.1

2.3 Reference Documents

Reference documents have background or supplementary information to this specification. They do not include requirements or specifications that are considered part of this document.

[R01] Cryptographic Algorithm Validation Program
https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/secure-hashing

[R02] Secure Hash Standard Validation System:
https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/shs/SHAVS.pdf

[R03] Secure Hash Algorithm-3 Validation System (SHA3VS):
https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/sha3/sha3vs.pdf

[R04] SP800-90B Entropy Assessment Software:
https://github.com/usnistgov/SP800-90B_EntropyAssessment

[R05] Cryptographic Module Validation Program (CMVP)
https://csrc.nist.gov/Projects/cryptographic-module-validation-program/entropy-validations/announcements-1

[R06] SP800-90B Entropy Source Validation Workshop
https://www.nist.gov/news-events/events/2021/04/sp-800-90b-entropy-source-validation-workshop

[R07] NIST SP 800-193. Platform Firmware Resiliency Guidelines
https://doi.org/10.6028/NIST.SP.800-193

3. Reference Information

3.1 Numerical Formats

All numbers are decimal unless explicitly designated otherwise.

3.1.1 Decimal Numbers

Numbers explicitly identified as decimal are identified with a suffix of “d”.

3.1.2 Floating Point Numbers

Numbers explicitly identified as floating point are identified with a suffix of “f”.

3.1.3 Binary Numbers

Numbers in binary format are indicated by a suffix of “b”.

Unless otherwise indicated, all binary numbers are unsigned. All signed binary numbers are two’s complement.

3.1.4 Hexadecimal Numbers

Numbers in hexadecimal format are indicated by a suffix of “h”.

3.1.5 Examples

255d ⇔ FFh ⇔ 11111111b

175d ⇔ AFh ⇔ 10101111b

1.2f

3.2 Abbreviations, Acronyms And Definitions

Table 3-1. Abbreviations, Acronyms and Definitions Used In This Specification

Term	Definition
ACK	Acknowledge
API	Application Programming Interface
HMAC	Hashed Message Authentication Code
HMAC	Hashed Message Authentication Code
KDF	Key Derivation Function
KDF	Key Derivation Function
KMAC	Keccak Message Authentication Code
MAC	Message Authentication Code
MAC	Message Authentication Code

Term	Definition
NACK	Not Acknowledge
NVM	Non-volatile Memory
OTP	One-time programmable memory (fuses)
PEC	Packet Error Code
PRoT	Platform Root of Trust
PRoT	Platform Root of Trust
PSK	Pre-shared key

4. General Requirements

4.1 Compliance

The goal of this specification is to prevent PMBus targets from being exploited for the purpose of causing permanent or temporary denial of service. It defines a set of commands to authenticate that target devices populated on the PMBus are the exact devices with the exact firmware and configuration intended by the platform designer. Additionally, it defines a methodology to help ensure that PMBus targets only accept firmware updates specifically authorized by the platform designer. Finally, it presents methods that allow restricting the use of potentially exploitable PMBus commands.

Figures demonstrating specific physical implementation, such as a given physical memory size or arrangement, are made for illustrative purposes only. Internal implementations may vary provided external specification is met, though implementation per the standard implementation of PMBus Specification Part IV [A03] is recommended.

4.1.1 Commands

To be compliant to the PMBus secure device specification: A device must support all the specified Commands, Command Options, Security Actions, and Access Controls required for that level of the Application Profile. If a device accepts a PMBus secure device command, it must execute the associated function as described in this document and in alignment with PMBus Specification Part IV [A03] functionality.

If a device does not accept a given PMBus secure device command, it must respond per the “Invalid or Unsupported Command Fault” in STATUS_CML as per PMBus Specification Part II [A02].

A device may support non-required Commands or Security Action Requests, including MFR_SPECIFIC commands or requests, per the PMBus Specification Part II, Revision 1.5 Specification unless such commands or requests are prohibited by the supported Application Profile Level.

It is possible for a device to meet PMBus secure device application profile requirements but not utilize the reference-level register implementation defined in [A03] if and only if provided with the full set of APIs required by the application profile. This provides high-level software commonality where vendor-specific differences in this reference implementation may exist.

Security level 0 profile uses commands 0Eh and 0Fh, shown in Table 4-1. In security levels 2 and 3, these commands are expected to be configured via NVM, write-protected with no more access changes allowed. In security level 3, a PMBus command with host attestation is defined in [A03] section 11.1 is defined to allow ACCESS_CONTROL command to be executed only by an attested host.

To access the security commands defined in security levels 1, 2, and 3 the PMBus commands 70h-72h are utilized. These commands are detailed in PMBus part II [A02], with sections given in Table 4-1.

The PMBus secure device specification consumes five commands at the PMBus level.

Table 4-1. PMBus Secure Device Command Codes by Level

Command Code	Command Name	Level 0 Profile	Level 1 Profile	Level 2 Profile	Level 3 Profile	[A02] Section
0Eh	PASSKEY	R	R			19.3
0Fh	ACCESS_CONTROL	R				19.2
70h	SECURITY_BYTE		R	R	R	19.5
71h	SECURITY_BLOCK				8 Bytes	19.6
72h	SECURITY_AUTOINCREMENT				8 Bytes	19.7

R – Required Command Support for the listed Profile Level Compliance

Note: SECURITY_BLOCK and SECURITY_AUTOINCREMENT command support required with a minimum block length of the number bytes shown in the table.

4.1.2 Operation

PMBus secure devices must support all required aspects of the profile required for a given security level. For all security levels, the authenticated commands shall be the only method to store to or restore from NVM. When deployed, PMBus STORE and RESTORE commands must be permanently disabled for all PMBus secure devices. These commands bypass the NVM protection mechanisms.

At the discretion of the manufacturer and system integrator, devices may be delivered to a system integrator in an “unsecured” state. These devices require the capability of permanently disabling the STORE and RESTORE commands through NVM supported ACCESS_CONTROL functions or firmware updates and is not considered secure until STORE and RESTORE are disabled.

4.2 Testing

It is incumbent on PMBus target designers to ensure compliance to this profile. It is incumbent on the system integrators to verify compliance to a profile and the security of their platform root of trust (PRoT) software.

5. Security Actions

The security actions in this architecture are carried out originating from a platform root-of-trust (PRoT). In this specification, it is presumed that the PRoT is the device running software that calls the APIs defined in this specification to control the PMBus target. For

more information on PRoT architecture, refer to [R07]. The PRoT needs secure storage for storing PSKs in security level 1 and above. At those levels, it also must store the public key used to authenticate any incoming update request package before passing the request to the voltage regulator.

5.1 Attestation of Target

Attestation is a process that can be used to provide assurance that the PMBus target connected to the platform root of trust is the device intended by the system integrator and has correct, unmodified firmware and configuration. This feature is required in security level 1 and higher.

The attestation process relies heavily on hashing functions. A hashing function takes as input an incoming message consisting of many bits and produces a digest of a predetermined length, typically a shorter length than the message as shown in Figure 5-1. The hashing function must have certain properties, including the avalanche effect where a slight change in input produces a substantial change in the output digest. Further, the hashing function must be difficult to either invert or predict. For example, it must be exceedingly difficult to compute a message that produces a predetermined digest. Example hashing functions used in this specification include the Secure Hash Algorithm (SHA) 2 or 3 from NIST, defined in [A04] and [A08] respectively.

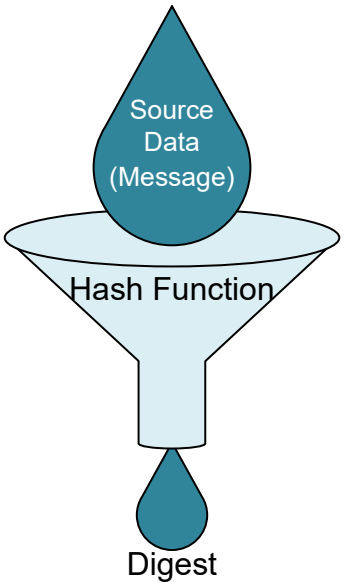

Figure 5-1. Hashing Function

For PMBus target attestation, the firmware and active configuration of the VR must be incorporated in the message. This message must also contain data indicating the device to which this request is being sent, as detailed in Figure 5-5. The hashing process computes a digest that represents the message. In this specification, a digest resulting from hashing a message including firmware, active configuration data, and device address is called a measurement of the firmware. This process is shown in Figure 5-2. Because there are fewer digest bits than (typical) message bits, many messages can produce the same measurement. Proper hashing function design makes it very difficult to compose a message that matches a predetermined digest. This is especially true when dealing with PMBus targets where specific firmware and configuration bits have a fixed function.

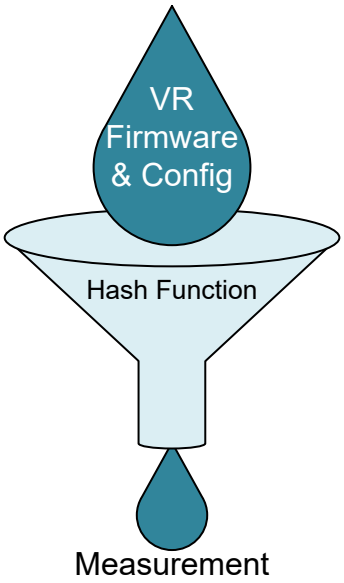

Figure 5-2. Firmware and Active Configuration Measurement

The attestation algorithm is designed ultimately to compare the measurement of firmware and active configuration on the PMBus target with the expected value stored in the PRoT secure memory. The PRoT cannot simply ask the VR for its measurement, as if it were transmitted in the clear any attacker may potentially snoop it for future replay attacks. To compare the firmware measurements, the results must be hashed with a time-varying key.

The time-varying, or ephemeral, key is formed from two components: (1) a random number called a nonce that is transmitted between the PRoT and the PMBus target, and (2) a pre-shared key (PSK) known to both the PRoT and the target but never transmitted over the bus. The function used to compute this key is known as a key derivation function (KDF). The KDF itself is a hashing function. This process is shown in Figure 5-3. The nonce contributes the time-varying quality to the ephemeral key as must be different for each attestation request. The PSK ensures that a snooper listening to the bus activity cannot deduce the ephemeral key from snooping the nonce, as the snooper also needs the non-transmitted PSK to derive the same ephemeral key. Ensuring that the PSK remains secret is key to maintaining security. The components and their role in the process are summarized in Table 5-1. Key derivation functions are described in [A09].

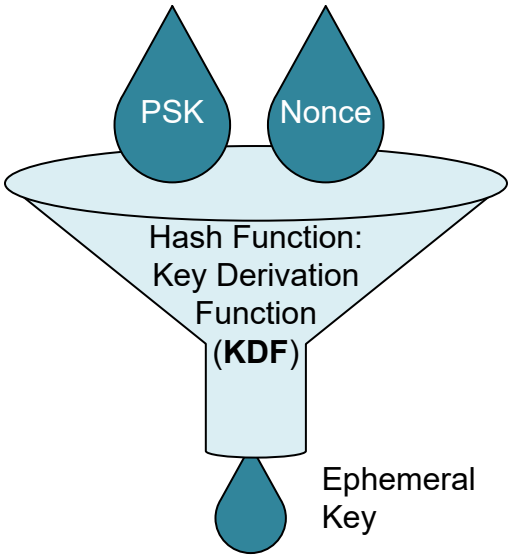

Figure 5-3. Ephemeral Key Calculation

Table 5-1. Ephemeral Key Components

Component A: Nonce	**Component B: Pre-Shared Key**
Unique for each attestation request	Unique per instance. Preloaded into both VR and the PRoT.
Non-Secret, transmitted from PRoT to VR	Secret. Never transmitted over the bus.
Ensures each attestation call has different ephemeral key.	Ensures ephemeral key cannot be computed from data transmitted in the clear.

Finally, the firmware and active configuration measurement hash is combined with the ephemeral key to produce a message authentication code. Once again, a hashing algorithm is used. In this case, the hashing algorithm is known as a “Message Authentication Code hashing function”. The result is coincidentally called a Message Authentication Code (MAC). The resulting MAC can be transmitted in the clear from the device attesting its identity to the trusted device. If they match, then the devices can be considered a likely match on the attested quantities: firmware, active configuration, and pre-shared key. Two message authentication code families, HMAC and KMAC, are described in [A05] and [A08] respectively. The overall MAC calculation is shown in Figure 5-4.

This MAC match can be considered a gateway in most systems to steps such as powering on a voltage regulator output. The exact mathematics and algorithms used to implement this process are explained in [A03]. An overall flowchart of the process, with software API calls described in this document is shown in Figure 5-5. Software APIs are described in section 8.3, with standard hardware implementation locatable in [A03] per Table 11-1.

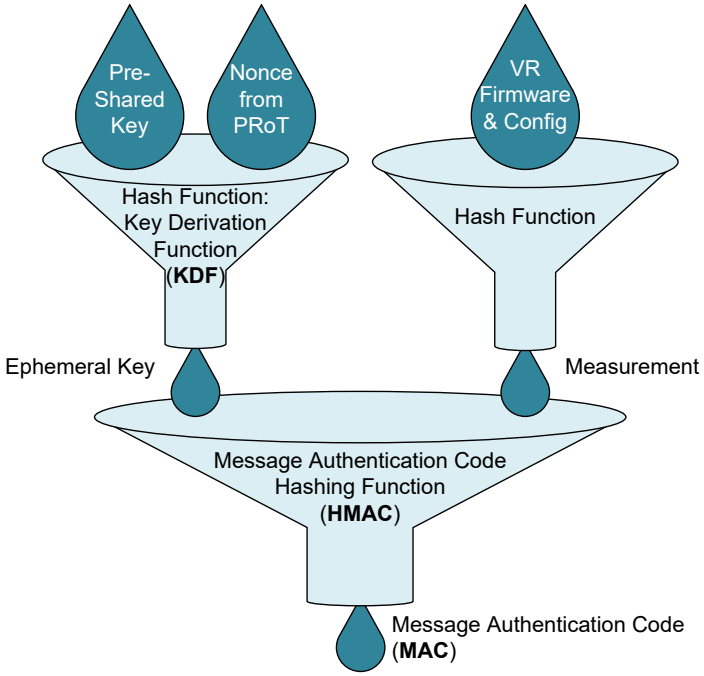

Figure 5-4. Message Authentication Code Calculation

This process requires the following hardware capabilities to support attestation:

- The PMBus target is required to support a hash function of its ROM and firmware.
- The PRoT is required to support a digital random bit generator (DRBG).
- Both PRoT and target must support a key derivation function (KDF) and a keyed hash function (HMAC/KMAC).
- A secure memory in both the PMBus target and the PRoT that cannot be accessed from off-chip or aggressor processes to store the secret PSK.
- If a device supports multiple configurations, the manufacturer must include the active configuration file and firmware in the hash calculation. It is optional to include inactive configuration files in the hash calculation. If the inactive configuration files are included as part of the hash calculation, the hash must incorporate data including which file is active, so the attestation can detect if the active configuration file choice has been changed. This attestation shall be computed over the firmware, NVM, and configuration data and need not be dynamically updated per any volatile commands applied by the PRoT after power-on. It must be recomputed after the push of an authenticated firmware or configuration update to either active volatile or non-volatile memory. The attestation shall not be affected by “transient” operations such as PMB VOUT_COMMAND that do not alter the firmware or default configuration settings.
- Handling must be taken to comprehend decommissioned firmware or configuration if it remains in regions included by the measured during attestation. It could mean multiple valid measurements may apply based on which firmware updates were applied between initial and most recent updates. It is recommended to exclude decommissioned firmware and configuration from measured data.

From the software perspective, this is the API process is shown in Figure 5-6.

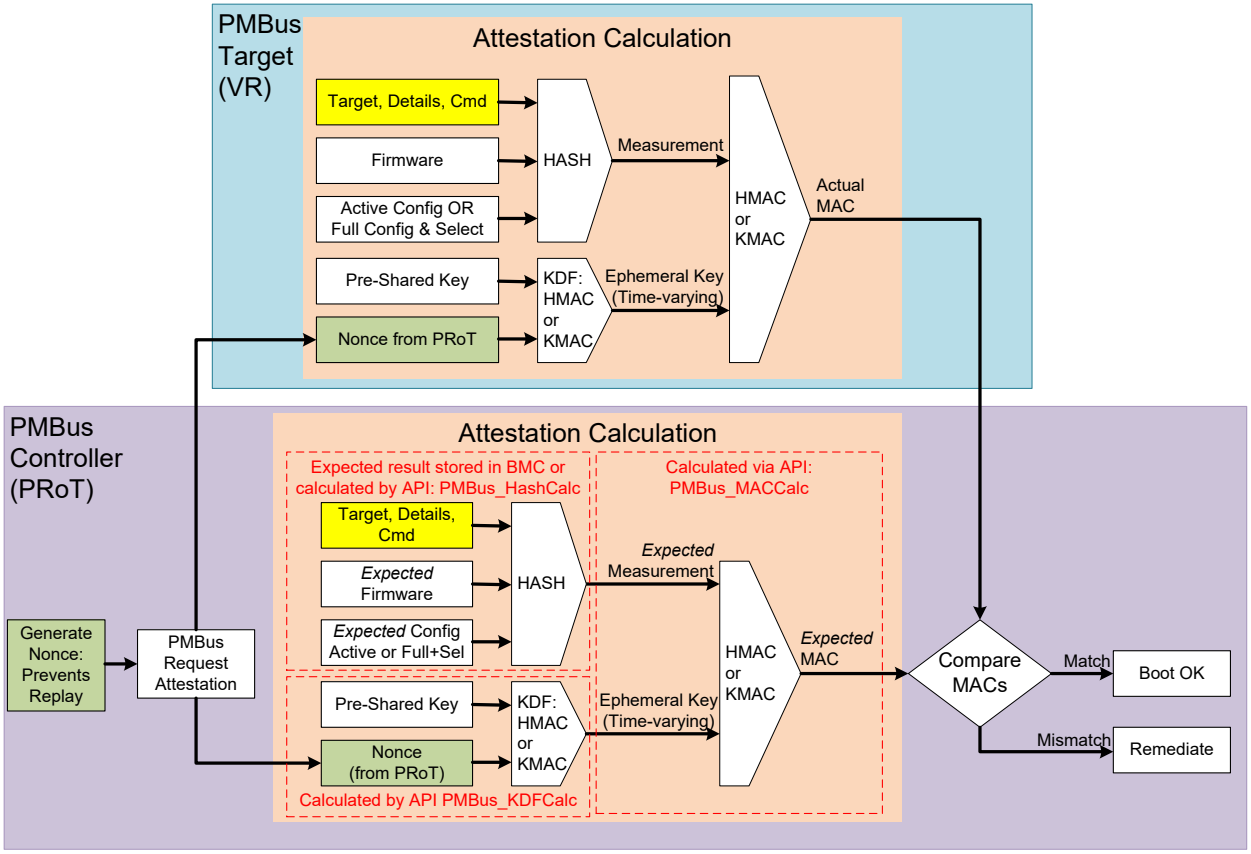

Figure 5-5. Process for Attestation of Target VR with Software Calls

5.2 Attestation of Host

The process described in this section attests that (1) the authorized controller (PRoT) is sending PMBus commands to the target device and (2) that the contents of those commands have not been altered. This procedure is similar to the attestation process described in section 5.1 used to attest the PMBus target. The process differs in that: (1) the data feeding the hash is based on the command being requested and its arguments including target address, (2) the target validates that the expected MAC delivered from PRoT matches the calculated in the target before executing the requested action, and (3) the nonce may be generated by the target.

The nonce is generated in the PRoT for commands where there is minimal chance of replay attack. For example, for PSK iteration the nonce can be PRoT generated as if the command runs successfully the PSK would be different. The differing PSK would make the old nonce and expected MAC pair invalid. Similarly, for PSK lockout non-volatile, the command could not be replayed as the PRoT is already locked.

For all other commands, replay-attacks are a concern, and DRBG in the PMBus target generates a nonce upon receiving a nonce request from the PRoT as described in section 8.5.1. Because generating a nonce with sufficient entropy is a difficult process, this capability is only required for security level 3 devices. Consequently, all remaining commands using attestation of host become required only at security level 3.

Requesting a nonce from the target dictates the attestation algorithm that will be utilized for the corresponding command. Typical targets only support 1 attestation algorithm at a

time. If following a reference implementation, the data to be measured is shown in [A03] Table 9-1.

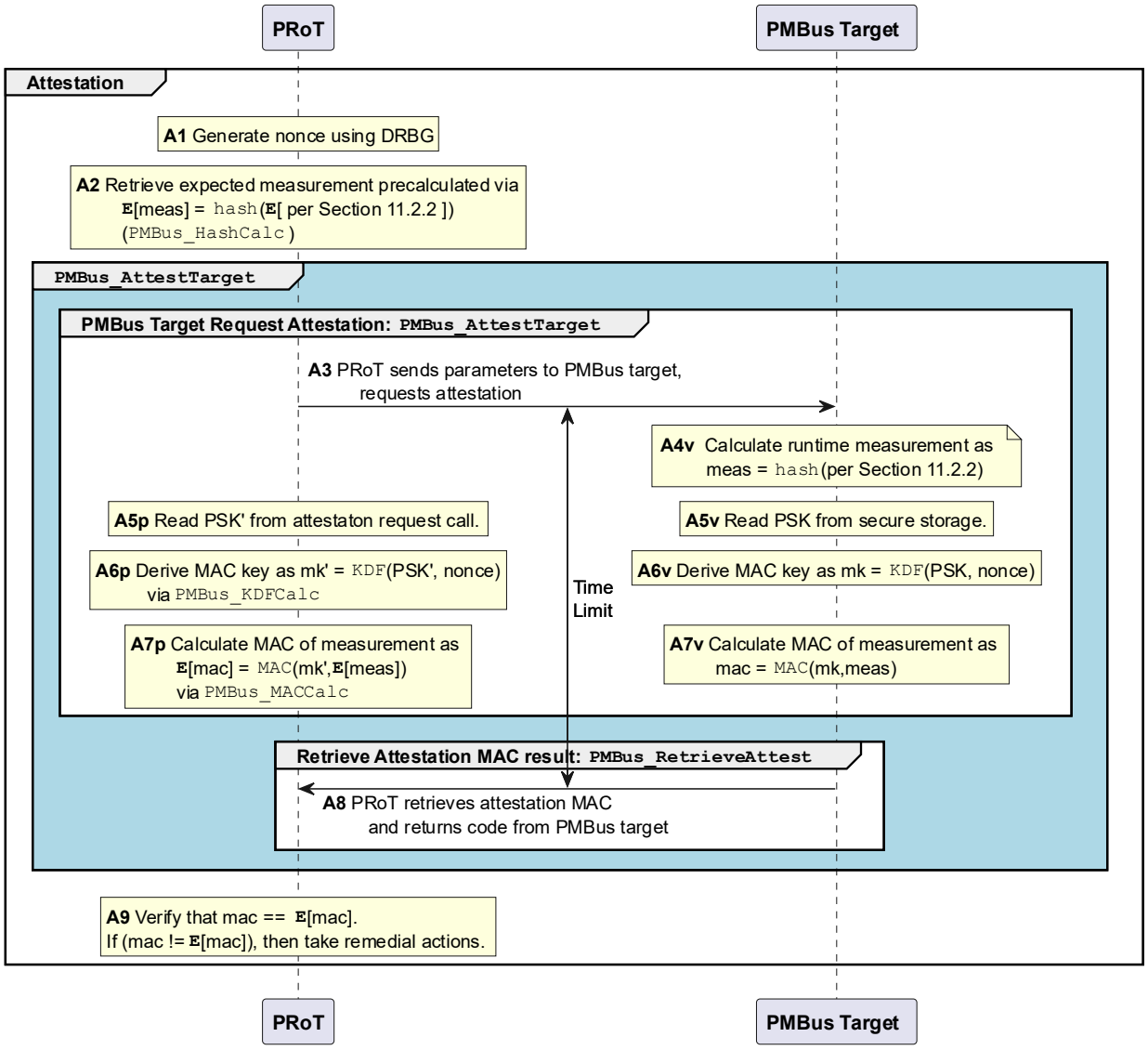

Figure 5-6. Attestation of Target Software Flow

Attesting the host allows opening writes to potentially harmful commands including VOUT_COMMAND, VOUT_MAX, protection threshold and responses, and access controls to be open to attested users, while denying access to another host that may be on the bus. Commands made possible with attestation of host include:

- PMBus Page Plus Write or Read from verified host
- Access Control Change Request
- PSK lock
- PSK iteration request
- PSK lock with target generated nonce
- Firmware fetch lockout

- Anytime power-on-reset request, allowing when output is on.
- Power-on-reset lockout to ensure no partial images on firmware update.
- Security action processor completion indication

Figure 5-7 presents the flow used for attestation of the host by the target. In this flow, the host first requests a nonce from the target, and computes a MAC encompassing that nonce, the mutually known PSK, and the command data. That expected MAC is sent along with the command request back to the target. The target will compute a MAC from its internal PSK, the nonce it generated, and the received command. The command will be executed if-and-only-if the expected MAC sent by the host matches the actual value computed within the target.

