

**Authentication with ATSHA204A Chip (Phase II)**

Abstract

The authentication algorithm of phase I is extended in such a way to include the digest of the microcontroller current program.

Table of Contents

[1- Summary 1](#_Toc496954263)

[1-1- Project summary 2](#_Toc496954264)

[2- Theory 1](#_Toc496954265)

[2-1- Introduction 2](#_Toc496954266)

[2-2- Terminology 2](#_Toc496954267)

[2-3- Authentication and Integrity 3](#_Toc496954268)

[2-4- Secure Hash Algorithm 256 (SHA256) 3](#_Toc496954269)

[2-4-1- Authentication without encryption 5](#_Toc496954270)

[2-4-2- Personalization using ATSHA unique serial number 9](#_Toc496954271)

[2-5- Key generation and safe storage 10](#_Toc496954272)

[2-5-1- Generating Diversified Key for Boot Secret 11](#_Toc496954273)

[2-5-2- Parent Code Encryption 11](#_Toc496954274)

[2-5-3- Storing secrets in MCU 12](#_Toc496954275)

[2-5-4- Choosing the parent key 12](#_Toc496954276)

[2-6- The proposed authentication algorithm (Basic scheme) 12](#_Toc496954277)

[2-6-1- Verifying ACK/NACK After Authentication 14](#_Toc496954278)

[2-7- Bootloader 14](#_Toc496954279)

[3- ATSHA204A CHIP 1](#_Toc496954280)

[3-1- Introduction 2](#_Toc496954281)

[3-2- ATSHA Data Storage Units and Buffers 2](#_Toc496954282)

[3-2-1- Configuration Zone (CZ) 3](#_Toc496954283)

[3-3- DZ Configuration 6](#_Toc496954284)

[3-3-1- TempKey (ATSHA SRAM) 7](#_Toc496954285)

[3-4- ATSHA Commands 8](#_Toc496954286)

[3-4-1- Implementation of SHA256 8](#_Toc496954287)

[3-4-1- MAC Command 10](#_Toc496954288)

[3-4-2- Nonce command 15](#_Toc496954289)

[3-4-3- DeriveKey command 19](#_Toc496954290)

[3-4-4- CheckMac command 24](#_Toc496954291)

[4- Algorithm and Abstract Implementation 1](#_Toc496954292)

[4-1- Introduction 2](#_Toc496954293)

[4-2- CZ and DZ configuration 2](#_Toc496954294)

[4-3- Authentication protocol state diagram 3](#_Toc496954295)

[4-3-1- Implementation approach 4](#_Toc496954296)

[4-3-2- Preparation State (CPD computation) 8](#_Toc496954297)

[4-3-3- PL1 Initialization State 8](#_Toc496954298)

[4-3-4- Create Child Key, RCPD and ROPD States 8](#_Toc496954299)

[4-3-5- Compute Signature States 11](#_Toc496954300)

[4-3-6- Authentication States 12](#_Toc496954301)

[4-3-7- Check ACK Authenticity States 13](#_Toc496954302)

[4-3-8- PL1 Finalization State 15](#_Toc496954303)

[4-3-9- Other States of the Protocol 15](#_Toc496954304)

[4-4- Impact of Unwanted MCU Reset 17](#_Toc496954305)

[5- Software Implementation 1](#_Toc496954306)

[5-1- Introduction 2](#_Toc496954307)

[Appendix I: Data Storage in AVR CPUs 1](#_Toc496954308)

[I-1 Introduction 2](#_Toc496954309)

[Appendix II: ATSHA Programmer 1](#_Toc496954310)

[II-1 Introduction 2](#_Toc496954311)

[II-2 Implementation of ATSHA Programmer 2](#_Toc496954312)



# Summary

## Project summary



# Theory

## Introduction

The most important aspect of employing the ATSHA chip is to establish the basic authentication algorithm. In this chapter, we aim to address the following topics:

1. Define a **secure** authentication algorithm that can be implemented with ATSHA.
2. Describe how this algorithm is implemented using ATSHA (what commands, data slots and etc. are needed to do so). Detailed discussion would be given in chapter three.
3. How to store secrets in a microcontroller without divulging them to an adversary.
4. How to compute the Hash of a program in a microprocessor in the fastest way possible.

In what follows, we discuss each one of the above topics. First, however, we need to establish some definitions.