APIs using this attestation of host capability are described in section 8.6.7. Details on attestation of host process are described in [A03].

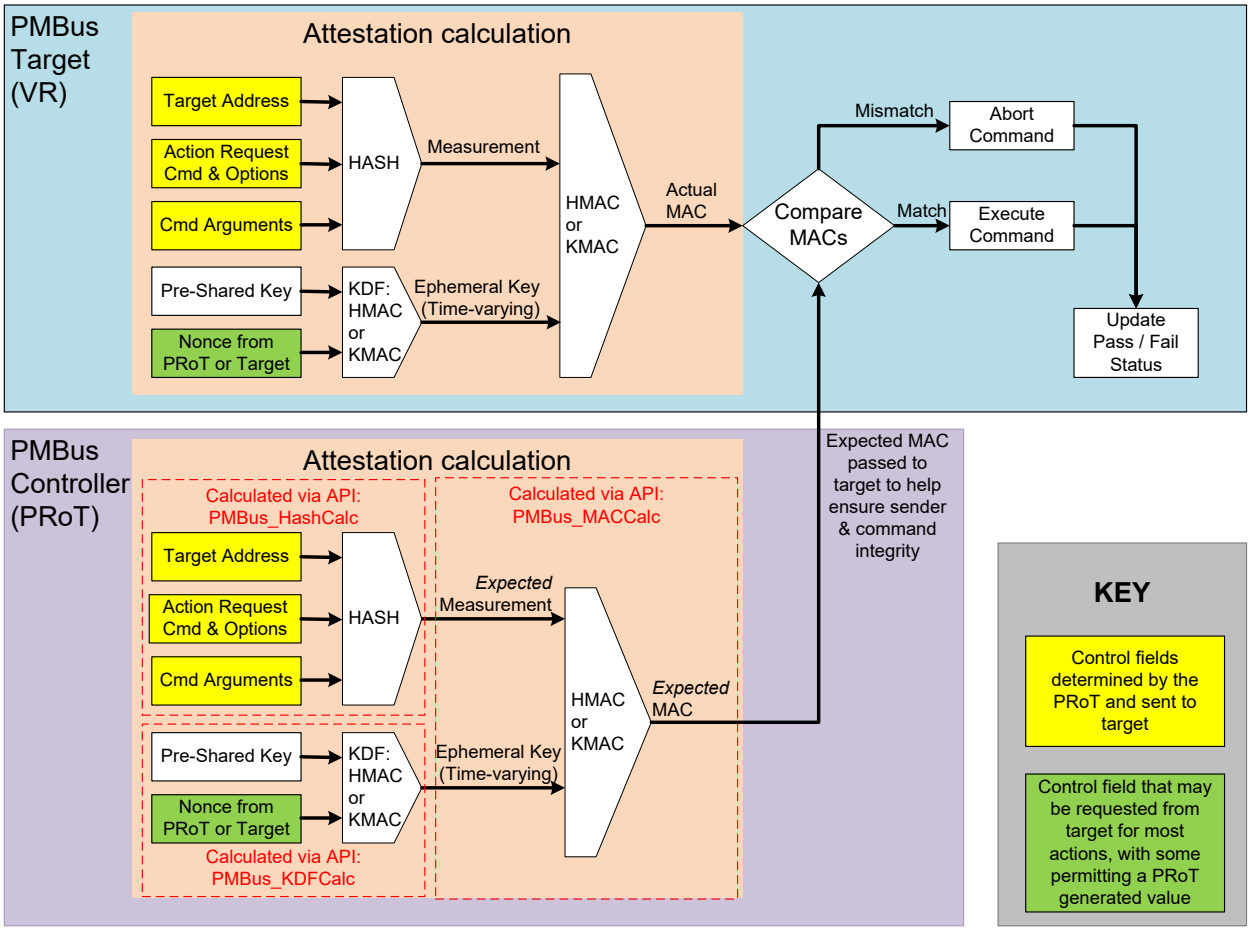

Figure 5-7. Process for Attestation of Host with Software APIs

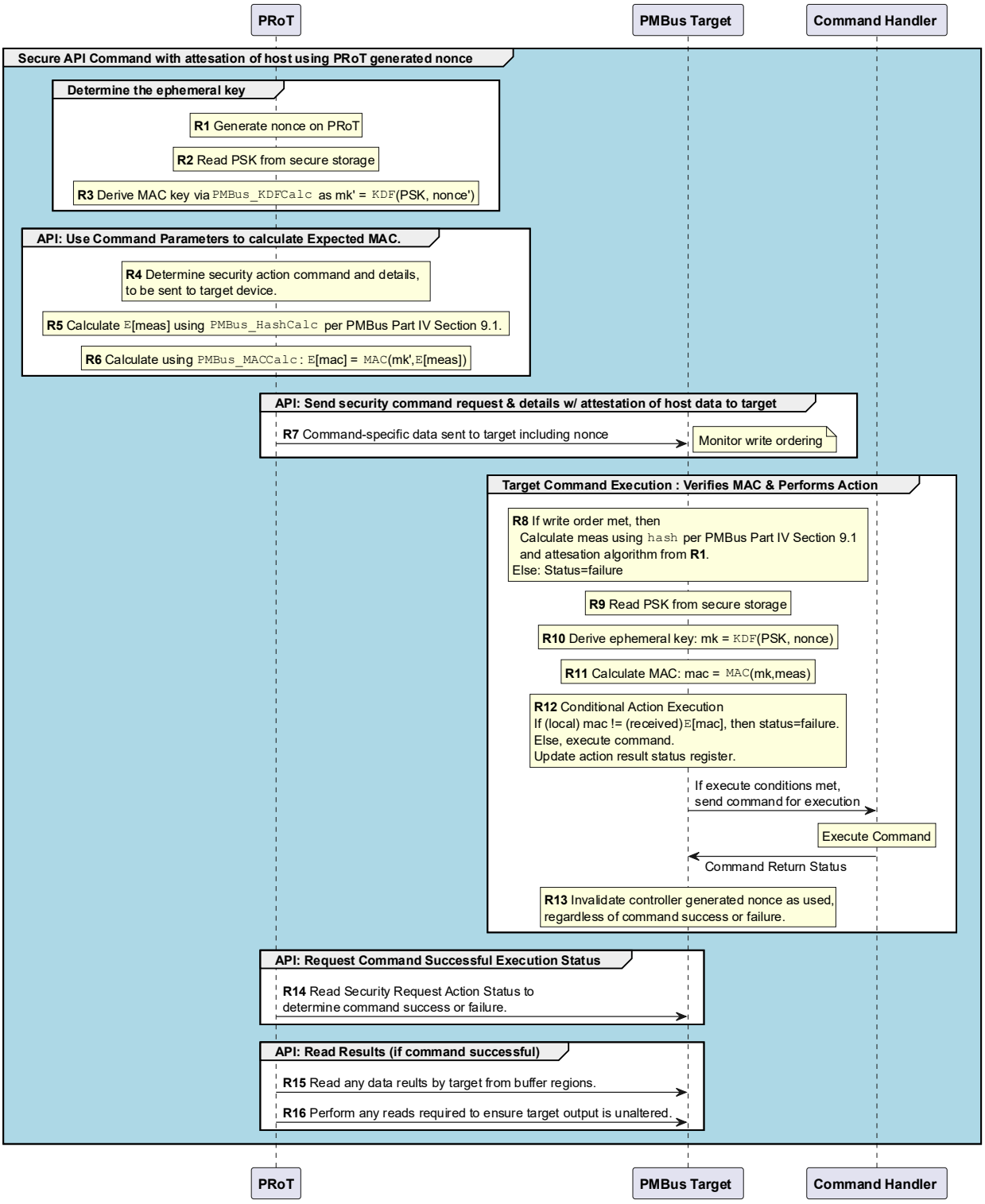

Figure 5-8. Command w/ Attestation of Host & PRoT Nonce (PSK Lock Forever / Iterate)

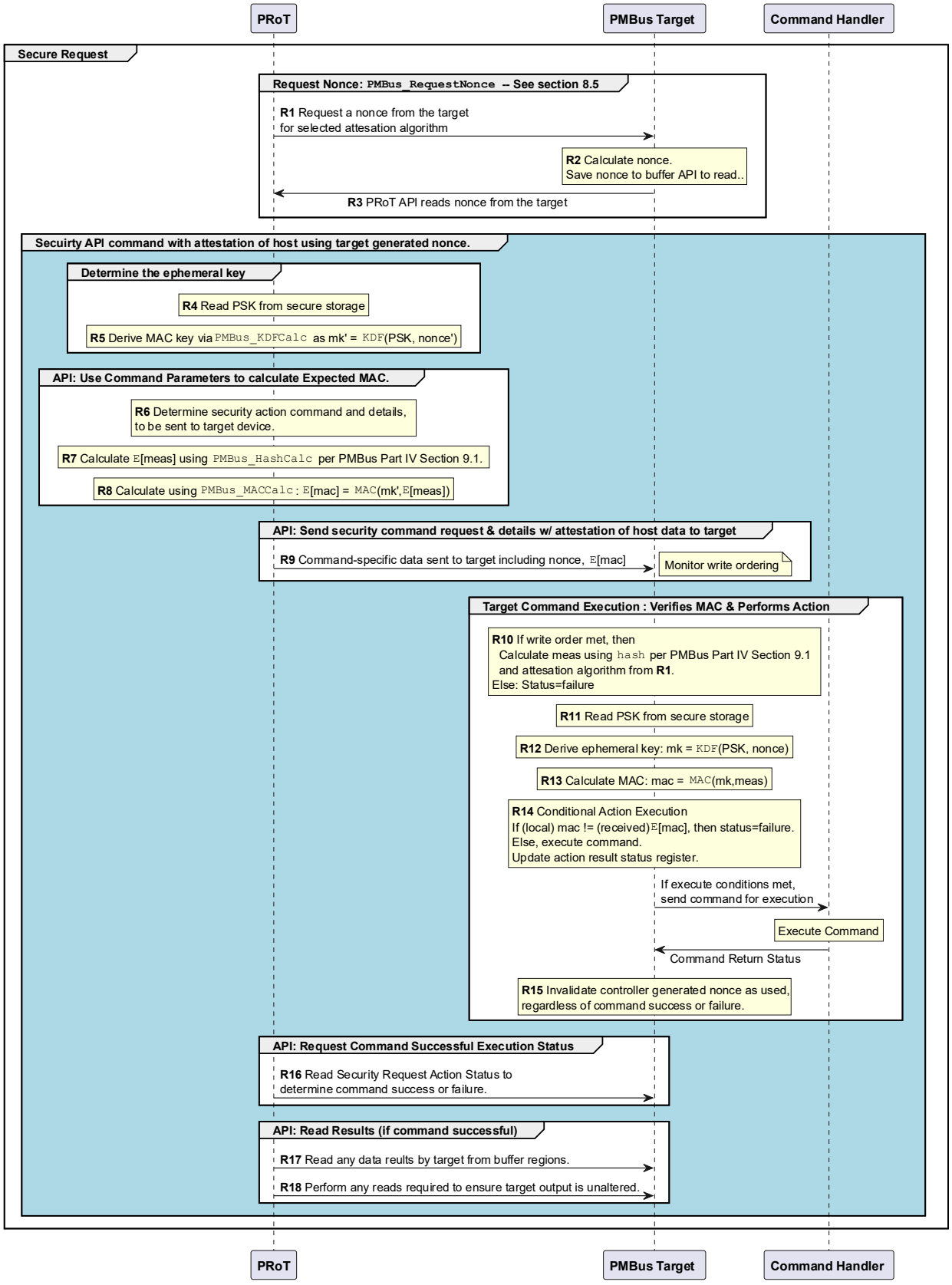

Figure 5-9. Command w/ Attestation of Host & Target Generated Nonce (Security Level 3)

5.3 Pre-Shared Key (PSK) Management

The pre-shared key plays a vital role in maintaining integrity of the attestation process. It is used in all security levels except 0. A command is defined in 8.2.1 to set the initial PSK (PSK_0) if the device does not come out of the factory with a PSK. This initial PSK command can only be safely utilized if three conditions are met: (1) it is invoked within a secure environment where no physical snoopers may listen to the bus, (2) it is only available if no initial PSK was set by the factory, and (3) the command becomes permanently unavailable after the initial PSK has been set. The initial PSK shall be the same length as the PSK resulting from the supported PSK iteration algorithm.

PSKs must be stored in a secure enclave, it cannot be openly readable. Ideally only the output of hashes would leave this enclave. Preservation of PSK secrecy is key to the attestation process. Care should be taken to eliminate or minimize the amount of time where the PSK is memory resident.

A command is defined in section 8.2.3 to allow iterating the PSK. Iteration presumes that the PSK is not compromised but that the PSK is being changed. This could involve moving from a manufacturer provided PSK to an instance specific one either at system integration or customer adoption. It could involve a rolling PSK that changes across time in a higher-threat environment. PSK iteration uses the same key derivation function (KDF) as is used to generate the ephemeral key from the PSK and nonce, only in this case the “ephemeral key” becomes the new PSK and the old PSK is deactivated. The combination of the secret static and non-secret dynamic component making it difficult for attackers to predict the next PSK. Both the PRoT (host) and target calculate the KDF, store the result as the new PSK, and deactivate the old PSK.

Commands are defined to identify the PSK iteration algorithms supported by a target as described in section 8.2.2, and (2) to lock the PSK as described in section 8.2.4. The lock can be applied either permanently or for the remainder of a power cycle only. Permanent locks can use a PRoT generated nonce, but power-cycle-only locks require a target generated nonce to be safe against replays.

For the standard hardware implementation corresponding to each reference API, refer to Table 11-1.

PSK provisioning is discussed further in section 6.

5.4 Firmware or Configuration Updates

5.4.1 Authenticated Update Overview

PMBus targets are increasingly becoming microcontroller based. This provides increased flexibility, telemetry, and even potential software-based fixes to power issues in deployed systems. With these new capabilities, new exploit possibilities are also opened. It is important to ensure that any firmware or configuration update accepted by the VR is validated to be: (1) from an authorized agent, (2) unmodified since release, and (3) newer than the existing firmware or configuration – not a reversion to earlier firmware.

There are two classes of firmware and/or configuration updates. One class contains items that must not be updated while a critical function, such as voltage regulator output is active. For a voltage regulator, this would contain items that affect loop dynamics such as filter bandwidths or internal gain settings within the controller. These updates must only be applied when the output is not enabled. The second class contains items which can be updated anytime. This includes items such as the

software to implement certain PMBus functions implemented in firmware, such as the security functions. The second class of updates is permitted to be applied immediately regardless of if device output is active or not. For each class, the updates may be applied in either a volatile manner or to the non-volatile memory.

These two classes imply the need for segmented memory updates, where updates are applied to portions of the volatile or non-volatile program memory. Segmented memory updates may also be needed to allow updates with limited RAM available on the VR to temporarily store a candidate firmware until it is authenticated.

Some PMBus target implementations may need to be effectively “rebooted” for the newly transferred image to take effect following a firmware or configuration update. The PMBus secure device application profile defines means to request a reboot. At security level 2 and below, the reboot request requires the VR output already be disabled and is presented in section 8.6.5. At security level 3, an attested host can also request VRs perform a full power cycle per section 8.5.6.

If either (a) the PMBus target is not pre-loaded with a full NVM image or (b) a firmware update is requested, then the PRoT must verify the new firmware’s signature using the public key stored securely within the PRoT. If a firmware does not pass authentication, the update flow is blocked, and the existing firmware shall continue to remain active. It must not be possible to generate a hardware “denial of service” attack via an improperly authenticated firmware-update. New firmware must be validated by the PRoT prior to overwriting the existing firmware. To preserve the integrity of this authentication flow, in normal operation no adjust-NVM-byte commands, such as PMBus STORE, are allowed as they would bypass the authentication. They must be locked out by access control per section 5.5.

Having a secure firmware/config upload capability enables potential low-NVM PMBus devices in future systems where the firmware is updated into volatile memory by the PRoT at each boot and attested to by the target prior to output enabling.

APIs for these functions are defined in 8.5. Refer to Table 11-1 to find standard hardware implementation definition within [A03].

5.4.2 PRoT Verification of Update Request

The PRoT device executes the firmware / configuration update in response to a request from an outside agent. To ensure those firmware / configuration update requests are made by an authorized system designer, the PRoT uses asymmetric cryptography.

The PRoT holds a public key. Firmware / configuration update requests must be signed by the requestor using the corresponding private key. It is the responsibility of the system designer to control the private key. For all PMBus security levels, the PRoT must successfully affirm the signature in the update request using its public key. If the signature verification fails, then the update request will be discarded before getting to the VR.

In limited cases where the VR itself is doing full public-private key signature verification (security level 3), it may be possible to skip the PRoT signature verification. PRoT verification of the incoming request is still strongly recommended to prevent an aggressor from introducing a large volume of bus traffic via failed update requests to the PMBus.

This step is separate from any authentication done within the PMBus target. The update package must include any information needed to perform the update, such as pointer to the appropriate driver code to use to calculate any values needed to do authentication within the target device.

5.4.3 Authentication Process

After incoming firmware / configuration update requests have been validated by the PRoT, the PRoT will attempt to push the update out to the PMBus targets / MBVRs. The targets themselves perform authentication before accepting the update image. This provides protection against a compromised device on the PMBus acting as a host driving firmware update requests. The method used for authenticating the request is a function of the security profile level and of the choice computing authentication over a segment or the full firmware image.

5.4.3.1 Segment of Full Image Verification

This document and the PMBus Specification Part IV security specification [A03] define firmware updates to be implementable via segments. Device manufacturers can decide their firmware topology with regard to number of segments, size(s) of each segment, volatile or non-volatile update capable. Additionally, the device manufacturer can decide if an update is applied, if the authentication criteria is computed only over the segment being updated or over the full image.

If the PMBus target authenticates updates using the whole firmware/memory image instead of just the segment to be updated, then it is preferred that non-volatile updates shall be applied only immediately following a PMBus target or power-on or reset cycle and prior to only volatile only updates. This avoids an ambiguity about which image to use when applying an NVM update when portion of the active memory may have had a volatile update applied over the non-volatile contents. An exception to this guideline is allowed where a device can commit all previously committed volatile segments along with a new non-volatile segment to NVM. It is permissible, but optional, for a device to allow simultaneous update to volatile and non-volatile memory.

This document defines APIs used to update firmware or configuration space. The PMBus Specification Part IV defines a standard interface implementation [A03].

5.4.3.2 Security Profile Levels 0 and 1 Authentication

In security levels 0 and 1, the PMBus secure device targets will use password protection prior to accepting updates. The recommendations below are for device manufacturers. Device manufacturers are responsible for the value of their parameter choices and the corresponding security strength.

Recommendations include:

- Limited password attempts, after which all further attempts are rejected until power cycle.
- Multi-byte password required to write target NVM or reconfigure target
- Password, meeting minimum requirements, is programmed into NVM by system vendor
- Password shall not be a constant across a model line, nor trivial.
- An example secure password is calculated via a strong hash function of motherboard instance-specific identifiers (example: serial number and other

factors) by an external entity such as BIOS or manufacturing environment, fused at platform assembly.

- Hash function itself needs to be obfuscated in any visible code or binary
- Vendors to provide guidance to end customers to ensure proper password adoption in systems

It is strongly recommended that a checksum also be integrated to ensure the integrity of incoming data and trigger update reject upon checksum failure.

5.4.3.3 Security Profile Level 2 Authentication

For security profile level 2, authentication of incoming firmware in the target is done using similar calculations as those used to authenticate the firmware. In security level 2, attestation is performed upon the candidate image prior to accepting the update. In this case, the PRoT supplies the target both the details about what to program and an expected attestation result. The target computes the attestation result (MAC) using the candidate image either over one firmware segment or the full image depending on selected update options.

If the expected attestation result transmitted by the PRoT to the target matches the value computed locally by the target itself, the firmware accepts the update for the selected segment.

Attestation calculation algorithm details and selection process is described along with the API in section 8.6.3 and the standard hardware interface description in [A03]. Individual customers will determine the hashing strength required for their application.

5.4.3.4 Security Profile Level 3 Authentication

For security profile level 3, the PMBus target itself required to verify the PMBus target firmware and configuration file image. In this case, the PRoT transmits to the target a firmware image including an embedded signature generated using a private key. The target itself must be preprogrammed with the corresponding public key. When the target receives the update request from the PRoT, it uses its public key to verify the incoming candidate image was signed using the private key corresponding to the public key carried within the target. If that signature is verified, then the target will accept the update for the selected segment.

The API for security level 3 updates is described in section 8.6.4 and the standard hardware interface description in [A03]. Acceptable signature algorithms are determined by the customer and their requirements. PMBus Specification Part IV Security specification [A03] contains recommendations.

5.4.4 Authenticated Update Space Remaining

An API, described in section 8.6.1, that indicates how many authenticated updates remain available. When this register reaches 0, there are no more authenticated updates to non-volatile memory remaining. The meaning of non-zero values from 1 to 6 is left to the manufacturer to determine. Value 7 means that the non-volatile firmware updates are effectively unlimited.

5.4.5 Command Called While Its Firmware Updates

It is allowable to NACK or assert STATUS_CML invalid command if a PMBus command is issued while the firmware used to implement that command is being updated.

5.5 Access Control

Access control is defined to restrict access to PMBus commands that may be utilized by an attacker to alter key system behaviors, while leaving commands open for intended telemetry configuration or monitoring. Examples of PMBus commands which could be used to provide temporary or permanent denial of service include VOUT_COMMAND, IOUT_OC_FAULT_LIMIT, and IOUT_OC_FAULT_RESPONSE and similar commands for overvoltage, undervoltage, and temperature.

Access Control methods are defined to enable managing PMBus visibility including restricting: general write access, general read access, allowing access to attested hosts, NVM store and restore access, and general write-once access. Furthermore, PMBus defines bits for the access control permissions to be locked either for the duration of the power cycle or permanently. Access control bits for PMBus commands are defined in [A02].

To minimize logic area, it is allowable for access control settings for similar groups to be controlled via the access control setting of a single command within that group. These groups may be determined on a per-application basis, though grouping recommendations are made within this document in section 9.

If the design implements security level 3, optional commands are defined to allow modification of permissions by an attested PMBus host. Otherwise, the access control must be configurable via initial settings, implemented by either NVM or fixed in metal for a given application. Access control requirements for devices meeting the PMBus Secure Device application profile are discussed in section 10.1.

6. Pre-Shared Key Requirements

During the manufacturing process of a unit, the manufacturer or system integrator may elect to generate a random number, PSK_0, to use as the initial pre-shard key (PSK). This PSK_0 is provisioned to both the PMBus target and the PRoT on the platform by the system integrator.

- The PSK shall be of bit length determined by the attestation algorithm supported.
- The PSK_0 can be generated by a reliable DRBG by the **manufacturer** or the **system integrator**. If PSK_0 is generated by the manufacturer, the manufacturer must provide the system integrator with a secure way to determine the PSK_0 of each device.
- After the PSK_0 has been programmed, the methods to write PSK_0 **must be** irreversibly fused off to ensure system security. It is recommended that methods used to track the PSK of individual VRs minimize any opportunity for accidental disclosure.
- Any future PSK updates, including those applied by system integrator to change manufacturer-set PSK_0, must be done via the PSK iteration API described in 8.2.3.

The PSK_0 is to be programmed only in a secure and controlled environment using the API PMBus_ProvisionPSK0 of section 8.2.1. The basic flow for provisioning PSK_0 into the PMBus host (PRoT) and the PMBus Target (MBVR) is shown in Figure 6-1.

If PSK_0 is fused by the manufacturer, it is mandatory to provide the system integrators with a secure method to determine the PSK on each device. This method **must** be irreversibly fused off by the system integrator after assembly to ensure system security. It is recommended that methods be utilized to track the PSK to the individual VR that have zero opportunity for accidental disclosure.

It is strongly recommended that the PSK used for each VR in the system be unique, thus preventing commands sent to one VR from being replayed to different VRs. This possible exploit is also prevented by using the guidelines set forth in [A03] by including data about the target in the data hashed for command acceptance.

A system integrator may elect for it to iterate PSKs in an internal trusted environment only, prior to product being released into use. A means is defined for PSK iterate to be permanently locked out by the system integrator prior to deploying a system to prevent any malicious update request. In other cases, with agreement of the system integrator, the end-user may want to be responsible for applying a further update to the PSK and possibly subsequently locking the PSK on their own.

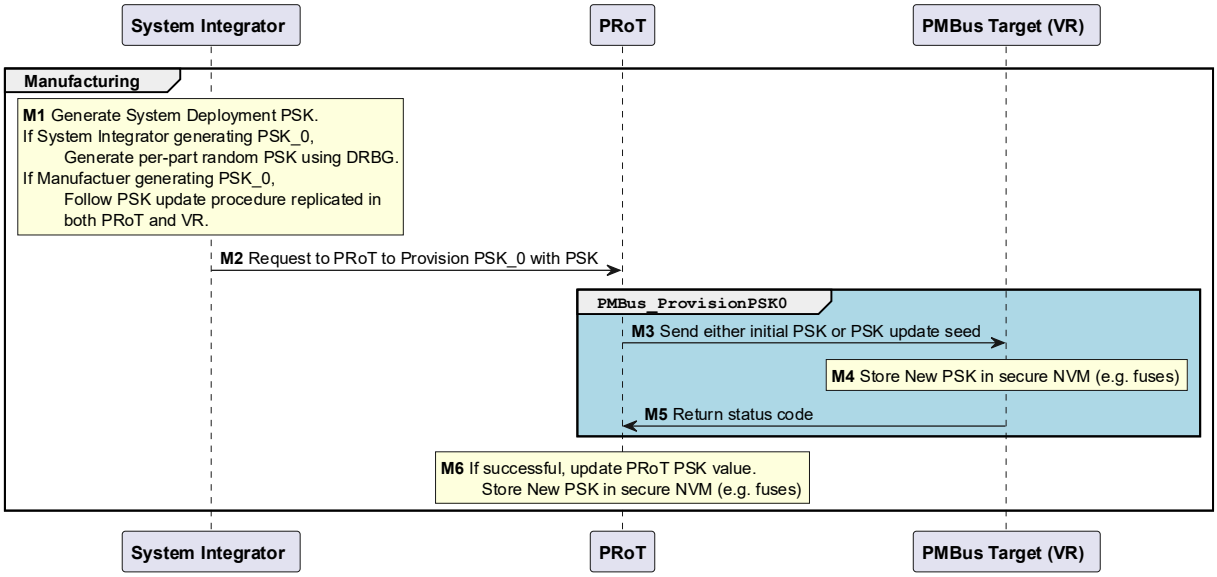

Figure 6-1. PSK Provisioning Process

7. Deployment Modes

Attestation allows confirming that the expected PMBus target device is present prior to allowing system boot. If the device is already pre-programmed with the firmware and configuration expected by the PRoT, then the boot flow proceeds to validate the next rail.

If attestation fails, the PRoT may elect to determine if the correct PMBus device is present by querying commands such as MFR_ID, MFR_MODEL, MFR_REVISION, IC_DEVICE_ID, and IC_DEVICE_REVISION. If the target is the correct model, then the PRoT may attempt to program the PMBus device with the proper firmware and configuration. This flow can be used in two scenarios. In the first scenario, this addresses a VR which may operate with partial NVM configuration. This may allow the PRoT to issue a unique VR configuration based upon what SoC is plugged into the socket. In a second scenario, the VR programming may have been unexpectedly altered. In this case, the

PRoT can attempt to restore the signed image. If the reprogrammed VR then attests successfully, the boot flow may continue to the next rail.

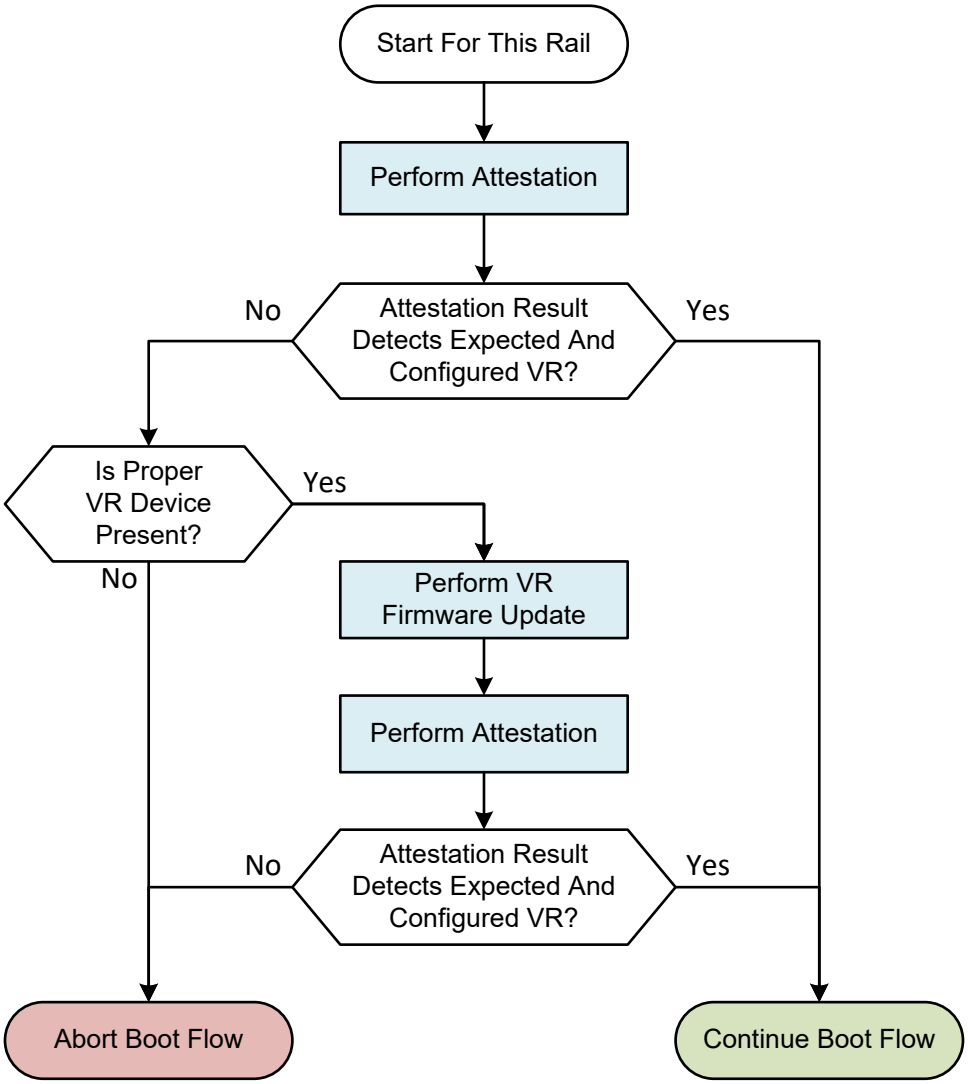

Figure 7-1. VR Firmware Update Flow Including Low/No NVM Option

If the PMBus target has secure storage space available for a replacement PSK, then it is possible to allow the system integrator to migrate from PSK_0 to PSK_1 when the system is built. This can be done using the PRoT and its DRBG.

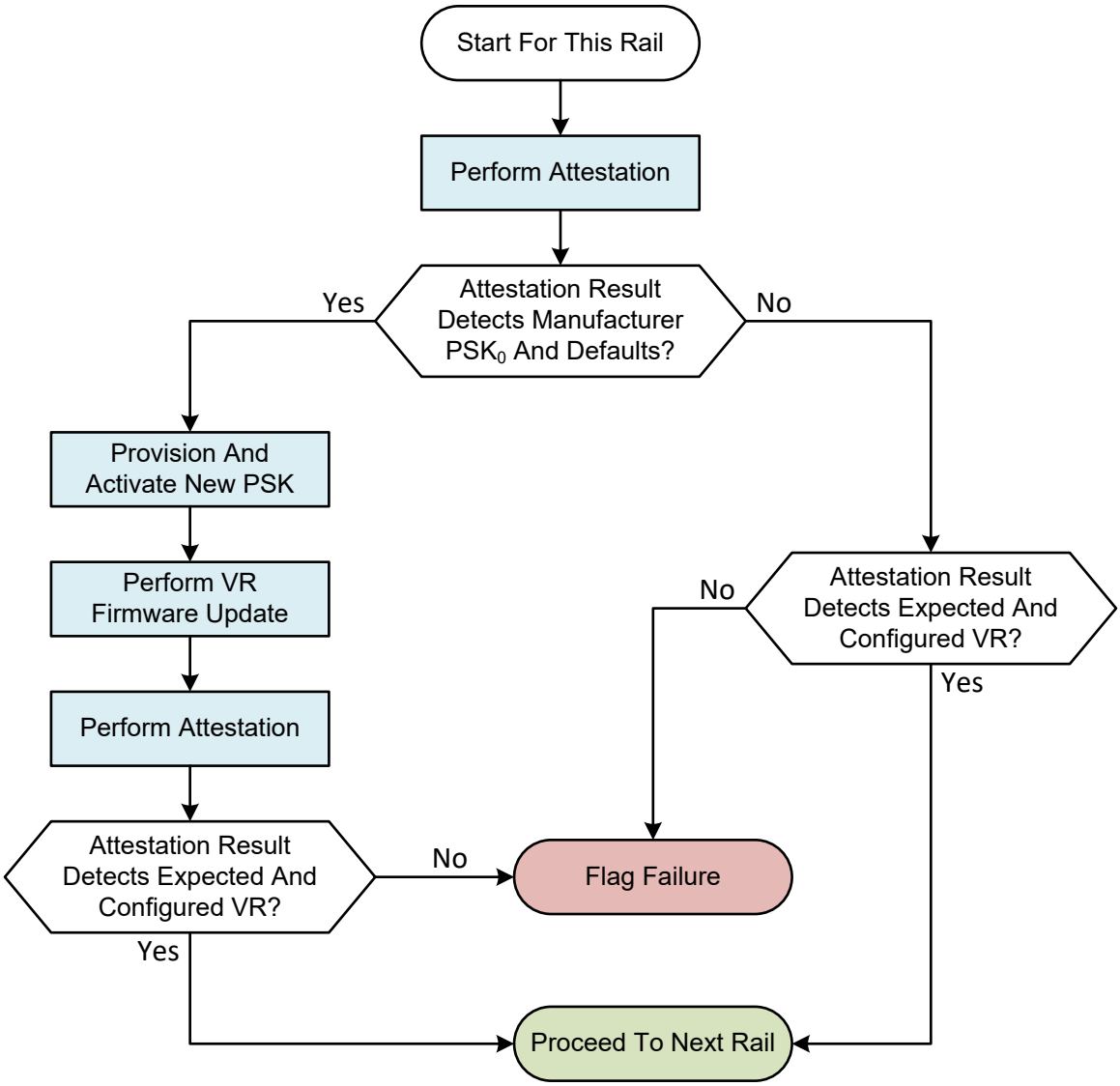

Figure 7-2. VR Firmware and PSK Update by System Integrator

8. Standard Application Programming Interface

PMBus Specification Part IV [A03] defines a standard set of capabilities and hardware implementation to harden PMBus devices and the platforms utilizing them against attack. It defines commands, and a register-level implementation used to access them. A standard API was defined to provide implementation flexibility and abstract away two hazard scenarios:

- Changes during specification review that can render devices developed against the draft specification non-compliant, for example: refinement of a bit definition.
- Differences in implementation of [A03] between suppliers.

These differences can be addressed by using vendor-specific API implementation. This paper defines an API wrapper for functions utilized in the PMBus Specification Part IV, Revision 1.5. It does not address the theory behind the PMBus security functions themselves, which can be found in the PMBus Specification.

The APIs defined in this document are made available for industry deployment of PMBus secure device application profile. The APIs are defined in both C and Python format. Within the arguments table used for each API, the datatype column contains two values. The first data type value corresponds to the data type for the C implementations of the API. The second data type, shown within square brackets, is the Python data type.

8.1 API Summary

Table 8-1 provides a summary of the PMBus secure device application profile defined functions, a brief description, and if they are required to support PMBus secure device application profiles. The standard hardware accessed by the API is defined in [A03] sections given by Table 11-1.

Table 8-1. API Summary

API	Section	Purpose	Required
PSK Management			
PMBus_ProvisionPSK0	8.2.1	Commit Initial PSK	Optional if all devices are manufactured with an active PSK, otherwise mandatory for all security levels ≥1
PMBus_ReqNewPSK_Algo	8.2.2	Report supported algorithms for PSK update	Mandatory at all security levels ≥1
PMBus_ReqNewPSK	8.2.3	Iterate PSK using a randomized seed	Mandatory at all security levels ≥1
PMBus_LockPSK	8.2.4	Lock the PSK	Optional if coverage provided by NVM update, else mandatory for all security levels ≥1
Attestation			
PMBus_AttestTarget	8.3.1	Requests attestation of target device, retrieves and compares expected MAC result	Required for all security levels ≥1
PMBus_AttestationAlgoSupport	8.3.2	Reports supported attestation algorithms	Required for all security levels ≥1
PMBus_ReqAttestTarget	8.3.3	Requests PMBus attestation of target. First half of PMBus_AttestTarget call	Recommended but Optional, may be lower-level and not exposed

API	Section	Purpose	Required
PMBus_RetrieveAttestTarget	8.3.4	Retrieve PMBus attestation calculation. Second half of PMBus_AttestTarget after target has completed MAC calculation.	Recommended but Optional, may be lower-level and not exposed
PMBus_HashCalc	8.4.1	Calculates the hash measurement (digest) for a string of incoming data for a selected attestation algorithm. Should be an internal function used within API implementations.	Recommended but optional, may be lower-level and not exposed
PMBus_KDFCalc	8.4.2	Calculates the ephemeral key from a nonce and PSK for a given attestation algorithm. Should be an internal function used within API implementations.	Recommended but optional, may be lower-level and not exposed
PMBus_MACCalc	8.4.3	Calculates the message authentication code from an ephemeral key and a measurement for a given attestation algorithm. Should be an internal function used within API implementations.	Recommended but optional, may be lower-level and not exposed
Firmware Update Commands See Requirement level below.			
PMBus_NewFwUpdatesRem	8.6.1	Indicates the amount of NVM free for firmware updates	Required for all security levels
PMBus_NewFwCommitSL1	8.6.2	Attempt to commit a candidate firmware or configuration using security level 1	Required for security level 1

API	Section	Purpose	Required
PMBus_NewFwCommitSL2	8.6.3	Attempt to commit a candidate firmware or configuration using security level 2	Required for security level 2
PMBus_NewFwCommitSL3	8.6.4	Attempt to commit a candidate firmware or configuration using security level 3	Required for security level 3.
PMBus_Reboot	8.6.5	Reboots the target if its output is already off. Used following a firmware update.	Required for all security levels if target needs reboot to complete update.
PMBus_NewFwDownload	8.6.6	Transfer a candidate firmware image from target	Optional
PMBus_FetchFw	8.6.7	Fetch a firmware segment from NVM into the target's buffer region	Optional
Security Level 3 Commands requiring Attestation of Host			
PMBus_RequestNonce	8.5.1	Request a nonce for commands requiring attestation of host. This generates the nonce required by remainder of commands in this section.	Required for security level 3.
PMBus_Secure_Access_Control	8.5.2	Call the PMBus access control command with added security resulting from attestation of the host and transmitted data contents	Required for security level 3.
PMBus_Secure_PagePlus	8.5.3	Call the PMBus Page Plus command with added security resulting from attesting the host and transmitted data contents	Required for security level 3.

API	Section	Purpose	Required
PMBus_Secure_AlertConfig	8.5.4	Changes the status of if SMBALERT# is asserted upon completion of a security command	Required for security level 3.
PMBus_RebootLockout	8.5.5	Allows blocking of reboot requests until a set of firmware updates has completed successfully.	Optional
PMBus_RebootFromOn	8.5.6	Provides ability to request a target to power down its output, reboot, and then power-on its output from an attested host.	Required for security level 3.
PMBus_FetchFwLockout	8.5.7	Prevent fetching firmware into the loading buffer	Optional
Miscellaneous Commands			
PMBus_Profile_SecurityVersion	8.7.1	Reports the security version supported by the target	Required for all security levels
PMBus_Device_FwConfigVersion	8.7.2	Reports the firmware and configuration version supported by the target	Required for all security levels
PMBus_Device_Profile	8.7.3	Gathers characteristics which dictate communication of other functions. Relevant fields read including PMBus security command code capability and supported block sizes.	Required for all security levels

8.2 APIs for PSK Management

The following functions relate to PSK initiation and management. They support the PSK functions described in PMBus Specification Part IV. The mappings of APIs to their detailed standard hardware specification, including specific hashing algorithms are listed in Table 11-1.

8.2.1 PMBus_ProvisionPSK0

The following API provides a means to program the first PSK, PSK_0, into the PMBus target. This API should not be part of a normal distribution. After being successfully

being called for a target endpoint, this API should be permanently locked out for that target regardless of power cycle. This command must never be available outside of a secure manufacturing environment. The API prototype is:

C variant

```
int PMBus_ProvisionPSK0 (int      devHandle,
                         uint8_t  pmbAddr  ,
                         int16_t  page     ,
                         uint8_t  psk0_len ,
                         uint8_t *psk0_x   )
```

Python variant

```
def PMBus_ProvisionPSK0( devHandle,
                         pmbAddr  ,
                         page     ,
                         psk_len  ,
                         psk0_x   )
```

In this command, the psk0_x is a pointer to an array of unsigned 8-bit values. Typically, this array is 32 unsigned 8-bit integers long, but the exact length will depend on the device being targeted by the API. Array element 0 bits [7:0] contain PSK bits [7:0]. Array element 1 bits [7:0] contain PSK bits [15:8], and so on. This API returns 0 if the operation completed successfully. If programming failed due to PSK_0 already having been written it returns -1. Other failures cause an integer return of -2.

Table 8-2. PMBus_ProvisionPSK0 Arguments

Flow	Data Type C/[Python]	Argument	Definition
input	uint8_t [int]	psk0_len	This is the length of the psk0_x array in bytes that is used to set the initial PSK. Length shall be the same as the PSK produced by the PSK iteration algorithm.
input	uint8_t * [bytearray]	psk0_x	psk0_x is a pointer to an array of bytes (uint8_t) with index 0 of the array equal being the 8 least significant bits. Each incrementally higher index contains the next 8-bits. The length of the array should be equal to the size known to be utilized by that PMBus target. The API itself programs the appropriate length for the voltage regulator. In all present PSK implementations, the PSK is 256 bits (32 bytes) long.
output	int	Return Value	0: PSK0 successfully provisioned and PSK0 provisioning function permanently locked out -1: PSK0 setting failed -2: Other failures

8.2.2 PMBus_ReqNewPSK_Algo

The following API indicates the PSK iteration algorithms implemented from the permitted list in the corresponding [A03] section, indicated for this API via Table 11-1. The algorithms use the current PSK and a seed value to determine the next PSK.

C variant

```
int PMBus_ReqNewPSK_Algo(int       devHandle   ,
                         uint8_t   pmbAddr     ,
                         int16_t   page        ,
                         uint32_t *algoSupport ,
                         uint8_t  *remainingPSK)
```

Python Variant

```
def PMBus_ReqNewPSK_Algo( devHandle    ,
                          pmbAddr      ,
                          page         ,
                          algoSuport   ,
                          remainingPSK )
```

It accepts two pointers where it stores its output. The parameter `algoSupport` is a pointer to an unsigned 32-bit integer indicating supported algorithms. The `remainingPSK` points to a value indicating if there is room to store an iterated PSK.

Table 8-3. PMBus_ReqNewPSK_Algo Arguments

Flow	Data Type C/[Python]	Argument	Definition
output	`uint32_t [int]`	`algoSupport`	Algorithm support is a pointer to a 32-bit unsigned integer. The pointed to integer is updated by this API. Each bit location within the integer corresponds to a PSK iteration algorithm from [A03], mirrored in this document as Table 11-2. Bit Location — Description [0] (LSB) — PSK iterate algorithm 0 supported [1] — PSK iterate algorithm 1 supported [2] — PSK iterate algorithm 2 supported [3] — PSK iterate algorithm 3 supported Other Bits — Set to 0. Reserved for future algorithms.
output	`uint8_t * [dictionary with key "value" storing int]`	`remainingPSK`	Points to an unsigned integer the API loads with the number of remaining PSK iterations requests supported. Value — Description 0 — No more PSK iteration requests supported 1…6 — 1…6 more PSK iteration requests supported 7 — More than 6 PSK iteration requests supported. Unlimited updates also returns value 7 Else — Reserved for future applications

Flow	Data Type C/[Python]	Argument	Definition
output	`int`	`Return Value`	0: Successfully retrieved the PSK iteration algorithm support and the number of PSK iterations remaining -1: Failed to retrieve the PSK iteration algorithm support or the number of PSK iterations remaining successfully provisioned and PSK0 provisioning function permanently locked out

8.2.3 PMBus_ReqNewPSK

A new PSK value can be requested in a deployed system, but this PSK must not be sent in the clear over the PMBus where an aggressor may snoop it. Consequently, the new PSK must be calculated from the existing PSK and a transmitted random seed using a key derivation function (KDF). The PRoT and the PMBus target will both compute the new PSK from these two components and store the result. When the new PSK is stored, the prior PSK is invalidated.

It must not be possible for an attacker to initiate a PSK iteration. If there were no protection, an attacker would be able to iterate a PSK without knowing the original. Although the attacker would not know the new resulting PSK to issue commands, a PRoT unaware of the iteration would be blocked from successfully attesting the target. This may render a system inoperable.

Incoming new PSK (iteration) requests must be attested to ensure that the sender knows the existing PSK. In this isolated scenario successful request completion results in a new PSK. Consequently, protection against replay attacks is of decreased significance. Replaying the same request would fail as the underlying PSK has changed due to the first request. Thus, although the command request is attested similarly to the “attestation of host” in [A03], the target does not need to generate the nonce used to attest the PRoT / host knows the existing PSK. This allows running this PSK iteration command at security level 1 and higher, and not requiring the target generated nonce of security level 3. At security level 3, either nonce is an option.

The PRoT itself shall hold its copy of the prior PSK until such time as it has validated the new PSK has been successfully applied via a completed attestation or similar request. This will make it difficult to lose any ability to send commands in the event of a failed PSK updated request.

The following API provides a means to request a new PSK be calculated from a given seed and the prior PSK according to [A03] in the section number resolved via Table 11-1.

C variant

```
int PMBus_ReqNewPSK(int       devHandle      ,
                    uint8_t   pmbAddr        ,
                    int16_t   page           ,
                    uint8_t   attestAlgo     ,
                    uint8_t   psk_len        ,
                    uint8_t*  psk_x          ,
                    uint8_t   pskIter_algo   ,
```

```
                        uint8_t   seed_len       ,
                        uint8_t  *seed_x         ,
                        uint8_t   nonce_len      ,
                        uint8_t  *nonce_x        ,
                        uint8_t  *newPsk_x       )
```

Python variant

```
     def PMBus_ReqNewPSK(devHandle, pmbAddr, page,
                         attestAlgo, psk_x, psk_len,
                         pskIter_algo      ,
                         seed_len , seed_x ,
                         nonce_len, nonce_x,
                         newPsk_x          )
```

In this command, the `seed_x` is a pointer to an array of unsigned byte values of length specified by `seed_len`. Array element 0 contains seed bits [7:0], and upwards for each incrementally higher element.

This request can work in one of two ways. In security levels 1 to 3, the request can be performed using a PRoT generated nonce. Because execution of this command results in a new PSK, there is not a risk of replay provided the command data including target address is included in the hashed packet data.

This verification that the requestor (host) knows the PSK is described in the [A03] section “PMBus Host Attestation”. The process requires the request include an expected MAC. The formula for calculating the expected MAC must include the command, the seed, a nonce, the iteration algorithm selected, and the device address, as described in [A03] and mirrored in Table 11-7 to feed the calculations of measurement (section 8.4.1), ephemeral key (section 8.4.2), and message authentication code (section 8.4.3). The nonce in this calculation can be generated by the PRoT in security levels 2 and below. It is mandatory in security level 3 and optional in lower levels that the nonce be generated by the target.

This API returns 0 if the operation completed successfully. It returns a negative integer if it was unable to successfully update the PSK using the new seed.

In response to this command, the PMBus target begins the processing of computing the new PSK. The PRoT itself must also update its pre-shared key with the same seed. These new PSK become immediately effective. This command should be followed by a new attestation request.

The sequence used by this API using a PRoT generated nonce is shown in [A03] Figure “PSK Iteration Flowchart”. The sequence used by this API using a target generated nonce is shown in [A03] section as indicated from Table 11-1.

Table 8-4. PMBus_ReqNewPSK Arguments

Flow	Data Type C/[Python]	Data Type	Definition
input	uint8_t [int]	attestAlgo	This byte indicates the algorithm used by the target for attesting the seed and command parameters.

Flow	Data Type C/[Python]	Data Type	Definition
input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation algorithm to ensure the PSK iterate request is authentic.
input	uint8_t * [byteArray]	psk_x	This is the actual existing PSK used to compute the expected MAC for the target. The PSK is one input to this algorithm; combined with the nonce it makes an ephemeral (time-varying) key. The ephemeral key hashes a measurement of data that includes the seed, the PSK iterate command, the target, and other factors per [A03], mirrored in Table 11-2 for ease of reference. The least significant byte is in index [0] of the array.
input	uint8_t [int]	pskIter_algo	Selects a PSK iteration algorithm index from the list of [A03], mirrored in Table 11-2 for ease of reference. A value of FFh indicates the driver can choose the algorithm.
input	uint8_t [int]	seed_len	Specifies the length of the seed used to iterate the PSK, stored in seed_x, as an integer number of bytes. Seed lengths of up to 255 bytes are allowed.
Input	uint8_t * [bytearray]	seed_x	Pointer to a seed, represented as an array of bytes (unsigned 8-bit integers). The least significant byte is index 0 of the array. The array must contain the appropriate number of indexes for the seed length requested by the selected PSK iteration algorithm.
input	uint8_t [int]	nonce_len	This is the length of the PRoT-generated nonce used to perform attestation calculations. If nonce_len == 0, then the API will use a target generated nonce (requiring hardware support in target).
input	uint8_t * [bytearray]	nonce_x	A pointer to an array of bytes that hold the PRoT-generated nonce. The length of this array is determined by nonce_len. It can be a null pointer if nonce_len == 0, indicating the API will request and utilize a target-generated nonce. The least significant byte is in index [0] of the array.
output	uint8_t [byteArray]	newPsk_x	This is the PSK resulting from the PSK iterate function. It is determined by the API and NOT read back from the target as the new PSK must not be transmitted in the clear. It is of length psk_len as iterating the PSK should not change its length. The least significant byte is in index [0] of the array.

Flow	Data Type C/[Python]	Data Type	Definition
output	int	Return Value	0: PSK iteration successfully completed -1: PSK iteration failed as the target was out of writeable PSK storage. -2: PSK iteration failed as the requested algorithm was not supported -3: Attestation of incoming command and seed failed -4: PSK is locked, iteration request failed.

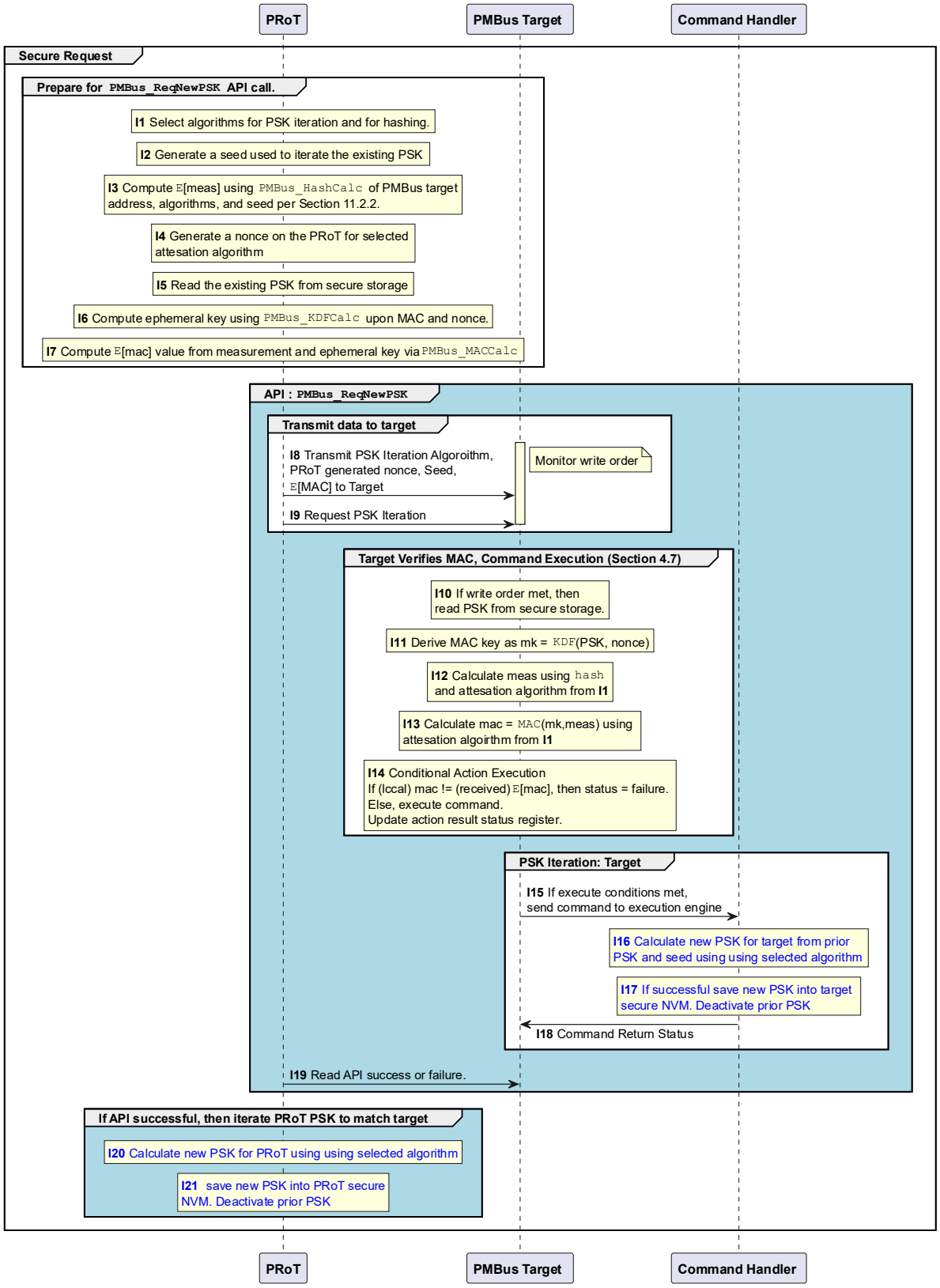

Figure 8-1. Pre-shared Key Iteration with PRoT Nonce

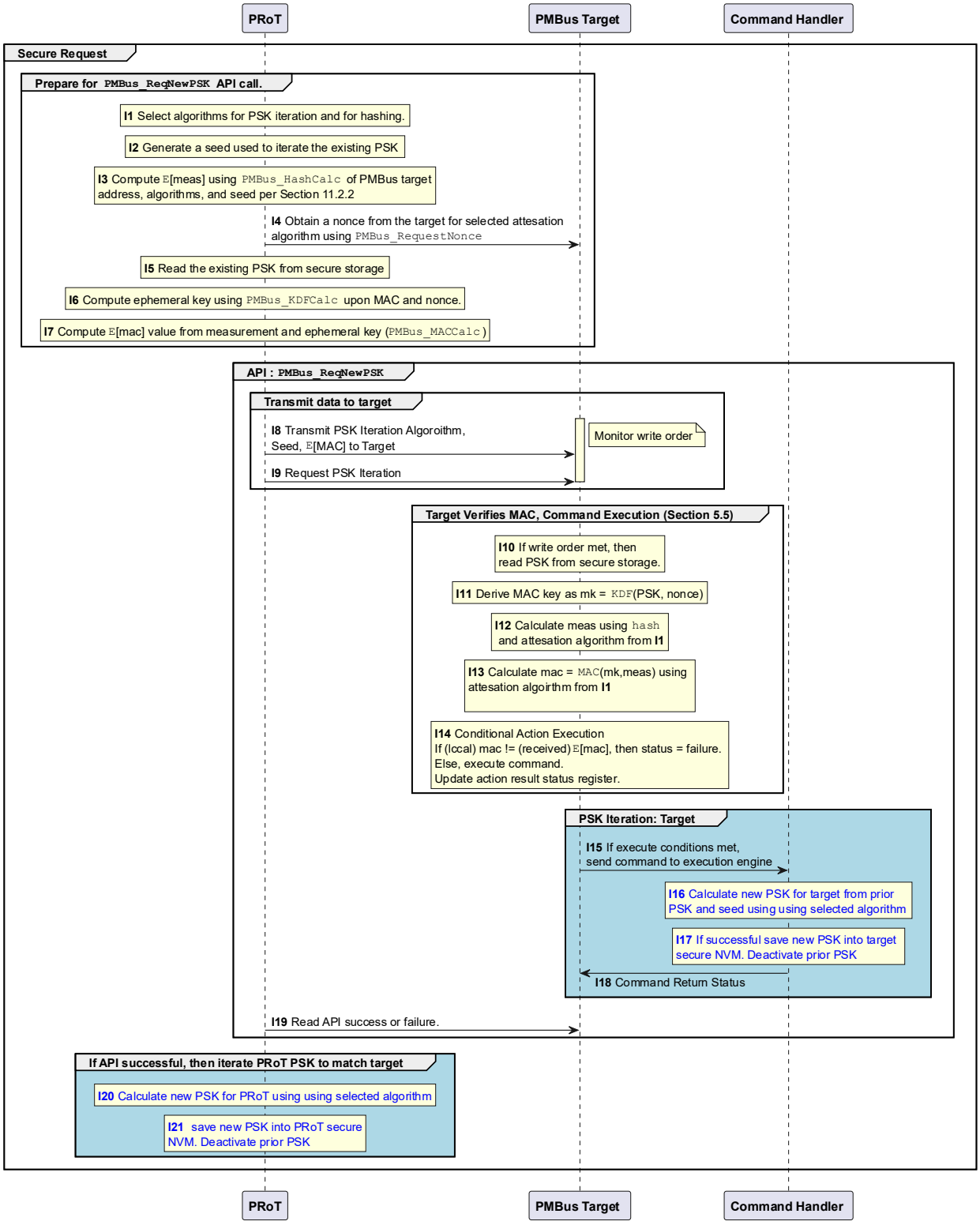

Figure 8-2. Pre-shared Key Iteration with Target Nonce

8.2.4 PMBus_LockPSK

The following API provides a means to lock out PSK changes. In security levels 1 and 2, the command uses a PRoT generated nonce to lock out PSK changes forever. In security level 3, this command uses a target generated nonce which enables a choice of applying the PSK lock either forever or for the remainder of a power cycle.

Locking out the PSK for only the remainder of a power-on-reset cycle requires a target generated nonce to be safe against replay attack. A target generated nonce is only required in security level 3. A PRoT generated nonce is acceptable for permanent PSK-lock function at security levels 1, 2, and 3. Replaying the command would just lock the already-set lock. Security level 0 does not use a PSK so the command is not relevant.

The function returns zero if the PSK is successfully locked against future updates as requested or it was already locked at an equal or higher level. It returns -1 if the lock request fails, and -2 if the lock type is not supported by the target.

C variant

```
int PMBus_LockPSK(int      devHandle   ,
                  uint8_t  pmbAddr     ,
                  int16_t  page        ,
                  uint8_t  attestAlgo  ,
                  uint8_t  psk_len     ,
                  uint8_t *psk_x       ,
                  uint8_t  nonce_len   ,
                  uint8_t *nonce_x     ,
                  uint8_t  porCycleOnly)
```

Python variant

```
def PMBus_LockPSK( devHandle, pmbAddr, page,
                   attestAlgo, psk_len, psk_x,
                   nonce_len, nonce_x,
                   porCycleOnly      )
```

Once the PSK has been locked, any request to update the PSK will fail and report an error-code return value per Table 8-5.

PSK lock may be set via a dedicated hardware mechanism or may be set via a dedicated firmware update. In the case PSK lock is only set via authenticated update, then this API would be replaced by an authenticated update API call.

The PSK lock can only be cleared through an authorized firmware update that would set the PSK unlock bit back to 0. It cannot be cleared via this API call.

Table 8-5. PMBus_LockPSK Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	porCycleOnly	If 0, then the PSK is to be locked permanently in NVM. If 1, then the PSK is to be locked in for the remainder of the PMBus target power-on reset cycle. Other values: Reserved.

Flow	Data Type	Argument	Definition
input	uint8_t [int]	nonce_len	This is the length of a PRoT-generated nonce used during the PSK lock operation. If 0, the nonce used for PSK lock is generated by the target rather than the PRoT. The pointer nonce_x is not utilized, and its length is 0. If ≥1, the nonce used is generated by the PRoT and passed to this API via the pointer nonce_x, with a length of nonce_len bytes. This is only permitted when porCycleOnly == 0.
input	uint8_t * [bytearray]	nonce_x	A pointer to an array of bytes that holds the nonce. The length of this array is determined by nonce_len. Its contents are transmitted to the target by the PRoT only if the PRoT generated the nonce is used. The least significant byte is in index [0] of the array.
input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation algorithm to ensure the PSK lock request is authentic.
input	uint8_t * [bytearray]	psk_x	This is the PSK used to compute the expected MAC for the target. The PSK is one input to this algorithm; combined with the nonce it makes an ephemeral (time-varying) key. That key hashes data that includes the lock type, the PSK lock command, the target, and other factors per [A03] section “PMBus Host Attestation” mirrored in Table 11-7 for convenience. The least significant byte is in index [0] of the array.
input	uint8_t [int]	attestAlgo	This byte indicates the algorithm used for attestation algorithm to be used. If porCycleOnly == 1, then this attestation algorithm works per [A03] attesting the host with a target-generated nonce. If porCycleOnly == 0, then this attestation algorithm works per [A03] sections as indicated by Table 11-1 typically with a PRoT generated nonce. Executing a porCycleOnly == 0 request with a target generated nonce is allowed when target supports it.
output	int	Return Value	0: PSK is locked as requested. -1: API was unable to lock PSK as requested (MAC failure). -2: Operation not supported (including PRoT generated nonce and (porCycleOnly == 1) lock request.

Table 8-6. New PSK Hash Requirements

Measurement Hash	Required Components
All hashing algorithms	Hashed message data must include PMBus target 7-bit address, power-on-reset cycle only argument and the attestation algorithm. Refer to PMBus secure device specification for standard implementation. An overview of commands with host attestation are shown in [A03] section “PMBus Host Attestation” mirrored in Table 11-7 for convenience.

8.3 APIs for Attestation of Target

The following functions related to attestation of a PMBus target device.

8.3.1 PMBus_AttestTarget

The attestation process can be called via a single API. This API itself is implemented from the lower-level API’s described throughout sections 8.3 and 8.4. The API that takes care of the entire attestation process is as follows:

C variant

```
int PMBus_AttestTarget(int      devHandle,
                       uint8_t  pmbAddr,
                       int16_t  page,
                       uint8_t  attestAlgo,
                       uint8_t  psk_len,
                       uint8_t *psk_x,
                       uint8_t  nonce_len,
                       uint8_t *nonce_x,
                       uint8_t  expMeas_len,
                       uint8_t *expMeas_x);
```

Python variant

```
def PMBus_AttestTarget( devHandle, pmbAddr, page,
                        attestAlgo, psk_len, psk_x,
                        nonce_len, nonce_x,
                        expMeas_len, expMeas_x  )
```

This command requests the attestation of the firmware and configuration stored within the PMBus target. It requests the target perform attestation calculation (API described in Section 8.3.3), waits for the results to become available, retrieves the results (API described in Section 8.3.4), and compares the results. It returns 0 if the attestation is successful and a negative number if the attestation result fails. If a device chooses to pipeline attestation (sequentially perform a request attestation to each VR, and then sequentially retrieve the resulting MAC from each VR) then it can call the lower-level API directly.

The parameters and API return value for the full PMBus attestation calculation are described in Table 8-7. If this API returns zero, the attestation was successful. If the API returns a negative integer, the attestation was unsuccessful, and remedial action should be taken to address the firmware and/or configuration mismatch.

Table 8-7. PMBus_AttestTarget Arguments

Flow	Data Type	Argument	Definition
input	`uint8_t [int]`	`attestAlgo`	Passes the bit index of the attestation algorithm that the PMBus target is to apply in calculating the expected MAC (refer to [A03] mirrored in section 11.2.2 for supported algorithms).
input	`uint8_t [int]`	`psk_len`	This is the length in bytes of the PSK used by the attestation algorithm.
input	`uint8_t * [bytearray]`	`psk_x`	This is the PSK used when calculating the message authentication code (MAC). This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the firmware and configuration measurement to make the MAC. The least significant byte is in index [0] of the array.
input	`uint8_t [int]`	`nonce_len`	Length of the nonce passed into the attestation request.
input	`uint8_t * [bytearray]`	`nonce_x`	Pointer to an array of bytes specifying the nonce to use to seed the attestation calculation and to protect against replay attacks. The least significant byte of the array is in array index [0]. The length of the nonce array should be as expected per the attestation algorithm selected via `attestAlgo`.
input	`uint8_t [int]`	`expMeas_len`	This is the length of the expected measurement value, which is calculated from hashing the firmware and configuration.
input	`uint8_t * [bytearray]`	`expMeas_x`	Pointer to an array of bytes specifying the expected measurement to of firmware and configuration for the purpose of producing the expected MAC value. If the PRoT expected MAC value matches the MAC value produced using this measurement, then the attestation is successful. The least significant byte is in index [0] of the array.

Flow	Data Type	Argument	Definition
output	`int`	`Return Value`	0: This indicates the PMBus attestation was successful and yielded a MAC matching that produced using the expected measurement of firmware and configuration -1: The PMBus attestation failed as the attestation algorithm requested is not implemented by the PMBus target -2: The PMBus attestation was rejected due to trivial nonce, indicating a potentially compromised DRBG. -3: The PMBus attestation algorithm failed due to a communication error with the PMBus target. -4: Return code failed as MAC computed using expected measurement does not match MAC returned by the target.

8.3.2 PMBus_AttestationAlgoSupport

In this command, an integer is supported in which each bit of the integer column represents support for an attestation algorithm. The standard allows for up to 24 attestation algorithms to be supported in the future.

When called, this API updates the value in the 32-bit unsigned integer pointed to by the variable `attSupport`. The value returned is shown in Table 8-8.

The API for this command is as follows:

C variant

```
int PMBus_AttestationAlgoSupport(int       devHandle,
                                 uint8_t   pmbAddr,
                                 int16_t   page,
                                 uint32_t *attSupport)
```

Python variant

```
def PMBus_AttestationAlgoSupport( devHandle, pmbAddr, page,
                                  attSupport)
```

Table 8-8. PMBus_AttestationAlgoSupport Arguments

Flow	Data Type	Argument	Definition
output	uint32_t * [dictionary with key “value” storing int]	attSupport	Attestation support is a pointer to a 32-bit unsigned integer. The pointed to integer is updated by this API. Each bit location within the integer corresponds to an attestation algorithm from [A03] mirrored in Table 11-3. Bit Location — Description [0] (LSB) — If 1, attestation algo set 0 supported [1] — If 1, attestation algo set 1 supported [2] — If 1, attestation algo set 2 supported [3] — If 1, attestation algo set 3 supported [4] — If 1, attestation algo set 4 supported [5] — If 1, attestation algo set 5 supported [6] — If 1, attestation algo set 6 supported [7] — If 1, attestation algo set 7 supported [8] — If 1, attestation algo set 8 supported [9] — If 1, attestation algo set 9 supported [10] — If 1, attestation algo set 10 supported [11] — If 1, attestation algo set 11 supported [20] — If 1, attestation algo set 20 supported [21] — If 1, attestation algo set 21 supported [22] — If 1, attestation algo set 22 supported [23] — If 1, attestation algo set 23 supported Other bits: — Set to 0. Reserved for future attestation algorithms.
	int	Return Value	0: The PMBus target successfully retrieved all the supported attestation algorithms -1: Operation Unsuccessful

8.3.3 PMBus_ReqAttestTarget

In this command, `nonce_x` is a pointer to an array of unsigned 8-bit values with length derived from the attestation algorithm selected. This API returns 0 if the operation is completed successfully. It returns a negative integer if it was unable to successfully send the nonce to the VR. In response to receiving this command, the VR begins the process of computing the MAC in response to the attestation challenge. This computation must complete within **10ms**, at which point the PRoT will retrieve the attestation response from the VR and compare it with the expected MAC response. The parameters and API return value are further described in Table 8-9.

The API for this command is as follows:

C variant

```
int PMBus_ReqAttestTarget(int      devHandle      ,
                          uint8_t  pmbAddr        ,
                          int16_t  page           ,
                          uint8_t  attestAlgo     ,
```

```
                                uint8_t  nonce_len       ,
                                uint8_t *nonce_x         );
```

Python variant

```
    def PMBus_ReqAttestTarget( devHandle, pmbAddr, page,
                               attestAlgo             ,
                               nonce_len, nonce_x     )
```

Table 8-9. PMBus_ReqAttestTarget Arguments

Flow	Data Type	Argument	Definition
input	`uint8_t [int]`	`attestAlgo`	Passes the bit index of the attestation algorithm that the PMBus target is to apply in calculating the expected MAC
input	`uint8_t [int]`	`nonce_len`	Length of the nonce pointed to by `nonce_x`. This field should agree with the length implicitly set by `attestAlgo`.
input	`uint8_t * [bytearray]`	`nonce_x`	Pointer to an array of bytes specifying the nonce to use to seed the attestation calculation and to protect against replay attacks. The least significant byte of the array is in array index [0]. The length of the nonce array is determined by the attestation algorithm selected via `attestAlgo`.
output	`int`	`Return Value`	0: This indicates the PMBus attestation request and nonce were successfully transmitted to the PMBus target -1: The PMBus attestation failed as the attestation algorithm requested is not implemented by the PMBus target -2: The PMBus attestation was rejected due to trivial nonce, indicating a potentially compromised DRBG. -3: The PMBus attestation algorithm failed due to a communication error with the PMBus target.

8.3.4 PMBus_RetrieveAttest

Once the MAC has been calculated the PRoT must gather the result from the VR. PMB Alert upon request completion is optional, the PRoT may rely on the maximum allowed calculation time as specified in 8.3.3 or on return status code. After this maximum calculation time has passed, the PRoT queries the PMBus target with the following API.

C variant

```
     int PMBus_RetrieveAttestTarget(int       devHandle   ,
                                    uint8_t   pmbAddr     ,
                                    int16_t   page        ,
                                    uint16_t *nonce_LSWord,
```

```
                                        uint8_t   mac_len     ,
                                        uint8_t  *mac_x       )
```

Python variant

```
     def PMBus_RetrieveAttestTarget( devHandle, pmbAddr, page,
                                      nonce_LSWord  ,
                                      mac_len, mac_x)
```

The command returns 0 if it was able to successfully retrieve the MAC and the least-significant word of the nonce used to calculate it. Because the return SMBus packet payload would exceed 32-bytes with the sub-command code, a split transaction structure like nonce transmission is used but in opposite direction. It returns a negative integer if it was unable to retrieve the MAC or the corresponding nonce. The parameters for this function are described in Table 8-10.

See [A03] section number as given in Table 11-1 for a standardized hardware implementation.

Table 8-10. PMBus_RetrieveAttestTarget Arguments

Flow	Data Type	Argument	Definition
output	`uint16_t * [dictionary with key "value" storing int]`	`nonce_LSWord`	API loads least-significant 16-bits of the nonce used to generate this MAC array into the value pointed to by `nonce_LSWord`.
output	`uint8_t * [dictionary with key "value" storing int]`	`mac_len`	This is the length of the MAC result pointed to by `mac_x` in bytes. It should agree with the length expected based upon the requested attestation algorithm. If no valid MAC result was calculated by the task, it should return 0.
output	`uint8_t * [bytearray]`	`mac_x`	API loads the MAC value calculated into the array pointed to by this variable. It an array of unsigned 8-bit integers (`uint8_t`) representing 1 byte each. The least significant byte is in array index [0] and others follow subsequently. The length of the array must be pre-allocated per the MAC length expected for the given attestation standard.
output	`int`	`Return Value`	0: API successfully retrieved PMBus target's successfully computed MAC. -1: API request failed as PMBus target failed to compute a MAC. -2: API request failed due to other communication with the PMBus target.

8.4 APIs for Attestation Calculation

Attestation commands, including attestation of target as discussed in section 5.1 and attestation of host as discussed in section 5.2, utilize the same mathematical operations upon different inputs. To avoid duplication of the computational code, three additional APIs are defined. These API are not intended to be exposed at the top-level, but rather to ensure re-use and to minimize code size while implementing the high-level attestation related API. The commands performed in the steps are annotated by the dashed boxes in: (1) Figure 5-5 where the host attests the target device and firmware and (2) Figure 5-7 where target attests hosts and command. These functions are:

- PMBus_HashCalc : Calculates the measurement from command data.
- PMBus_KDFCalc : Calculates the ephemeral key from PSK and nonce.
- PMBus_MACCalc : Calculates the MAC from measurement and ephemeral key.

8.4.1 PMBus_HashCalc

This function takes as an input the data to be hashed, also known as a message and a hashing algorithm selection. It produces as output the message digest, also known as a measurement. Although the message cannot be uniquely reproduced from its measurement, a change in measurement does indicate the underlying message has been altered. This function implements the block labeled with this API name in within both Figure 5-5 and Figure 5-7.

When attesting the target device and firmware at power-on, it is possible that the PRoT may only have the resulting expected measurement from this calculation stored in its NVM, rending a live calculation of this value via the API unnecessary. Target will compute the measurement from the requested command and its firmware and active configuration contents.

When attesting the host rather than the target, the data to be hashed is a concatenation of the incoming command and its arguments arranged per [A03], mirrored in this document as Table 11-7. The API will need to know the data to be hashed (message) and the selected algorithm to be used in determining the measurement (digest).

C variant

```
int PMBus_HashCalc (int       devHandle  ,
                    uint8_t   pmbAddr    ,
                    int16_t   page       ,
                    uint8_t   attestAlgo ,
                    uint32_t  message_len,
                    uint8_t  *message_x  ,
                    uint8_t  *meas_len   ,
                    uint8_t  *meas_x )
```

Python variant

```
def PMBus_HashCalc ( devHandle, pmbAddr, page,
                     attestAlgo             ,
                     message_len, message_x  ,
                     meas_len   , meas_x     )
```

This command does not communicate with the target, so the device handler PMBus address, and page are not likely needed but are maintained in case they play a role in identifying the implementation to use when multiple suppliers or device models are involved. The parameters have values as defined in the following table.

Table 8-11. PMBus_HashCalc Arguments

Flow	Data Type	Argument	Definition
input	`uint8_t [int]`	`attestAlgo`	Attestation algorithm from [A03], mirrored in Table 11-3, whose hashing algorithm selection is applied to the measurement data.
input	`uint32_t * [int]`	`message_len`	This is the length in bytes of the message (incoming data) to be hashed.
input	`uint8_t * [bytearray]`	`message_x`	This is the incoming data (message) to be hashed. The least significant byte is in index [0] of the array.
output	`uint8_t * [dictionary with key "value" storing int].`	`meas_len`	Pointer at which length of the resulting digest or measurement is to be stored by the API call.
output	`uint8_t * [bytearray]`	`meas_x`	Pointer to an array containing the digest or measurement resulting from the hash. The least significant byte is stored in index [0] of the array.
output	`int`	`Return Value`	0: Hashing function calculated successfully -1: Hashing failed, algorithm selected not available -2: Hashing failed, other reason

8.4.2 PMBus_KDFCalc

An API is defined to calculate the ephemeral key from the PSK and nonce. The algorithm to be used is selected via a constant. When termed as per the KMAC documentation [A09], the PSK used is the KIN input key. The nonce is used as the context. The algorithm to be used is selected via an input parameter from the list of attestation algorithms from [A03], mirrored in Table 11-3.

C variant

```
int PMBus_KDFCalc(int       devHandle,
                  uint8_t   pmbAddr,
                  int16_t   page,
                  uint8_t   attestAlgo,
                  uint8_t   psk_len,
                  uint8_t  *psk_x,
                  uint8_t   nonce_len,
                  uint8_t  *nonce_x,
                  uint8_t  *ephKey_len,
                  uint8_t  *ephKey_x   )
```

Python variant

```
def PMBus_KDFCalc( devHandle, pmbAddr, page,
                   attestAlgo        ,
                   psk_len   , psk_x       ,
                   nonce_len , nonce_x     ,
                   ephKey_len, ephKey_x    )
```

This command does not communicate with the target, so the device handler PMBus address and page are not likely needed but are maintained in case they play a role in identifying the implementation to use when multiple suppliers or device models are involved. The parameters have values as defined in Table 8-12.

Table 8-12. PMBus_KDFCalc Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	attestAlgo	Attestation algorithm from [A03], mirrored in Table 11-3, whose hashing algorithm selection is applied to the measurement data.
input	uint8_t [int]	psk_len	This is the length in bytes of the PSK to be used in the ephemeral key calculation.
input	uint8_t * [bytearray]	psk_x	Pointer to an array containing the PSK to be used in the key derivation function (KDF). This is used as the input key for the KDF. The least significant byte is stored in index [0] of the array.
input	uint8_t [int]	nonce_len	This is the length in bytes of the nonce to be used in the ephemeral key calculation.
input	uint8_t * [bytearray]	nonce_x	Pointer to an array containing the nonce to be used in the key derivation function. This is used as the input key for the KDF. The least significant byte is stored in index [0] of the array.
output	uint8_t * [dictionary with key “value” storing int]	ephKey_len	Pointer to the location where the API will store the length of the resulting ephemeral key in bytes.
output	uint8_t * [bytearray]	ephKey_x	Pointer to the location where the API shall store the ephemeral key itself. The least significant byte is stored in index [0] of the array.
output	int	Return Value	0: Ephemeral key derivation succeeded -1: Key derivation failed; algorithm selected not available -2: Key derivation failed, other reason

8.4.3 PMBus_MACCalc

An API is defined to calculate the message authentication code (MAC) from the measurement (hash digest) and the ephemeral key. The algorithm to be used is

selected via an input parameter from the list of attestation algorithms from [A03], mirrored in Table 11-3. The API is defined as:

C variant

```
int PMBus_MACCalc (int       devHandle,
                   uint8_t   pmbAddr,
                   int16_t   page,
                   uint8_t   attestAlgo,
                   uint8_t   ephKey_len,
                   uint8_t  *ephKey_x,
                   uint8_t   meas_len,
                   uint8_t  *meas_x,
                   uint8_t  *mac_len,
                   uint8_t  *mac_x   )
```

Python Variant

```
def PMBus_MACCalc ( devHandle, pmbAddr, page,
                    attestAlgo       ,
                    phKey_len, ephKey_x     ,
                    meas_len , meas_x       ,
                    mac_len  , mac_x        )
```

This command does not communicate with the target, so the device handler PMBus address and page are not likely needed but are maintained in case they play a role in identifying the implementation to use when multiple suppliers or device models are involved. The parameters have values as defined in the following table.

Table 8-13. PDF_MACCalc Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	attestAlgo	Attestation algorithm from [A03], mirrored in Table 11-3, whose hashing algorithm selection is applied to the measurement data.
input	uint8_t [int]	ephKey_len	This is the length in bytes of the ephemeral key to be used in the message authentication code calculation.
input	uint8_t * [bytearray]	ephKey_x	Pointer to an array containing the ephemeral key to be used in the message authentication code (MAC) calculation. The least significant byte is stored in index [0] of the array.
input	uint8_t [int]	meas_len	This is the length in bytes of the measurement (hash digest) to be used in the MAC calculation.
input	uint8_t * [bytearray]	meas_x	Pointer to an array containing the measurement (hash digest) to be used by the message authentication code function. The least significant byte is stored in index [0] of the array.

Flow	Data Type	Argument	Definition
output	uint8_t * [dictionary with key "value" of type int]	mac_len	Pointer to the location where the API will store the length of the resulting message authentication code in bytes.
output	uint8_t * [bytearray]	mac_x	Pointer to the location where the API shall store the message authentication code itself. The least significant byte is stored in index [0] of the array.
output	int	Return Value	0: Message authentication code calculation succeeded. -1: MAC calculation failed; algorithm selected not available. -2: MAC calculation failure, other reason

8.5 Attestation of Host (Security Level 3 with Target Nonce)

Security level 3 APIs leverage the ability to attest the PMBus host sending the action request to the target. This allows execution of some commands by authorized requestors only without leaving the command open to be executed by an unauthorized host on the bus. For the commands in this section, the host is attested using a target generated nonce to prevent replay attack.

8.5.1 PMBus_RequestNonce

The following command requests a nonce from the target device. This is a key part of the attestation of host process, outlined in section 5.2. Generating a nonce in the target helps prevent against replay attacks possible when the PRoT both generates the nonce, and the command data, and computes the expected MAC. This function call will be used either at PRoT level code or called from inside other API for security profile level 3 devices. In most cases this function shall be called form within the other security level 3 API calls, rather than utilized as a standalone call.

C variant

```
int PMBus_RequestNonce(int       devHandle   ,
                       uint8_t   pmbAddr     ,
                       int16_t   page        ,
                       uint32_t  attestAlgo  ,
                       uint8_t  *tgtNonce_len,
                       uint8_t  *tgtNonce_x  )
```

Python variant

```
def PMBus_RequestNonce( devHandle, pmbAddr, page,
                        attestAlgo             ,
                        tgtNonce_len, tgtNonce_x)
```

The command arguments are given in the following table.

Table 8-14. PMBus_RequestNonce Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	attestAlgo	Passes the bit index of the attestation algorithm that the target will use to validate the device sending the PMBus commands. This algorithm also identifies the length and characteristics of the nonce generated in response to this command. It is selected from the list of [A03] mirrored in Table 11-3.
input	uint8_t * [dictionary with key “value” storing int]	tgtNonce_len	This parameter points to an unsigned byte indicating the length in bytes of the target nonce provided to combine with the PSK to form the ephemeral key used in HMAC calculations used to validate the integrity and sender of commands sent to the PMBus Secure Device target.
input	uint8_t * [bytearray]	tgtNonce_x	This parameter points to an array of unsigned bytes that contain the nonce retrieved from the target in response to this request to generate a nonce. If there was a failure during nonce generation, the pointer can be a null value. The least significant byte is in index [0] of the array.
output	int	Return Value	0: Command successfully generated a target nonce -1: Request failed as attestation algorithm not supported by target -2: Request failed for other reason

In the API implementation, summarized in [A03]] section as indicated via Table 11-1, a request is made to a DRBG in the PMBus target that causes it to generate a nonce for the specified attestation algorithm. When the target completes the nonce generation, the API implementation will check to see if the target was able to successfully generate a nonce. If the nonce generation was successful, the API reads It from the target device. The target shall retain the requested nonce and attestation algorithm for use in verifying the subsequent security command request as described in

To reduce the chances of security issue, PMBus Secure Device target and the PRoT should minimize the time any key spends in memory during command execution.

Once a nonce has been successful requested from the target, it is required the next security action API be the only one to which that nonce is applied. If nonce is invalidated in the target if either (1) it is overwritten, (2) after any security action request is received by the target regardless of if the submission results in successful or failed command execution.

After the request nonce API has been called:

- The PRoT will have obtained a nonce from the target.
- The target will know the attestation algorithm to be used for attesting the next security action request command.

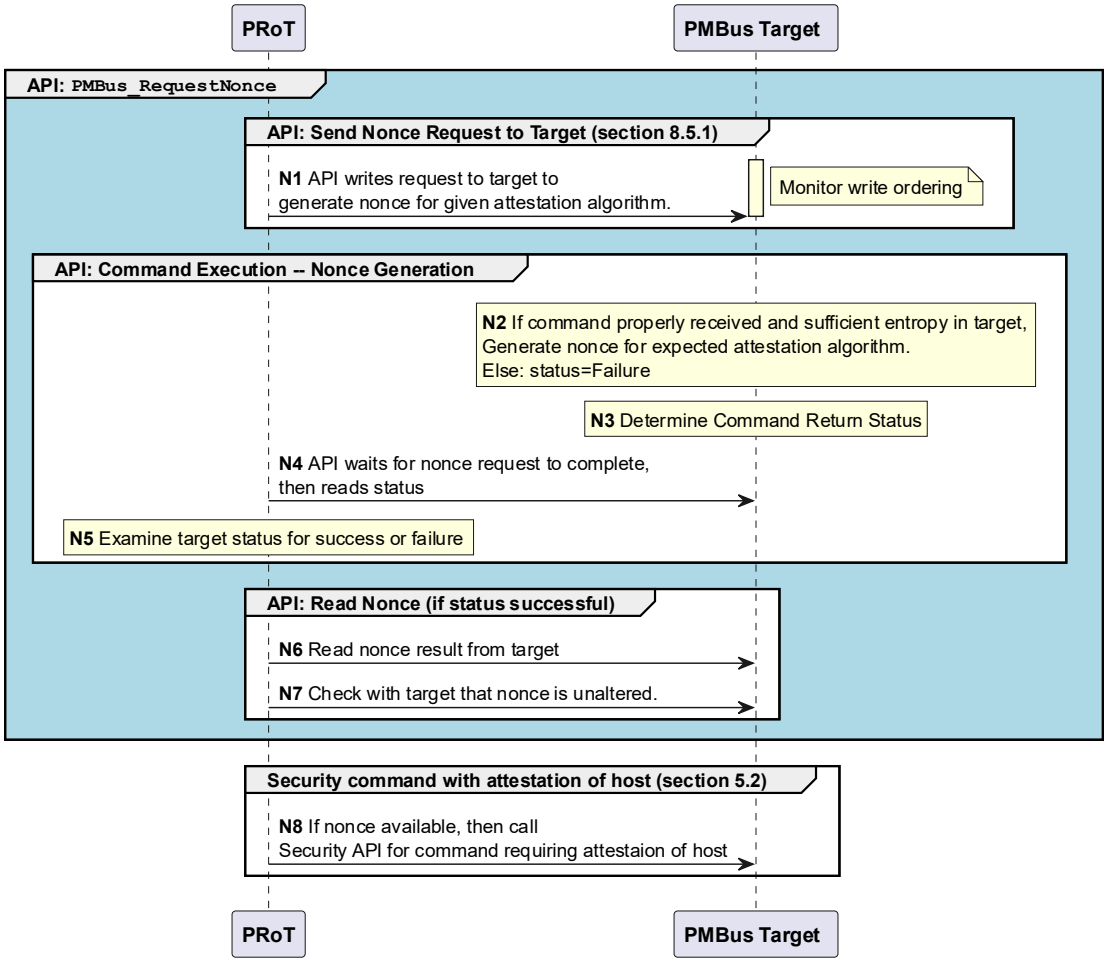

Figure 8-3. PMBus_RequestNonce Flow

8.5.2 PMBus_Secure_Access_Control

PMBus security provides a method to allow attested hosts to issue PMBus ACCESS_CONTROL commands to the target without leaving the ACCESS_CONTROL command open to any device able to drive the bus. This ability is optional prior to security level 3, but mandatory thereafter. In security levels 0-2, it is required to be able to set ACCESS_CONTROL in the most restrictive way permitted by the application. This means disabling write and where applicable reads to commands not anticipated to be used in the application and locking those access permissions with “never again” bit via NVM.

This command ensures request integrity via the on-target attestation of host described in section 5.2 and must use a target generated nonce.

The API is shown next, with parameters listed in Table 8-15.

C variant

```
int PMBus_Secure_Access_Control(int      devHandle       ,
                                uint8_t  pmbAddr         ,
                                int16_t  page            ,
```

```
                                        uint8_t  attestAlgo      ,
                                        uint8_t  psk_len         ,
                                        uint8_t *psk_x           ,
                                        uint8_t  ac_pairs        ,
                                        uint8_t *ac_cmds         ,
                                        uint8_t *ac_settings     )
```

Python variant

```
def PMBus_Secure_Access_Control( devHandle, pmbAddr, page,
                                 attestAlgo, psk_len, psk_x,
                                 ac_pairs, ac_cmds, ac_settings)
```

Table 8-15. PMBus_Secure_Access_Control Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	attestAlgo	This byte indicates the algorithm used for attestation of the host. It is selected from the list of [A03] mirrored in Table 11-3.
input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation algorithm.
input	uint8_t * [bytearray]	psk_x	This is the PSK used when calculating the message authentication code (MAC) used to authenticate the host. This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the action request and its arguments to make the MAC. The least significant byte is in index [0] of the array.
input	uint8_t * [int]	ac_pairs	Number of access control command code and setting pairs being requested via this command.
input	uint8_t * [list of ints]	ac_cmds	Pointer to an array of bytes of length ac_pairs. Each byte in the array contains a command code to be modified by the PMBus ACCESS_CONTROL request.
input	uint8_t * [list of ints]	ac_settings	Pointer to an array of bytes of length ac_pairs. Each byte in the array contains the permission settings to be requested for the command code noted by the corresponding indexed entry of ac_cmds array.

Flow	Data Type	Argument	Definition
output	`int`	`Return Value`	0: The request was completed successfully, and all command code permissions in the list were executed successfully starting at the command code / access permissions pair at index 0 and going up to the pair at (ac_pairs-1). -1: The request failed as the target was unable to successfully attest the host. -2: Errors occurred during the secured access control action request. Either the command itself was unable to run due to its own ACCESS_CONTROL permission setting on permission bits [1] or [0], or the ACCESS_CONTROL bit [5] for the requested command was set to 1 disallowing this security action API from changing the command permissions.

When the data in this API successfully attests the host identity to the target, then the target will act as if there are a series of independent ACCESS_CONTROL commands submitted with host attestation.

The target will iterate through the command code and permission setting pairs starting at index 0 and working its way to index (`ac_pairs`-1). Modifying any access control permissions will be blocked if the permission bit [5] for the ACCESS_CONTROL command itself is set to 1. The setting of an individual command's access control permission would also be blocked by permissions bit [1] or [0], "No more" and "Never Again" respectively. Note that using this command does not require use of the PMBus PASSKEY command as the attestation calculation is stronger than a password transmitted in the clear.

A standard hardware implementation of this command is given in the PMBus Specification Part IV [A03].

8.5.3 PMBus_Secure_PagePlus

PMBus Secure Device defines an API to support the PMBus secure device Page Plus R/W with Attestation of Host command. This command is used to provide a method of attesting the sender of a command along with the integrity of the command data. It is useful when used in cooperation with the PMBus Specification Part II [A02] defined command code "ACCESS_CONTROL". Aside from usual write permission to a given command, PMBus revision 1.5 defines a method that allows writes from via a security command where the bus host issuing the transaction has been attested. This feature is mandatory in security level 3 and optional in security level 0, 1, and 2.

This command ensures request integrity via the on-target attestation of host described in [A03] section as indicated via Table 11-1 and must use a target generated nonce.

The API used for this command is:

C variant

```
int PMBus_Secure_PagePlus(int      devHandle   ,
                          uint8_t  pmbAddr     ,
```

```
                                    int16_t  page        ,
                                    uint8_t  attestAlgo  ,
                                    uint8_t  psk_len     ,
                                    uint8_t *psk_x       ,
                                    uint8_t  wrBlockCount,
                                    uint8_t *wrData_x    ,
                                    uint8_t  rdBlockCount,
                                    uint8_t *rdData_x    )
```

Python variant

```
     def PMBus_Secure_PagePlus( devHandle, pmbAddr, page,
                                attestAlgo, psk_len, psk_x,
                                wrBlockCount, wrData_x,
                                rdBlockCount, rdData_x)
```

Command parameters are given in Table 8-16.

Table 8-16. PMBus_Secure_PagePlus Arguments

Flow	Data Type	Argument	Definition
input	uint8_t	attestAlgo	This byte indicates the algorithm used for attestation of the host. It is selected from the list of [A03] mirrored in Table 11-3.
Input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation of host algorithm.
input	uint8_t * [byte array]	psk_x	This is the PSK used when calculating the message authentication code (MAC) used to authenticate the host. This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the action request and its arguments to make the MAC. The least significant byte is in index [0] of the array.
input	uint8_t [int]	wrBlockCount	This value corresponds to the “Block Count” utilized during the write portion of either a Page Plus Write or Page Plus Read command. Refer to the PMBus PAGE_PLUS_WRITE and PAGE_PLUS_READ documentation to determine the proper value to be loaded into this field for a given transaction type.
input	uint8_t * [bytearray]	wrData_x	This is a pointer to an array of bytes utilized for the write portion of a Page Plus Write or Page Plus Read command. It contains all bytes transmitted after the “Block Count”, so wrBlockSize determines its length. For the encapsulated PMBus PAGE_PLUS command, the PEC is unnecessary since the command integrity is already checked by the attestation mechanism. The least significant byte is in index [0] of the array.

Flow	Data Type	Argument	Definition
input	uint8_t [int]	rdBlockCount	This value corresponds to the “Block Count” utilized during the read portion of a PAGE_PLUS_READ command. If this command is called to send a PAGE_PLUS_WRITE sequence, then this field must be 0. Refer to PAGE_PLUS_READ documentation to determine the proper value to be loaded into this field for a given transaction type. In the PMBus Specification Part II, this is illustrated by the block “Block Count (= M)” field.
Output	uint8_t * [bytearray]	rdData_x	This is a pointer to an array of bytes where the data returned by the read portion of the corresponding PAGE_PLUS_READ command are stored. The length is determined by rdBlockCount. If the command is a PAGE_PLUS_WRITE operation, then this parameter can be passed a null pointer. If the command is a read, this array should already be allocated the proper number of bytes for the read operation. The least significant byte is in index [0] of the array.
input	Int	Return Value	0: The request was completed successfully. The equivalent PAGE_PLUS_WRITE or PAGE_PLUS_READ command was able to be executed successfully. -1: The request failed as the target was unable to successfully attest the host, so the command was not attempted. -2: Errors occurred during the requested command.

A standard hardware implementation compatible with this API is given in the PMBus Specification Part IV [A03].

8.5.4 PMBus_Secure_AlertConfig

The PMBus Specification Part IV [A03] provides a method to define if PMBus security action requests trigger PMBus STATUS_OTHER bit [7] and SMBALERT# upon their completion by the target. This must be configurable by the NVM on the target.

Reconfiguring, specifically to turn off the alert, may hang software that is expecting the alert to arrive at completion of the requested task. Because of this potential complication, commands that change the action of the SMBALERT# at command completion are protected such that attestation of the host is required. Only security level 3 requires support to change this alert configuration dynamically, as protecting the transaction against replay attack requires attestation of the host using target generated nonce. All security configurations must support configuring this alert via NVM.

This command ensures request integrity via the on-target attestation of host described in in Section 5.2 and must use a target generated nonce.

C variant

```
int PMBus_Secure_AlertConfig(int      devHandle   ,
                             uint8_t  pmbAddr     ,
                             int16_t  page        ,
                             uint8_t  attestAlgo  ,
                             uint8_t  psk_len     ,
                             uint8_t *psk_x       ,
                             uint8_t  alertRequest,
                             uint8_t  alertLock   )
```

Python variant

```
def PMBus_Secure_AlertConfig( devHandle, pmbAddr, page,
                              attestAlgo, psk_len, psk_x,
                              alertRequest, alertLock )
```

This determines if the hardware will trigger an SMBALERT# allowing for the host to wait for an interrupt, or if the host must poll status registers on the PMBus target to tell if an action has completed.

Table 8-17. PMBus_Secure_AlertConfig Arguments

Flow	Data Type	Argument	Definition
input	uint8_t	attestAlgo	This byte indicates the algorithm used for attestation of the host. It is selected from the list of [A03] mirrored in Table 11-3.
Input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation of host algorithm.
input	uint8_t * [byte array]	psk_x	This is the PSK used when calculating the message authentication code (MAC) used to authenticate the host. This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the action request and its arguments to make the MAC. The least significant byte is in index [0] of the array.
input	uint8_t [int]	alertRequest	If this value is 1, the target is set up to return SMBALERT# upon the completion of the security action request. If this value is 0, the target is not setup to return SMBALERT# upon the completion of the security request.
input	uint8_t [int]	alertLock	If this value is 1, the setup of the alertRequest bit is locked for the remainder of the power cycle. If 0, the lock is not set. The lock cannot be cleared within a given power cycle once the lock has been set.
output	Int	Return Value	0: The SMB Alert for security action request was completed successfully. -1: The request failed as the target was unable to successfully attest the host, so the command was not attempted. -2: Errors occurred during the requested command.

The standard implementation is defined in the “PMBus secure device” application profile in the section “PMBus Action Completion Alert Generation”.

8.5.5 PMBus_RebootLockout

This function prevents a reboot instruction from being called with a partially updated NVM image. This command ensures that all requested firmware segments have successfully completed authenticated updates before a reboot is allowed. It is particularly useful for scenarios where an aggressor calling a reboot between NVM firmware segments could leave a device in an unusable state. Because the VR needs to attest the host issuing the stay blocking reboot requests until all segments are done, this command is not supported in security levels 0, 1, or 2.

To make sure this reboot-blocking command is applied by an attested host, it is issued with a nonce and an attestation MAC. The data to be passed with this function call is included in Table 8-18.

Note: This function should be redundant if the recommendation at the start of section 8.5 is met, namely that the PMBus target shall be in a recoverable if power is lost during an NVM update. There may be situations where this is not possible, such as when one-time-programmable memory, such as fuses, are completely used up after not being fully burned.

C variant

```
int PMBus_RebootLockout(int       devHandle ,
                        uint8_t   pmbAddr   ,
                        int16_t   page      ,
                        uint8_t   attestAlgo,
                        uint8_t   psk_len   ,
                        uint8_t  *psk_x     ,
                        uint32_t  loWord    ,
                        uint32_t  hiWord    )
```

Python variant

```
def PMBus_RebootLockout( devHandle,  pmbAddr, page ,
                         attestAlgo, psk_len, psk_x,
                         loWord , hiWord
                         )
```

Table 8-18. PMBus_RebootLockout Arguments

Flow	Data Type	Argument	Definition
input	uint8_t	attestAlgo	This byte indicates the algorithm used for attestation of the host. It is selected from the list of [A03] mirrored in Table 11-3.
input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation algorithm used to validate the host

Flow	Data Type	Argument	Definition
input	uint8_t * [byte array]	psk_x	This is the PSK used when calculating the message authentication code (MAC) used to authenticate the host. This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the action request and its arguments to make the MAC. The least significant byte is in index [0] of the array.
input	uint32_t [int]	loWord	For this 32-bit word of bits [31:0], a 1 in each column indicates the corresponding firmware region must be updated before the PMBus Reboot APIs can be called. If one Is not called, then reboot will fail. Bit — Function [0] (LSB) — Firmware segment 0 must be updated before reboot [1] — Firmware segment 1 must be updated before reboot … [31] (MSB) — Firmware segment 31 must be updated before reboot
input	uint32_t [int]	hiWord	For this 32-bit word of bits [31:0], a 1 in each column indicates the corresponding firmware region must be updated before the PMBus Reboot APIs can be called. If one Is not called, then reboot will fail. Bit — Function [0] (LSB) — Firmware segment 32 must be updated before reboot [1] — Firmware segment 33 must be updated before reboot … [31] (MSB) — Firmware segment 63 must be updated before reboot

8.5.6 PMBus_RebootFromOn

This API allows a target supporting attestation of host to execute a reboot cycle in response to an incoming command, even when one of that target's outputs is on. This includes the controller itself executing the power down sequence, a wait, and then a power-on sequence per the [A02] dictated configuration. The arguments are shown in Table 8-19.

Rails will resume their existing ON/OFF state unless they are controlled by a pin. If controlled by a pin, the rail's powering on will be gated by the pin state. Rails obey the TOFF_DELAY, TOFF_FALL, TON_DELAY, TON_RISE sequence per the PMBus Specification Part IV [A03].

Table 8-19. PMBus_RebootFromOn Arguments

Flow	Data Type	Argument	Definition
input	`uint8_t`	`attestAlgo`	This byte indicates the algorithm used for attestation of the host. It is selected from the list of [A03] mirrored in Table 11-3.
input	`uint8_t [int]`	`psk_len`	This is the length in bytes of the PSK used by the attestation algorithm used to validate the host
input	`uint8_t * [byte array]`	`psk_x`	This is the PSK used when calculating the message authentication code (MAC) used to authenticate the host. This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the action request and its arguments to make the MAC.
output	`int`	`Return Value`	Indicates the authentication operation status. Value(s): Meaning 0: Power-on reset request complete. -1: Reboot failed due to flawed attestation of host. -2: Power-on reset failed for other reasons including lockout. Other values: Reserved for future applications.

8.5.7 PMBus_FetchFwLockout

This API permanently locks out the PRoT's ability to fetch firmware, NVM, or configuration memory location to be retrieved by the PMBus firmware fetch buffer command. It must be possible in NVM to set the fetch to be locked out from power-on for all production devices. This fetch-lockout shall be the nominal condition for all production devices. This command must use attestation of the PRoT using a target generated nonce per [A03] section indicated by Table 11-1. The API definition is shown here with parameters as given in Table 8-20.

C variant

```
int PMBus_FetchFwLockout(int      devHandle       ,
                         uint8_t  pmbAddr         ,
                         int16_t  page            ,
                         uint8_t  attestAlgo      ,
                         uint8_t  psk_len         ,
                         uint8_t *psk_x           )
```

Python variant

```
def PMBus_FetchFwLockout( devHandle, pmbAddr, page,
                          attestAlgo, psk_len, psk_x )
```

Table 8-20. PMBus_FetchFwLockout

Flow	Data Type	Argument	Definition
input	uint8_t [int]	attestAlgo	This byte indicates the algorithm used for attestation of the host. It is selected from the list of [A03] mirrored in Table 11-3.
input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation algorithm.
input	uint8_t * [bytearray]	psk_x	This is the PSK used when calculating the message authentication code (MAC) used to authenticate the host. This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the action request and its arguments to make the MAC. The least significant byte is in index [0] of the array.
output	int	Return Value	This value indicates if the operation was successful. 0: Successful -1: Failed to update firmware fetch lockout setting 1-2 to -9: Reserved for vendor specific failure ≤-10: Reserved for future failure codes

8.6 APIs for Firmware Updates

The following functions relate to the application of a firmware update or configuration file updates to a PMBus target device.

Note: It is required that all PMBus targets are designed to not be permanently unusable in the event a firmware update is interrupted. The target should at least be able to get to a "waiting for firmware update retry" state with the firmware authentication criteria (CRC, password, attestation, or signature verification) still intact.

8.6.1 PMBus_NewFwUpdatesRem

The following API is effectively a "fuel gauge" indicating how many firmware updates remain available in NVM. This function yields a space remaining argument between 0 and 7, which is unitless but should convey something about the amount of space remaining. The device manufacturer should specify how many writes are covered by a quantity of "1" on the gauge. At a minimum, each value should indicate that there is enough write space to commit one full NVM segment write to the device. It is permitted to use "1" representing one write and the value 7 to indicate that there are 7 or more segment writes available in the NVM.

C variant

```
int PMBus_NewFwUpdatesRem(int      devHandle,
                          uint8_t  pmbAddr,
                          int16_t  page,
                          uint8_t *spaceRemaining)
```

Python variant

```
int PMBus_NewFwUpdatesRem( devHandle, pmbAddr, page,
                           spaceRemaining          )
```

Table 8-21. PMBus_NewFwUpdates Argument

Flow	Data Type	Argument	Definition
output	`uint8_t * [dictionary with key "value" storing int]`	`spaceRemaining`	Pointer to the location where a uint8_t variable will indicate the gauge value corresponding to the number of NVM segment updates remaining as available to use. The meaning of the value in range 0…7 is as defined in this section, subject to manufacturer definition.
output	`int`	`Return Value`	0: indicates this operation completed successfully, space remaining value is valid -1: Indicates this operation failed, space remaining value is not valid

8.6.2 PMBus_NewFwCommitSL1

Level 0 and level 1 authenticated updates depend upon a checksum and a password. This API provides a method to provide those data as part of a firmware commit request. Note, this password should have a minimum length of 32-bits and is locked-out for the remainder of the PMBus target power cycle after four failed attempts. The parameters and return value associated with this request are shown in Table 8-22.

The return code is listed in prioritized order. If the firmware was committed successfully, the return code is 0. For this to happen successfully: the password must not have been guessed incorrectly more than 4 times, the password must match the expected value, the checksum calculation must match expectation, and there must be sufficient space in the correct memory type to commit the data. If the commit does not happen successfully, failure code is given via the function return value.

C variant

```
int PMBus_NewFwCommitSL1(int       devHandle,
                         uint8_t   pmbAddr,
                         int16_t   page,
                         uint8_t   memLocation,
                         uint32_t  memImage_len,
                         uint8_t  *memImage_x,
                         uint16_t  password_len,
                         uint8_t  *password_x,
                         uint16_t  checksum_len,
                         uint8_t  *checksum_x,
                         uint8_t   checkSegment,
                         uint8_t   commitType )
```

Python variant

```
def PMBus_NewFwCommitSL1( devHandle, pmbAddr, page ,
                          memLocation,
                          memImage_len, memImage_x ,
                          password_len, password_x ,
                          checksum_len, checksum_x ,
                          checkSegment, commitType )
```

Table 8-22. PMBus_NewFwCommitSL1 Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	memLocation	Firmware segment to be overwritten by this request.
input	uint32_t [int]	memImage_len	The length of the data field that is to be written by this request, in bytes.
input	uint8_t * [bytearray]	memImage_x	This is an array of unsigned 8-digit integers (bytes) used as the program memory. The lowest index [0] contains the address 000000h byte of the candidate firmware or configuration data. The next highest index [1] of the array contains byte 000001h of the candidate image, and so on until [memInage_len-1].
input	uint16_t [int]	password_len	This indicates the length in bytes of the array used in password_x.
input	uint8_t * [bytearray]	password_x	This is an array of unsigned 8-digit integers (bytes) used as the password to allow writing the NVM. It has length specified by the next parameter password_len. When both the password and checksum match before the maximum number of passwords retries is reached, the write shall be permitted and committed to memory. The least significant byte of the password is at array element [0], and increasingly significant bytes are at increasing element indexes.
input	uint8_t [int]	checksum_len	Length of the variable checksum_x in bytes.
input	uint8_t * [bytearray]	checksum_x	This is an array of unsigned bytes used as the checksum to allow writing the NVM. It has length specified by the next parameter checksum_len. The manufacturer themselves determines the checksum formula. The least significant byte of the checksum is at array element [0], and increasingly significant bytes are at increasing element indexes.
input	uint8_t [int]	checkSegment	If 0, the checksum calculation is done for the entire memory image If 1, the checksum calculation is done for the new segment only.
input	uint8_t [int]	commitType	If 0, this update is applied to non-volatile memory. If 1, this update is applied to volatile memory. If 2, this update request commits this and all existing volatile updates to non-volatile memory. This is an optional feature. If 3, this update is applied to both volatile and non-volatile memory simultaneously. This is an optional feature.

Flow	Data Type	Argument	Definition
output	int	Return Value	1: The new firmware was committed successfully. A reboot is required to activate it. 0: The new firmware was committed successfully and is active. -1: The request failed after being blocked by an authentication failure, wrong security level, or the security action being unavailable. -2: The request failed as the device was out of free NVM space. -3: The request failed as the candidate firmware version was invalid (typically this prevents application of older firmware). -4 to -9: Reserved for PMBus defined failures -10 to -19: Reserved for manufacturer defined failures.

8.6.3 PMBus_NewFwCommitSL2

Level 2 authenticated updates depend upon using attestation of the candidate image to determine the authenticity of the candidate firmware image. In this case, knowledge of the common PSK by both the PRoT and the PMBus target device is implied. The PMBus target accepts the candidate firmware image if the PMBus target successfully reproduces the expected MAC calculation result using the candidate image, selected attestation algorithm, nonce, and the PSK. Arguments used to transmit this request and gather resulting status are shown in Table 8-23.

C variant

```
int PMBus_NewFwCommitSL2(int       devHandle,
                         uint8_t   pmbAddr,
                         int16_t   page,
                         uint8_t   attestAlgo,
                         uint8_t   psk_len,
                         uint8_t   psk_x,
                         uint8_t   memLocation,
                         uint32_t  memImage_len,
                         uint8_t  *memImage_x,
                         uint8_t   nonce_len,
                         uint8_t  *nonce_x,
                         uint8_t   measurement_len,
                         uint8_t  *measurement_x,
                         uint8_t   checkSegment,
                         uint8_t   commitType )
```

Python variant

```
int PMBus_NewFwCommitSL2( devHandle, pmbAddr, page,
                          memLocation     ,
                          memImage_len    , memImage_x   ,
```

```
                                    nonce_len       , nonce_x      ,
                                    measurement_len , measurement_x,
                                    psk_len         , psk_x        ,
                                    attestAlgo,
                                    checkSegment    ,
                                    commitType )
```

Table 8-23. PMBus_NewFwCommitL2 Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	attestAlgo	This value indicates the attestation algorithm used for the expected MAC calculation. It is selected from the list of [A03] mirrored in Table 11-3.
input	uint8_t [int]	psk_len	This is the length in bytes of the PSK used by the attestation algorithm.
input	uint8_t * [bytearray]	psk_x	This is the PSK used when calculating the message authentication code (MAC) used to authenticate the host. This value, when hashed with the nonce, forms the ephemeral key. The ephemeral key hashes the action request and its arguments to make the MAC.
input	uint8_t [int]	memLocation	Firmware segment to be overwritten by this request.
input	uint32_t [int]	memImage_len	The length of the data field that is to be written by this request, in bytes.
input	uint8_t * [bytearray]	memImage_x	This is an array of unsigned 8-digit integers (bytes) used as the program memory. The lowest index [0] contains the address 000000h byte of the candidate firmware or configuration data. The next highest index [1] of the array contains byte 000001h of the candidate image, and so on until [memInage_len-1].
input	uint8_t [int]	nonce_len	Length of nonce_x in bytes.
input	uint8_t * [bytearray]	nonce_x	This is the random data used to seed the attestation algorithm and protect against replay-based attacks. The least significant byte of the nonce is at array element [0], and increasingly significant bytes are at increasing element indexes. The length of this nonce array is determined from the requested attestation algorithm.
input	uint8_t [int]	measurement_len	If measurement length is zero, then the API is required to provide the measurement used in the MAC calculation. If the measurement length is non-zero, then the measurement_x field contains the measurement used in calculating the MAC.

Flow	Data Type	Argument	Definition
input	uint8_t [bytearray]	measurement_x	This field provides the measurement if the measurement to be used in calculating the MAC is directly passed instead of being calculated within the API. The length of this field is determined by measurement_len. The least significant byte is in index [0] of the array.
input	uint8_t [int]	checkSegment	If 0, the checksum calculation is done for the entire memory image If 1, the checksum calculation is done for the new segment only.
input	uint8_t [int]	commitType	If 0, this update is applied to non-volatile memory. If 1, this update is applied to volatile memory. If 2, this update request commits this and all existing volatile updates to non-volatile memory. This is an optional feature. If 3, this update is applied to both volatile and non-volatile memory simultaneously. This is an optional feature.
output	int	Return Value	1: The new firmware was committed successfully. A reboot is required to activate it. 0: The new firmware was committed successfully and is active. -1: The request failed after being blocked by an authentication failure, wrong security level, or the security action being unavailable. -2: The request failed as the device was out of free NVM space. -3: The request failed as the candidate firmware version was invalid (typically this prevents application of older firmware). -4 to -9: Reserved for PMBus defined failures -10 to -19: Reserved for manufacturer defined failures.

8.6.4 PMBus_NewFwCommitSL3

Level 3 authenticated updates depend upon performing on-die signature verification within the PMBus target device itself. It is implied that firmware itself has a signature of the image embedded within the image. This signature is then validated within the PMBus target itself as per [A03], section number as indicated via Table 11-1. The parameters used to make this function call are shown in Table 8-24.

C variant

```
int PMBus_NewFwCommitSL3(int       devHandle   ,
                         uint8_t   pmbAddr     ,
                         int16_t   page        ,
                         uint8_t   memLocation ,
                         uint32_t  memImage_len,
                         uint8_t  *memImage_x  ,
                         uint8_t   checkSegment,
                         uint8_t   commitType  )
```

Python variant

```
def PMBus_NewFwCommitSL3( devHandle, pmbAddr, page,
                          memLocation,
                          memImage_len, memImage_x,
                          checkSegment, commitType)
```

Table 8-24. PMBus_NewFwCommitSL3 Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	memLocation	Firmware segment to be overwritten by this request.
input	uint32_t [int]	memImage_len	The length of the data field that is to be written by this request, in bytes.
input	uint8_t * [bytearray]	memImage_x	This is an array of unsigned 8-digit integers (bytes) used as the program memory. The lowest index [0] contains the address 000000h byte of the candidate firmware or configuration data. The next highest index [1] of the array contains byte 000001h of the candidate image, and so on until [memInage_len-1]. This image has the signature used to authenticate the firmware
input	uint8_t [int]	checkSegment	If 0, the checksum calculation is done for the entire memory image If 1, the checksum calculation is done for the new segment only.
input	uint8_t [int]	commitType	If 0, this update is applied to non-volatile memory. If 1, this update is applied to volatile memory. If 2, this update request commits this and all existing volatile updates to non-volatile memory. This is an optional feature. If 3, this update is applied to both volatile and non-volatile memory simultaneously. This is an optional feature.

Flow	Data Type	Argument	Definition
output	int	Return Value	1: The new firmware was committed successfully. A reboot is required to activate it. 0: The new firmware was committed successfully and is active. -1: The request failed after being blocked by an authentication failure, wrong security level, or the security action being unavailable. -2: The request failed as the device was out of free NVM space. -3: The request failed as the candidate firmware version was invalid (typically this prevents application of older firmware). -4 to -9: Reserved for PMBus defined failures -10 to -19: Reserved for manufacturer defined failures,

8.6.5 PMBus_Reboot

This command is added to allow power-on-resetting a PMBus target while its output is turned off. The primary purpose of this instruction is to trigger a re-load from the new NVM if required following an authenticated firmware update. Essentially, this is analogous to restarting an OS after an update. The return codes resulting from this API code are shown in Table 8-25.

C variant

```
int PMBus_Reboot(int      devHandle,
                 uint8_t  pmbAddr  ,
                 int16_t  page     )
```

Python variant

```
def PMBus_Reboot( devHandle, pmbAddr, page )
```

Table 8-25. PMBus_Reboot Arguments

Flow	Data Type	Argument	Definition
output	int	Return Value	Indicates the authentication operation status. Value(s): Meaning 0: Power-on reset request complete. -1: Power-on reset failed as the VR was still on. -2: Power-on reset failed for other reasons including lockout. Other values: Reserved for future applications.

8.6.6 PMBus_FetchFw

This API causes the PMBus target to copy the actual contents of the specified memory location into the candidate firmware buffer retrievable via the command PMBus_NewFwDownload described in section 8.6.6. The mechanisms allowing this command must not be possible in a product, and it shall not be possible in the silicon to read the PSK using this command. The arguments for this API are shown in Table 4 36. This command is defined only for test purposes to validate that a design has written the firmware correctly to the buffer by reading it back. In production uses, it must be locked out via an NVM-based initial setting per the lockout mechanism defined in Section 8.5.7. The lockout can be overridden in security level 3 if desired by an attested request.

C variant

```
int PMBus_FetchFw(int      devHandle      ,
                  uint8_t  pmbAddr        ,
                  int16_t  page           ,
                  uint8_t  memLocation    )
```

Python variant

```
def PMBus_FetchFw( devHandle, pmbAddr, page  ,
                   memLocation                )
```

Table 8-26. PMBus_FetchFW Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	memLocation	Firmware segment to be buffered in location readable by PMBus_NewFwDownload command.
output	int	Return Value	This value indicates if the operation was successful. 0: Firmware or configuration successfully buffered within target -1: Firmware or configuration not successfully buffered within the target. Firmware fetch lockout setting blocking command. -2: Firmware or configuration not successfully buffered within the target device, other failures -3 to -4: Reserved -5 to -9: Reserved for vendor specific failure ≤-10: Reserved for future failure codes

8.6.7 PMBus_NewFwDownload

The following API is defined to verify the transfer of a firmware or configuration image from the PRoT to the PMBus target device. This command only reads data pushed by the new firmware upload command of sections 8.6.2, 8.6.3, or 8.6.4 or following a fetch firmware command as described in section 8.6.6. This command is defined only

for test purposes to validate that a design has written the firmware correctly to the buffer by reading it back.

The parameters associated with this command are shown in Table 8-27.

C variant

```
int PMBus_NewFwDownload(int       devHandle    ,
                        uint8_t   pmbAddr      ,
                        int16_t   page         ,
                        uint32_t  memImage_len,
                        uint8_t  *memImage_x  )
```

Python variant

```
def PMBus_NewFwDownload( devHandle, pmbAddr, page,
                         memImage_len, memImage_x  )
```

Table 8-27. PMBus_NewFwDownload Arguments

Flow	Data Type	Argument	Definition
input	uint8_t [int]	memImage_len	The length of the buffer memory to be read by this request, in bytes. The system doing the firmware update must know the length of the firmware segment from which it is reading.
output	uint8_t * [bytearray]	memImage_x	This is an array of unsigned 8-digit integers (bytes) read from the specified memory location. The lowest index [0] contains the address 000000h byte of the retrieved firmware or configuration data. The next highest index [1] of the array contains byte 000001h of the retrieved image, and so on until [memImage_len-1].
output	int	Return Value	This value indicates if the operation was successful. 0: Firmware or configuration successfully read from the target device. -1: Firmware or configuration not successfully read from the target device, firmware fetch lockout controls the blocking command. -2: Firmware or configuration not successfully read from the target device, other failures such as length longer than buffer region. -3 to -4: Reserved -5 to -9: Reserved for vendor specific failure ≤ -10: Reserved for future failure codes

8.7 Miscellaneous APIs

8.7.1 PMBus_Profile_SecurityVersion API

This API needs to be supported by all PMBus Secure Devices to communicate the security profile expectations per the profile supported by the target hardware.

C variant

```
int PMBus_Profile_SecurityVersion (int      devHandle,
                                   uint8_t  pmbAddr,
                                   int16_t  page,
                                   uint8_t *securityLevel )
```

Python variant

```
def PMBus_Profile_SecurityVersion ( devHandle, pmbAddr, page,
                                    securityLevel )
```

Table 8-28. PMBus_Profile_SecurityVersion Arguments

Flow	Data Type	Argument	Definition
output	uint8_t * [dictionary with key “value” containing int]	securityLevel	This is a pointer to the value security level this device is configured to support.
Output	int	Return Value	0: The supported security level was successfully retrieved. -1: The supported security level was not successfully retrieved.

8.7.2 PMBus_Device_FwConfigVersion

This API needs to be supported by all PMBus Secure Devices to communicate the firmware and configuration version stored in the hardware.

C variant

```
int PMBus_Device_FwConfigVersion (int       devHandle,
                                  uint8_t   pmbAddr,
                                  int16_t   page,
                                  uint16_t *fwConfigVer )
```

Python variant

```
def PMBus_Device_FwConfigVersion ( devHandle, pmbAddr, page,
                                   fwConfigVer )
```

The command arguments include the 7-bit PMBus target address and the VR page to identify the device for which the firmware and configuration version tag is being queried.

This function returns the values from the following table.

Table 8-29. PMBus_Device_FwConfigVersion Arguments

Flow	Data Type	Argument	Definition
output	`uint16_t * [dictionary with key "value" containing int]`	`fwConfigVer`	Pointer to the location in which this API stores the firmware and configuration version as a 16-bit unsigned integer.
output	`int`	`Return Value`	0: Function successfully stored the firmware and configuration version into the location pointed to by fwConfigVer. -1: Function failed to successfully store the firmware and configuration version into the location pointed to by fwConfigVer.

8.7.3 PMBus_Device_Profile

This API causes the driver to read feature support whose values should be constant and influence the value of future transaction-level interaction. These values are proprietary to a given vendor's API implementation and may include properties such as:

- If the PMBus standard defined command SECURITY_BYTE (70h) and optional commands SECURITY_BLOCK (71h) and SECURITY_AUTOINC (72h) are available to communicate with the target device.
- For optional commands SECURITY_BLOCK (71h) and SECURITY_AUTOINC (72h), what block minimum and maximum block lengths for both read and write are supported to security command memory.
- What command code operations are supported for PMBus advanced telemetry operations.

The output of this command must be stored in variables with scope internal to that vendor's API. Storing these results internal to the API prevents the driver API from having to pass block length support as a parameter to every single function call, minimizing bus transaction durations.

For PMBus secure devices using the standard implementation of [A03], the method for determining supported block size values is standardized. The only output from this function indicates whether the device profiling was successful, all other returned results are stored internal to the API implementation. If using a standard implementation, calling this command may be necessary to optimally utilize SECURITY_BLOCK and SECURITY_AUTOINC command codes to minimize bus utilization given the capabilities of the specific device.

Due to their high importance to the end-user, the firmware and configuration version ID has a different API, presented in 8.7.2, that exposes that data.

C variant

```
int PMBus_Device_Profile(int     devHandle,
                         uint8_t pmbAddr  ,
                         int16_t page     )
```

Python variant

```
def PMBus_Device_Profile( devHandle, pmbAddr, page )
```

Table 8-30. PMBus_Device_Profile Arguments

Flow	Data Type	Argument	Definition
Global	Vendor	Global variables	Vendor can store directly into higher-level data structure when this function is called with any proprietary data read by this command
output	int	Return Value	0: Function successfully stored the firmware and configuration version into the location pointed to by fwConfigVer. -1: Function failed to successfully store the firmware and configuration version into the location pointed to by fwConfigVer.

9. Access Control Groupings

The PMBus Specification Part II [A02] defines a new ACCESS_CONTROL byte. This byte can be applied to a single command or to a group of commands. The required grouping of commands is shown in this section. Groupings are specified for all commands, if a command is not implemented, then it need not be considered in access control groups. If manufacturer specific direct memory accesses (DMA) commands can affect target output levels or responses, then the direct memory writes or reads shall also be blocked when the ACCESS_CONTROL command denies access.

9.1 PMBus Access Control Byte

The PMBus Specification Part II ACCESS_CONTROL provides limits on how certain commands can be accessed. This includes limiting read or write accesses, limiting how changes in default values can be stored or restored, and locking in those permission changes. In the PMBus Secure Device application profile, a couple of rules govern accesses.

First, any access that commits data to NVM using any method other than the authenticated update shall be blocked. Any open method allowing an external bus agent to commit data to NVM outside of the authenticated update would provide the ability to sidestep authenticated updates.

Second, if output voltage is controlled by AVS or other proprietary high-speed bus, the PMBus target shall have the ability to block writes to PMBus command codes that would induce output voltage changes. This includes commands that directly change the output voltage such as VOUT_COMMAND or VOUT_TRIM but also commands that may implicitly change the output voltage such as VOUT_SCALE_LOOP or OPERATION.

The different access control settings are specified in Table 9-1 taken from the PMBus Specification Part II [A02]. Refer to [A02] for the official language describing each of these control bits. In the event of any conflict between this document and [A02], the [A02] definition shall override the summary in this document. It is expected a PMBus secure device NVM shall hold these access control bits default in NVM for each group.

Table 9-1. PMBus 1.5 ACCESS_CONTROL Command Summary (from [A02])

Bit	Access Item	Function
7	Write Access	When set, the command code or command code group will treat write access attempts as invalid data
6	Read Access	When set, the command code or command code group will treat read access attempts as invalid data
5	Disable Secure Attested Access	When set, the command code or command code group will be blocked from being used via the secure attested access mechanism. If 0, then the host attested access commands can utilize the command.
4	NVM Store	When set at Power On Reset, STORE commands will not update the NVM stored value for the command code or command code group. **Secure Device Application Profile Note**: This is always set to 1 to block STORE commands in PMBus secure device applications.
3	NVM Restore	When set, RESTORE commands do not update the current value of the command code or command code group with the value from the NVM store. **Secure Device Application Profile Note**: This should always be set to 1 to block RESTORE commands in PMBus secure device applications.
2	Write Once	When set at Power On Reset, the command code or command code group will allow write access once, then treat all additional write access attempts as invalid data.
1	No More	When set, the Access Control Byte for this command code or command code group is not altered by a write to ACCESS_CONTROL to any command code within the group, even if PASSKEY is unlocked. When set to “1” in NVM, this bit behaves like “Never Again”. Design can elect to implement only bit [1] No-More with NVM default and bypass Never Again bit [0] as configurable default. Alternately, it would be allowed to get NVM default from Never Again bit [0], so long as both bit [1] and [0] are implemented.
0	Never Again	When set at Power On Reset or RESTORE, the Access Control Byte for this command code or command code group is not altered by a write to ACCESS_CONTROL to any command code within the group, even if PASSKEY is unlocked

9.2 PMBus Access Control Command Groups

There are up to seven potential different groups into which all PMBus access control commands are mapped for PMBus secure device targets. These different categories are separated primarily by what they control, as shown in Table 9-2. If a given PMBus command is not implemented, then access control is not needed for that command. If no command from a given group is implemented, then that group index can be omitted. The

mapping of commands to the access control bits is shown in Table 9-3. A device may separate these groups into finer granularity if desired.

The access control bits themselves should be settable via NVM. When a PMBus Secure Device target is deployed, the access control command itself may be write-protected to forbid any further changes. Each command must have the corresponding permission set as per that application's expected usage. At security level 3, a secured version of the ACCESS_CONTROL is made available that includes the ability to attest PRoT identity and command-request integrity, as described in section 8.5.2.

Where the ACCESS_CONTROL command is enabled during system operation, there is a single "base" command for each group that is expected to be implemented for each page. This command code should be used when modifying the permissions of a given page. This command code may change for some scenarios between VR pages and sensor-only pages. The access-control linked command codes are shown in Table 9-2. It is optional for other commands in the group to be able to control group access permissions.

Table 9-2. Access Control Command Groups

Grp Idx	Group Description	Base Command
0	**Protect Locks:** These commands can gate access to various PMBus commands. In most deployed cases, these commands would have access control set to disable all non-read accesses and set write no more. In some scenarios, it may be beneficial to leave access control bit [5] cleared to allow ACCESS_CONTROL commands via the access control security action with attestation of host as presented in 8.5.2. Access Controls Required to be Configurable: [7]: Write Access [5]: Disable Secure Attested Access (set 0 if host attestation supported per section 5.2) [1]: No More NVM initialization required for all required bits.	VR and Sensor-Only: 0Fh ACCESS_ CONTROL
1	**SMB Alert Mask Lock:** This commands setting may elect to be writeable or not based on system usage. If employed, write protections on this command code would be designed that interrupts are not masked while the system is waiting for a SMB Alert before proceeding to next operation. Access Controls Required to be Configurable: [7]: Write Access [5]: Disable Secure Attested Access (set 0 if host attestation supported per section 5.2) [1]: No More NVM initialization required for all required bits.	VR and Sensor-Only: 1Bh SMBALERT_ MASK

Grp Idx	Group Description	Base Command
2	**Voltage Dynamics:** These commands allow controlling the target voltage output from the PMBus. In Intel server applications, the voltage rails are managed via the SVID bus. In customer applications, this group of access controls should be set to disable writes and to "never again" allow access control changes. For post-Si validation applications, a platform may elect to leave these commands writeable. Access Controls Required to be Configurable: [7]: Write Access [5]: Disable Secure Attested Access (set to 0 if host attestation supported per section 5.2) [1]: No more NVM initialization required for all required bits.	VR: 01h OPERATION Sensor-Only: N/A
3	**PMBus Restore NVM:** This command allows reloading from the NVM using the PMBus RESTORE_* commands. These commands must be disabled for all writes in a PMBus Secure Device with a "Never Again" setting on access control modifications. Access Controls Required to be Configurable: None, fixed to not allowed in PMBus secure device in favor of authenticated updates.	VR & Sensor-Only: 12h RESTORE_DEFAULT_ALL
4	**PMBus Store NVM:** This command allows writes to the PMBus NVM using the PMBus STORE_* commands. These commands must be disabled for all writes in PMBus Secure Device with a "Never Again" setting on access control modifications. Access Controls Required to be Configurable: None, fixed to not allowed in PMBus secure device in favor of authenticated updates.	VR & Sensor-Only: 11h STORE_DEFAULT_ALL
5	**Device Attributes:** The commands in these fields are inherent attributes of the PMBus target and are not subject to change by application. These include commands such as those that indicate the maximum limits of the device as manufactured, its present firmware version, its manufacturer ID, model number, and so on. These are expected not to be writeable in most applications. Access Controls Required to be Configurable: [7]: Write Access [5]: Disable Secure Attested Access (set 0 if host attestation supported per section 5.2) [1]: No more NVM initialization required for all required bits.	VR & Sensor-Only: ADh IC_DEVICE_ID

Grp Idx	Group Description	Base Command
6	**Fault Configuration** The commands in these fields allow changing things such as the OVP/UVP/OCP/OTP protection thresholds and responses. Access Controls Required to be Configurable: [7]: Write Access [5]: Disable Secure Attested Access (set 0 if host attestation supported per section 5.2) [1]: No more NVM initialization required for all required bits.	VR: 46h IOUT_OC_FAULT_LIMIT Sensor-Only: 5Dh IIN_OC_WARN_LIMIT

Table 9-3. PMBus command codes and access groups

Cmd Code	Command Name	Description	Grp Idx	Grouping
00h	PAGE	Page Select.	-	None
01h	OPERATION	Off response, Voltage control method	2	Voltage Dynamics
02h	ON_OFF_CONFIG	Determines On/Off mechanism	2	Voltage Dynamics
03h	CLEAR_FAULTS	Clearing faults	-	None
04h	PHASE	Phase Selection	-	None
05h	PAGE_PLUS_WRITE	Page Plus Write Command	-	None
06h	PAGE_PLUS_READ	Page Plus Read Command	-	None
07h	ZONE_CONFIG	PMB Zone Operation Commands	-	None
08h	ZONE_ACTIVE	PMB Zone Operation Commands	-	None
⋮	⋮	⋮	⋮	⋮
0Eh	PASSKEY	Passkey for altering access control	0	Protect Locks
0Fh	ACCESS_CONTROL	Alters access protection bits	0	Protect Locks
10h	WRITE_PROTECT	Blocks writes to specific commands	0	Protect Locks
11h	STORE_DEFAULT_ALL	Write all op. mem to default NVM	3	PMBus NVM
12h	RESTORE_DEFAULT_ALL	Set all op. mem per default NVM	4	Reload NVM
13h	STORE_DEFAULT_CODE	Write one op mem to default NVM	3	PMBus NVM

Cmd Code	Command Name	Description	Grp Idx	Grouping
14h	RESTORE_DEFAULT_CODE	Set one op mem per default NVM	4	Reload NVM
15h	STORE_USER_ALL	Write all op mem to user NVM	3	PMBus NVM
16h	RESTORE_USER_ALL	Set all op mem per user NVM	4	Reload NVM
17h	STORE_USER_CODE	Write single op mem to user NVM	3	PMBus NVM
18h	RESTORE_USER_CODE	Set single op-mem to user NVM	4	Reload NVM
19h	CAPABILITY	Reports PMBus support options	-	None (Read-only)
1Ah	QUERY	Reports Command Support	-	None
1Bh	SMBALERT_MASK	Blocks Alert from triggering pin	1	Mask Lock
⋮	⋮	⋮	⋮	⋮
20h	VOUT_MODE	Voltage command format	2	Voltage Dynamics
21h	VOUT_COMMAND	Sets output voltage	2	Voltage Dynamics
22h	VOUT_TRIM	Fixed Vout offset, user assembly	2	Voltage Dynamics
23h	VOUT_CAL_OFFSET	Fixed Vout offset, factory calibration	2	Voltage Dynamics
24h	VOUT_MAX	Voltage Limiter, Maximum	5	Device Attributes
25h	VOUT_MARGIN_HIGH	Voltage set point for margining	2	Voltage Dynamics
26h	VOUT_MARGIN_LOW	Voltage set point for margining	2	Voltage Dynamics
27h	VOUT_TRANSITION_RATE	Slew rate for Voltage Changes	2	Voltage Dynamics
28h	VOUT_DROOP	Load-line setting	2	Voltage Dynamics
29h	VOUT_SCALE_LOOP	Output voltage div to sense point	2	Voltage Dynamics
2Ah	VOUT_SCALE_MONITOR	Same as div for scaling OV/ReadVolt	2	Voltage Dynamics
2Bh	VOUT_MIN	Voltage Limiter, Minimum	5	Device Attributes
⋮	⋮	⋮	⋮	⋮

Cmd Code	Command Name	Description	Grp Idx	Grouping
30h	COEFFICIENTS	Coefficients for DIRECT format	-	None
31h	POUT_MAX	Transition point to CP instead of CV	2	Voltage Dynamics
32h	MAX_DUTY	Maximum Duty Cycle	2	Voltage Dynamics
33h	FREQUENCY_SWITCH	Frequency Switch	2	Voltage Dynamics
34h	POWER_MODE	Max Power vs Max Efficiency Switch	2	Voltage Dynamics
35h	VIN_ON	Input voltage where VR turns on	2	Voltage Dynamics
36h	VIN_OFF	Input voltage where VR turns off	2	Voltage Dynamics
37h	INTERLEAVE	Coordinate PMB devices as phases	2	Voltage Dynamics
38h	IOUT_CAL_GAIN	Iout sense gain adjustment	5	Device Attributes
39h	IOUT_CAL_OFFSET	Iout sense offset adjustment	5	Device Attributes
3Ah	FAN_CONFIG_1_2	Fan1/2: Presence, CmdType, TkSetup	-	None
3Bh	FAN_COMMAND_1	Speed control for Fan 1	6	Fault Config
3Ch	FAN_COMMAND_2	Speed control for Fan 2	6	Fault Config
3Dh	FAN_CONFIG_3_4	Fan3/4: Presence, CmdType, TkSetup	-	None
3Eh	FAN_COMMAND_3	Speed control for Fan 3	6	Fault Config
3Fh	FAN_COMMAND_4	Speed control for Fan 4	6	Fault Config
40h	VOUT_OV_FAULT_LIMIT	Vout OV Fault Limit	6	Fault Config
41h	VOUT_OV_FAULT_RESPONSE	Vout OV Fault Behavior	6	Fault Config
42h	VOUT_OV_WARN_LIMIT	Vout OV Warning – Fast Alert	6	Fault Config
43h	VOUT_UV_WARN_LIMIT	Vout UV Warning – Fast Alert	6	Fault Config
44h	VOUT_UV_FAULT_LIMIT	Vout UV Fault Limit	6	Fault Config
45h	VOUT_UV_FAULT_RESPONSE	Vout OV Fault Behavior	6	Fault Config
46h	IOUT_OC_FAULT_LIMIT	Iout OC Fault Limit	6	Fault Config
47h	IOUT_OC_FAULT_RESPONSE	Iout OC Fault Response	6	Fault Config
48h	IOUT_OC_LV_FAULT_LIMIT	Iout OC C2C LV Fault Limit	6	Fault Config
49h	IOUT_OC_LV_FAULT_RESPONSE	Iout OC C2C LV Fault Limit	6	Fault Config

Cmd Code	Command Name	Description	Grp Idx	Grouping
4Ah	IOUT_OC_WARN_LIMIT	Iout OC Warn Limit – Fast Alert	6	Fault Config
4Bh	IOUT_UC_FAULT_LIMIT	Iout UC Limit – Usually Reverse Load	6	Fault Config
4Ch	IOUT_UC_FAULT_RESPONSE	Iout UC Limit response	6	Fault Config
⋮	⋮	⋮	⋮	⋮
4Fh	OT_FAULT_LIMIT	Over-temperature fault limit	6	Fault Config
50h	OT_FAULT_RESPONSE	Over-temperature fault response	6	Fault Config
51h	OT_WARN_LIMIT	Over-temperature warn limit	6	Fault Config
52h	UT_WARN_LIMIT	Under-temperature warning limit	6	Fault Config
53h	UT_FAULT_LIMIT	Under-temperature fault limit	6	Fault Config
54h	UT_FAULT_RESPONSE	Under-temperature fault response	6	Fault Config
55h	VIN_OV_FAULT_LIMIT	Input over-voltage fault level	6	Fault Config
56h	VIN_OV_FAULT_RESPOSNE	Input over-voltage fault response	6	Fault Config
57h	VIN_OV_WARN_LIMIT	Input over-voltage warning limit	6	Fault Config
58h	VIN_UV_WARN_LIMIT	Input under-voltage warning limit	6	Fault Config
59h	VIN_UV_FAULT_LIMIT	Input under-voltage fault limit	6	Fault Config
5Ah	VIN_UV_FAULT_RESPONSE	Input under-voltage fault response	6	Fault Config
5Bh	IIN_OC_FAULT_LIMIT	Input over-current fault limit	6	Fault Config
5Ch	IIN_OC_FAULT_RESPONSE	Input over-current fault response	6	Fault Config
5Dh	IIN_OC_WARN_LIMIT	Input over-current warning limit	6	Fault Config
5Eh	POWER_GOOD_ON	Vout where PowerGood goes up	6	Fault Config
5Fh	POWER_GOOD_OFF	Vout where PowerGood goes down	6	Fault Config
60h	TON_DELAY	Time from start condition until ramp	2	Voltage Dynamics
61h	TON_RISE	Ramp up time from start to regulate	2	Voltage Dynamics
62h	TON_MAX_FAULT_LIMIT	Ramp up timeout fault detect time	6	Fault Config

Cmd Code	Command Name	Description	Grp Idx	Grouping
63h	TON_MAX_FAULT_RESPONSE	Ramp up timeout fault response	6	Fault Config
64h	TOFF_DELAY	Time from off condition to stop Q transfer	2	Voltage Dynamics
65h	TOFF_FALL	Time from TOFF_DELAY until zero V target	2	Voltage Dynamics
66h	TOFF_MAX_WARN_LIMIT	Max Decay to 12.5% of prior target time	6	Fault Config
68h	POUT_OP_FAULT_LIMIT	Output over-power fault level	6	Fault Config
69h	POUT_OP_FAULT_RESPONSE	Output over-power fault response	6	Fault Config
6Ah	POUT_OP_WARN_LIMIT	Output over-power fault warning	6	Fault Config
6Bh	PIN_OP_WARN_LIMIT	Input over-power warning level.	6	Fault Config
⋮	⋮	⋮	⋮	⋮
70h	SECURITY_BYTE	Writes/Reads a byte into security memory	-	None
71h	SECURITY_BLOCK	Writes/Reads a block into security memory	-	None
72h	SECURITY_AUTO_INCREMENT	Writes/Reads a block into security buffers while auto incrementing the addresses	-	None
⋮	⋮	⋮	⋮	⋮
78h	STATUS_BYTE	L1 IRQ reporting byte	-	None. Write-clear
79h	STATUS_WORD	L1 IRQ reporting word	-	None. Write-clear
7Ah	STATUS_VOUT	L2 IRQ for output voltage faults	-	None. Write-clear
7Bh	STATUS_IOUT	L2 IRQ for output current faults	-	None. Write-clear
7Ch	STATUS_INPUT	L2 IRQ for input faults	-	None. Write-clear
7Dh	STATUS_TEMPERATURE	L2 IRQ for temperature faults	-	None. Write-clear
7Eh	STATUS_CML	L2 IRQ for PMBus Comm, mem, logic	-	None. Write-clear
7Fh	STATUS_OTHER	L2 IRQ for fuses and input faults	-	None. Write-clear

Cmd Code	Command Name	Description	Grp Idx	Grouping
80h	STATUS_MFR_SPECIFIC	L2 IRQ for manufacturer specific items	-	None. Write-clear
81h	STATUS_FANS_1_2	L2 IRQ for Fans 1 and 2	-	None. Write-clear
82h	STATUS_FANS_3_4	L2 IRQ for Fans 3 and 4	-	None. Write-clear
83h	READ_KWH_IN	Telem: Read Energy In in [m,1,k]Wh	-	None. Read-only
84h	READ_KWH_OUT	Telem: Read Energy Out [m,1,k]Wh	-	None. Read-only
85h	READ_KWH_CONFIG	Selects [m,1,k]Wh for READ_KWH	6	Fault Config
86h	READ_EIN	Telem: E_In Count/Rollover/Samps	-	None. Read-only
87h	READ_EOUT	Telem: E_Out Count/Rollover/Samps	-	None. Read-only
88h	READ_VIN	Input voltage	-	None. Read-only
89h	READ_IIN	Input current	-	None. Read-only
8Bh	READ_VOUT	Output voltage	-	None. Read-only
8Ch	READ_IOUT	Output current	-	None. Read-only
8Dh	READ_TEMPERATURE_1	Temperature 1	-	None. Read-only
8Eh	READ_TEMPERATURE_2	Temperature 2	-	None. Read-only
8Fh	READ_TEMPERATURE_3	Temperature 3	-	None. Read-only
90h	READ_FAN_SPEED_1	Fan Speed 1	-	None. Read-only
91h	READ_FAN_SPEED_2	Fan Speed 2	-	None. Read-only
92h	READ_FAN_SPEED_3	Fan Speed 3	-	None. Read-only
93h	READ_FAN_SPEED_4	Fan Speed 4	-	None. Read-only
94h	READ_DUTY_CYCLE	Duty Cycle	-	None. Read-only

Cmd Code	Command Name	Description	Grp Idx	Grouping
95h	READ_FREQUENCY	Frequency	-	None. Read-only
96h	READ_POUT	Output Power, Watts	-	None. Read-only
97h	READ_PIN	Input Power, Watts	-	None. Read-only
98h	PMBUS_REVISION	PMBus spec revision implemented	-	None. Read-only
99h	MFR_ID	Manufacturer Identifier	5	Device Attributes
9Ah	MFR_MODEL	Manufacturer Model	5	Device Attributes
9Bh	MFR_REVISION	Manufacturer Revision Number	5	Device Attributes
9Ch	MFR_LOCATION	Manufactured Location	5	Device Attributes
9Dh	MFR_DATE	Manufactured Date	5	Device Attributes
9Eh	MFR_SERIAL	Manufacturer Serial Number	5	Device Attributes
9Fh	APP_PROFILE_SUPPORT	App profile ID and revision number	5	Device Attributes
A0h	MFR_VIN_MIN	Input voltage minimum rating	5	Device Attributes
A1h	MFR_VIN_MAX	Input voltage maximum rating	5	Device Attributes
A2h	MFR_IIN_MAX	Input current maximum rating	5	Device Attributes
A3h	MFR_PIN_MAX	Input power maximum rating	5	Device Attributes
A4h	MFR_VOUT_MIN	Output voltage lowest set point	5	Device Attributes
A5h	MFR_VOUT_MAX	Output voltage highest set point	5	Device Attributes
A6h	MFR_IOUT_MAX	Output current maximum rating	5	Device Attributes
A7h	MFR_POUT_MAX	Output power maximum rating	5	Device Attributes
A8h	MFR_TAMBIENT_MAX	Ambient temperature max rating	5	Device Attributes

Cmd Code	Command Name	Description	Grp Idx	Grouping
A9h	MFR_TAMBIENT_MIN	Ambient temperature min rating	5	Device Attributes
AAh	MFR_EFFICIENCY_LL	Target efficiency point at low-line	5	Device Attributes
ABh	MFR_EFFICIENCY_HL	Target efficiency point at high-line	5	Device Attributes
ACh	MFR_PIN_ACCURACY	Power In Accuracy	5	Device Attributes
ADh	IC_DEVICE_ID	Set at manufacture. IC Type / Part#	5	Device Attributes
AEh	IC_DEVICE_REV	Set at manufacture. IC Revision	5	Device Attributes
⋮	⋮	⋮	⋮	⋮
C0h	MFR_MAX_TEMP_1	Max Temp 1 rating in deg C	5	Device Attributes
C1h	MFR_MAX_TEMP_2	Max Temp 2 rating in deg C	5	Device Attributes
C2h	MFR_MAX_TEMP_3	Max Temp 3 rating in deg C	5	Device Attributes

10. Target Requirements

10.1 Access Control

Access control must be supported by all profile levels of PMBus secure devices.

In all security levels, when supporting a PMBus secure device profile the following behaviors are required:

- The access control bits must support via NVM setting either the “No More” or “Never again” ACCESS_CONTROL bits, bits [1] and [0] of Table 9-1, within each group’s ACCESS_CONTROL byte. The effect of setting either of these bits via NVM is identical, so implementation can set both or either of these two.
- NVM Store and NVM Restore access control bits, shown in bits [4] and [3] of Table 9-1, must be disabled via NVM or by hardwired default for all command groups of Table 9-2.
- The “PMBus Store NVM” and “NVM Restore NVM” command groups of Table 9-2 must have all write accesses and secure access permanently disabled (access disabled and access control changes never again allowed) as they allow direct storage to or restoration from NVM, effectively bypassing security controls. This is accomplished by setting the ACCESS_CONTROL bits defaults to `0011_101x` or `0011_10x1` per Table 9-1 for both command groups.
- The “Voltage Dynamics”, “Fault Config”, “SMB Alert Mask Lock”, and “Device Attributes” groups of Table 9-2 shall support blocking both write access and secure access, with the ability to additionally configure NVM “No More” or “Never Again” to 1 via NVM to maintain those blocks indefinitely for each group.

- Disable secure attested access, bit [5] of Table 9-1, shall be set to 1 in all systems not supporting security profile level 3.
- Write-once support is optional per device. All devices must support behavior equivalent to setting "write-once" bit [2] of Table 9-1 to 0 if not supported.

Additional guidance below is not a requirement to meet PMBus secure device, but a recommendation.

In applications where the output voltage is not controlled by PMBus VOUT_COMMAND but rather by a higher-speed bus such as AVSBus or a proprietary control bus, then it is recommended that "voltage dynamics" group of Table 9-2 have write access cleared and access control locked by asserting "no more" or "never again" bits [1] or [0] of Table 9-1 in NVM.

Where conventional PMBus (not AVS) is the primary means of voltage control, it is possible in security level 3 to have the target attest the hosts' identity prior to executing a critical command, such as VOUT_COMMAND. In such cases, the PMBus host must use the secure access, and the secure access permissions shall be granted using bit [5] of Table 9-1. In this scenario, the open write access controlled by bit [7] of Table 9-1 shall be cleared.

10.2 API Requirements

APIs are required to be provided for each PMBus secure device per Table 8-1.

In cases where a driver / API is tailored specifically to the capabilities of a given device, the API need not always communicate with the target. For example, if the host is running a driver and asks a target-specific driver for supported attestation algorithms, then the function call on the host itself may provide the answer for the API call without communicating to the target.

If a single driver is general or supporting multiple parts, it may inquire with the PMBus target device to determine capabilities by reading hardware registers of the target, with recommendation to support the standard format set forth in PMBus Specification Part IV [A03].

The standard implementation defined in [A03] is strongly recommended to minimize security implementation and programming differences between different PMBus targets. It is not mandatory so long as devices are provided with validated API and meet the requirements set forth in this document.

11. References to other specifications

11.1 Mapping of API to PMBus Standard Hardware Specifications

Table 11-1 maps API descriptions of this document to their recommended standard hardware definition with full details in the PMBus Specification Part IV. This standard hardware is strongly recommended to ease compatibility between suppliers.

Table 11-1. Mapping API to PMBus Part IV Standard Hardware Implementation

API	App Profile Specification Section	PMBus 1.5 Part IV Standard Hardware Specification Section [A03]
PSK Management		

API	App Profile Specification Section	PMBus 1.5 Part IV Standard Hardware Specification Section [A03]
PMBus_ProvisionPSK0	8.2.1	8.1.3
PMBus_ReqNewPSK_Algo	8.2.2	8.2.1
PMBus_ReqNewPSK	8.2.3	8.2.2
PMBus_LockPSK	8.2.4	8.2.3, 8.2.4
Attestation		
PMBus_AttestTarget	8.3.1	8.4.1
PMBus_AttestationAlgoSupport	8.3.2	8.4.1.1
PMBus_ReqAttestTarget	8.3.3	8.4.1
PMBus_RetrieveAttestTarget	8.3.4	8.4.1.4
PMBus_HashCalc	8.4.1	8.4.2
PMBus_KDFCalc	8.4.2	8.4.3
PMBus_MACCalc	8.4.3	8.4.4
Firmware Update Commands		
PMBus_NewFwUpdatesRem	8.6.1	12.3.1
PMBus_NewFwCommitSL1	8.6.2	10
PMBus_NewFwCommitSL2	8.6.3	10
PMBus_NewFwCommitSL3	8.6.4	10
PMBus_Reboot	8.6.5	10.4.2
PMBus_NewFwDownload	8.6.6	10.5.1
PMBus_FetchFw	8.6.6	10.5.1
Security Level 3 Commands		
PMBus_RequestNonce	8.5.1	9.2
PMBus_Secure_Access_Control	8.5.2	11.1
PMBus_Secure_PagePlus	8.5.3	12.4
PMBus_Secure_AlertConfig	8.5.4	12.2
PMBus_RebootLockout	8.5.5	10.4.1
PMBus_RebootFromOn	8.5.6	10.4.3
PMBus_FetchFwLockout	8.5.7	10.5.2
Miscellaneous Commands		
PMBus_Profile_SecurityVersion	8.7.1	12.1.1
PMBus_Device_FwConfigVersion	8.7.2	12.1.2
PMBus_Device_Profile	8.7.3	N/A

11.2 Frequently Referenced Tables and Figures from Other Specifications

This section provides copies of key figures and tables from other specifications referred to in this specification. In the event of any mismatch between the tables copied here and

the source document from which they were copied, the source data will override the definition copied into this section.

11.2.1 PSK Iteration Algorithms (from [A03] section 8.2)

This table is taken from PMBus Specification Part IV [A03], Table 8-3.

Table 11-2. Permitted PSK Iteration Algorithms (from [A03] Table 8-3).

PSK Iteration Algorithm	Resultant PSK_N	KDF Operation
0	PSK_N (256-bit)	**KDF in counter mode per [A09] section 4.1 with: Keyed-Hashed Message Authentication Code SHA-256 per [A03]** With: PRF = HMAC with *h* = SHA-256 (SHA2) from [A03] L = length(PSK_N) = 256-bits h = 256-bits (SHA-256) $n = \lceil L/h \rceil = 1$ $K_{IN} = PSK_{N-1}$ $Label$ = ASCII of “PSK” $Context$ = Seed from Buffer Region [2] $[L]_2$ = 0x0100 $X = 0x0001 \| Label \| 0x00 \| Context \| [L]_2$ $PSK_N = K_{OUT} = PRF(K_{IN}, X)$ $= HMAC_{SHA-256}(K = PSK_{N-1}, text = X)$
1	PSK_N (128-bit)	**KDF Using KMAC mode per [A09] section 4.4 with:n$\mathbf{KMAC128}(K, X, L, S)$ based upon SHA3 from [A08].** With: $K = K_{IN} = PSK_{N-1}$ (128-bits) $X = Context$ = Seed from Buffer Region [2]b L = length(PSK_N) = 128 bits $S = Label$ = ASCII of “PSK” $PSK_N = K_{OUT} = \mathrm{KMAC128}(K, X, L, S)$
2	PSK_N (256-bit)	**KDF Using KMAC mode per [A09]section 4.4 with:** **$\mathbf{KMAC128}(K, X, L, S)$ based upon SHA3 from [A08].** With: $K = K_{IN} = PSK_{N-1}$ (256-bits) $X = Context$ = Seed from Buffer Region [2] L = length(PSK_N) = 256 bits $S = Label$ = ASCII of “PSK” $PSK_N = K_{OUT} = \mathrm{KMAC128}(K, X, L, S)$

PSK Iteration Algorithm	Resultant PSK_N	KDF Operation
3	PSK_N (256-bit)	**KDF Using KMAC mode per [A09] section 4.4 with:** **$\mathbf{KMAC256}(K, X, L, S)$ based upon SHA3 from [A08].** With: $K = K_{IN} =$ PSK_{N-1} (256-bits) $X = Context =$ Seed from Buffer Region [2] $L =$ length(PSK_N) = 256 bits $S = Label$ = ASCII of "PSK" $PSK_N = K_{OUT} = \mathrm{KMAC256}(K, X, L, S)$

11.2.2 Attestation Algorithm Options from [A03] section 8.3

The following three tables are copied from section 8.3 of [A03] at the time of this document's publishing. Please validate these tables still accurately mirror [A03] at the time of use. In the event of any conflict, the equivalent tables from the latest version of [A03] override the values in this table. These tables are copied here for convenience only.

Table 11-3. Attestation Algorithm Options from [A03]

Attestation Algorithm Set	Hashing Algorithm Used to Develop Measurement of Firmware & Configuration	Keyed Hash Function Used to Determine Ephemeral Key and Resulting Hash Message Authentication Code
0	Hash A from Table 11-4	Keyed Hash A from Table 11-5 and Table 11-6
1	Hash A from Table 11-4	Keyed Hash B from Table 11-5 and Table 11-6
2	Hash A from Table 11-4	Keyed Hash C from Table 11-5 and Table 11-6
3	Hash A from Table 11-4	Keyed Hash D from Table 11-5 and Table 11-6
4	Hash B from Table 11-4	Keyed Hash A from Table 11-5 and Table 11-6
5	Hash B from Table 11-4	Keyed Hash B from Table 11-5 and Table 11-6
6	Hash B from Table 11-4	Keyed Hash C from Table 11-5 and Table 11-6
7	Hash B from Table 11-4	Keyed Hash D from Table 11-5 and Table 11-6
8	Hash C from Table 8-10	Keyed Hash A from Table 11-5 and Table 11-6
9	Hash C from Table 8-10	Keyed Hash B from Table 11-5 and Table 11-6
10	Hash C from Table 8-10	Keyed Hash C from Table 11-5 and Table 11-6
11	Hash C from Table 8-10	Keyed Hash D from Table 11-5 and Table 11-6
12-19	Reserved	Reserved
20	Manufacturer Defined Algo 0	Manufacturer Defined Algo 0
21	Manufacturer Defined Algo 1	Manufacturer Defined Algo 1
22	Manufacturer Defined Algo 2	Manufacturer Defined Algo 2
23	Manufacturer Defined Algo 3	Manufacturer Defined Algo 3

Table 11-4. Attestation Measurement Hashing Algorithm from [A03]

Hashing Algorithm	Message Digest	Resultant Measurement Length	Operation	Data Input Message (M)
Hash A	meas	384-bits	**SHA-384** from SHA2 [A03], Block Size = 1024	*For device attestation command:* see [A03] Table 8-15. *For PSK iteration command*: see [A03] Table 8-2 *For PSK lock forever command:* see [A03] Table 8-6.
Hash B not recommended beyond 2030	meas	256-bits	**SHA3-256** from [A06]	
Hash C	meas	384-bits	**SHA3-384** from	

Table 11-5. Ephemeral Key Determination Options (KDF) from [A03]

Keyed Hash Algorithm Selection	Ephemeral Key (K_{OUT})	KDF Operation
Keyed Hash A	mk (256-bit)	**KDF in counter mode per [A09] section 4.1 with: Keyed-Hashed Message Authentication Code SHA-256 per [A03]** With: PRF = HMAC with *h* = SHA-256 (SHA2) from [A03] L = length(PSK_N) = 256-bits h = 256-bits (SHA-256) $n = \lceil L/h \rceil$ = 1 $K_{IN} = PSK_N$ $Label$ = ASCII of "VR security protocol" $Context$ = Nonce (256-bits) $[L]_2$ = 0x0100 $X = 0x0001 \| Label \| 0x00 \| Context \| [L]_2$ $mk = K_{OUT} = PRF(K_{IN}, X)$ $= HMAC_{SHA-256}(K = PSK_N, text = X)$
Keyed Hash B Not recommended beyond 2030	mk (128-bit)	**KDF Using KMAC mode per [A09] section 4.4 with:** $\mathbf{KMAC128}(K, X, L, S)$ **based upon SHA3 from [A08].** With: $K = K_{IN} = PSK_N$ (128-bits) $X = Context$ = Nonce (256-bits) L = length(PSK_N) = 128 bits $S = Label$ = ASCII of "VR security protocol" $mk = K_{OUT} = \mathrm{KMAC128}(K, X, L, S)$

Keyed Hash Algorithm Selection	Ephemeral Key (K_{OUT})	KDF Operation
Keyed Hash C	mk (256-bit)	**KDF Using KMAC mode per [A09] section 4.4 with: $\mathbf{KMAC128}(\boldsymbol{K}, \boldsymbol{X}, \boldsymbol{L}, \boldsymbol{S})$ based upon SHA3 from [A08].** With: $K = K_{IN} =$ PSK_N (256-bits) $X = Context =$ Nonce (256-bits) $L =$ length(PSK_N) = 256 bits $S = Label$ = ASCII of "VR security protocol" $mk = K_{OUT} = \mathrm{KMAC128}(K, X, L, S)$
Keyed Hash D	mk (256-bit)	**KDF Using KMAC mode per [A09] section 4.4 with: $\mathbf{KMAC256}(\boldsymbol{K}, \boldsymbol{X}, \boldsymbol{L}, \boldsymbol{S})$ based upon SHA3 from [A08].** With: $K = K_{IN} =$ PSK_N (256-bits) $X = Context$ = Nonce (256-bits) $L =$ length(PSK_N) = 256 bits $S = Label$ = ASCII of "VR security protocol" $mk = K_{OUT} = \mathrm{KMAC256}(K, X, L, S)$

Table 11-6. Hashed Message Authentication Code Selection (MAC) from [A03]

Keyed Hash Algorithm Selection	Resultant HMAC (K_{OUT})	Keyed Hash Message Authentication Code
Keyed Hash A	mac (256-bit)	**mac = $HMAC_{SHA\text{-}256}$(H,K,X) from [A05] using SHA2 [A03]** H = SHA-256 (SHA2) [A03] K = mk (256-bit) of Table 11-5 X = Measurement of Table 11-4 with length ≥ length(mk)
Keyed Hash B Not recommended beyond 2030	mac (128-bit)	**mac = KMAC128(K,X,L,S) from [A09] using SHA3 [A08]** K = mk (≥128-bit) of Table 11-5 X = Measurement of Table 11-4 with length ≥ length(mk) L = 128-bits S = ""
Keyed Hash C	mac (256-bit)	**mac = KMAC128(K,X,L,S) from [A09] using SHA3 [A08]** K = mk (≥256-bit) of Table 11-5 X = Measurement of Table 11-4 with length ≥ length(mk) L = 256-bits S = ""

Keyed Hash Algorithm Selection	Resultant HMAC (K_{OUT})	Keyed Hash Message Authentication Code
Keyed Hash D	mac (256-bit)	**mac = KMAC256(K,X,L,S) from [A09] using SHA3 [A08]** K = mk (≥256-bit) of Table 11-5 X = Measurement of Table 11-4 with length ≥ length(mk) L = 256-bits S = “”

11.2.3 PMBus Host Attestation Formulas from [A03] “PMBus Host Attestation”

The following table is a mirror of Table 9-1 from [A03] placed here for convenience. If any conflict exists between this table and Table 9-1 from [A03], then the table from [A03] shall overrule this document which is frozen at one-point in time.

Table 11-7. Data considered for PRoT attestation measurment (from [A03] Table 9-1)

Opcode	Function	Section	Message Data used for Hash HMAC calculation
04h	PSK Iteration	8.2.3	{ (PMBus target 7-bit address << 1) || security action details[0] || security action details [1] || security action request || Seed from buffer region [2]} where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.
06h	PSK Lock with Target Generated Nonce	8.2.4	{ (PMBus target 7-bit address << 1) || security action details [0] || security action details[1] || security action request } where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.
09h	Firmware Fetch Lockout	8.5.7	{ (PMBus target 7-bit address << 1) || security action details [0] || security action details[1] || security action request } where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.
0Ah	Power-On Reset Lockout	8.5.5	{ (PMBus target 7-bit address << 1) || security action details [0] || security action details[1]

Opcode	Function	Section	Message Data used for Hash HMAC calculation
			|| security action request } where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.
0Bh	PMBus Status alert triggering	8.5.4	{ (PMBus target 7-bit address << 1) || security action details [0] || security action details[1] || security action request } where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.
0Ch	Anytime Power-On Reset	8.5.6	{ (PMBus target 7-bit address << 1) || security action details [0] || security action details[1] || security action request } where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.
0Dh	PMBus Page Plus Write and Page Plus Read with Host Attestation	8.5.3	For write portion: { (PMBus target 7-bit address << 1) || security action details [0] || security action details[1] || security action request } where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value. Note: for read portion protecting returned data: { security action details [0] || security action details[1] || security action request || buffer region [2]} where || is concatenation and << is left shift.
0Eh	Secured Access Control Action	8.5.2	{ (PMBus target 7-bit address << 1) || security action details [0] || security action details[1] || security action request || buffer region [2] contents used to transmit command & access pairs }

Opcode	Function	Section	Message Data used for Hash HMAC calculation
			where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.
10h	PSK lock forever	8.2.4	{ (PMBus target 7-bit address << 1) || security action details [0] || security action details[1] || security action request } where || is concatenation and << is left shift. Left shift of 7-bit value produces 8-bit value.

Appendix I. Summary Of Changes

DISCLAIMER: The section is provided for reference only and for the convenience of the reader. No suggestion, statement or guarantee is made that the description of the changes listed below is sufficient to design a device compliant with this document.

A summary of the changes made from Revision 1.0 to this revision, 1.1, is given below. This is not an exact list of every change made between the two documents; rather, it is a summary of the changes deemed significant by the editor.

- Section 8.6.6 and Table 8-26 updated to remove the references to the PSK as part of the PMBus_FetchFw command.