## Terminology

The following terms and definitions are used throughout this document:

* **Encryption vs. authentication**: Suppose we want to send a message *M*. Encryption is encoding and decoding of a message for confidentiality. It is always reversible. Encryption does not necessarily authenticate. Authentication is used to check the **identity** of the sender and the **integrity** of the message. The message is not necessarily encrypted. Hence by authentication we seek to ensure that when a sender sends a message, an adversary or a middle man has not done anything to change the content of the message.
* **Host**: In this project, a host is an entity that needs to be authenticated and starts the authentication process. In our application, host is a MicroController unit (MCU).
* **Client**: In this project, a client is an entity by which a host is authenticated. In our application, client is an ATSHA204A.
* **Challenge**: Challenge is a question that one side of an (authentication) algorithm asks and the other side must answer. A simple example is asking for a password. As another example, assume that a host and a client both possess a secret. The host may send a random number to the client as challenge. The response from the client is that it performs an XOR of random number with the secret and sends the response back to host. Now, the host checks the response to its challenge (by performing a separate XOR) and verifies whether the response to challenge is correct or not.
* **Digest**: A **digest** is simply a **hash** (short equivalent) of a message. It's the output of a cryptographic **hash** function applied to input data. The input data is typically referred to as a message. SHA256 is the hash algorithm used in ATSHA204.
* **Symmetric authentication:** With Symmetric Authentication using secrets stored on the host using Challenge-Response, like you might expect from the name, the host sends a challenge and the client makes a response using cryptographic tools and then the host runs the same process in order to make a comparison to see if it gets the same answer. It the answers on each side match then the client is real.
* **Asymmetric authentication:** Asymmetric algorithms are known as public key algorithms. Using asymmetric algorithms, a public key is given to all users and each entity would have its own private key. The advantage over symmetric algorithms is that there is no need to store a key on the host side.
* **Nonce:** a **nonce** is an arbitrary number that may only be used once. They are often random numbers or pseudo random numbers.
* **(Digital) Signature of message**: A **digital signature** is a mathematical scheme for demonstrating the authenticity of digital messages or documents. In symmetric authentication methods, a signature could be defined as the hash of a message plus the secret key. Digital signatures assure that the message received is original and is not meddled with by a third party after transmission.

## Authentication and Integrity

The following is based on [STALLING]. Encryption protects against passive attack (eavesdropping). A different requirement is to protect against active attack (falsification of data and transactions). Protection against such attacks is known as message authentication.

A message, file, document, or other collection of data is said to be authentic when it is genuine and comes from its alleged source. Message authentication is a procedure that allows communicating parties to verify that received messages are authentic. The two important aspects are to verify that the contents of the message have not been altered and that the source is authentic. We may also wish to verify a message’s timeliness (it has not been artificially delayed and replayed) and sequence relative to other messages flowing between two parties. All of these concerns come under the category of data integrity.

Note that encryption might be sufficient for authentication in some cases, especially only if only the sender and receiver know the authentication algorithm, but this is not sufficient in all cases. For example, in this project we only need authentication, and we do not encrypt the software used in microcontroller. This authentication without encryption is the topic of next section.

## Secure Hash Algorithm 256 (SHA256)

SHA256 is the building block of all authentication algorithms we implement in this project. Assume we denote SHA256 by and , where is the input and is the output. The main characteristics of are:

* It can be applied to a block of data of any size.
* It produces a fixed-length output.
* It is computationally infeasible to find such that . This means hash is one-way.
* For any given block x, it is computationally infeasible to find with .
* It is computationally infeasible to find any pair such that .

Note that the above also applies to other secure hash functions of category SHA-2 and SHA-3. Fig. 2 shows the standard implementation of SHA256 as given by international standards and used in ATSHA [The FIST standard].



Fig. Standard implementation of SHA256

In order to compute SHA256 of an input message, first we must pad the input message in the following way so that it becomes a multiple of 512 bits. Note that padding is necessary even if the message is a multiple of 512 bits. Assume that the message length in bits is denoted by :

**Step 1:** Append one bit of 1to the end of message

**Step 2:** Solve this equation for : . Then append an additional bits of zero to the end of message.

**Step 3:** Now append or size of the original message in unsigned integer format to end of the previous message in a 64 bit format.

Given the size of , the maximum message length for SHA256 is bits. The above padding results in a message length that is a multiple of 512 bits.

Now to compute the hash and as depicted in Fig. , the message is broken into blocks of 512 bit size. Each block is fed to a function F that manipulates data and generates an output of size 256 bits. This output block is then added to , which is the output of the previous hashing step to form the current hash of the message. After repetitions of the above step, would be the output hash. Note that is an initial value that has a standard form (see [FIST standard][STALLING]).

The important point about the above procedure is that SHA256 is calculated iteratively. Hence, to compute the hash of a very large message, we do as follows:

**Step 1:** Determine the length of the message

**Step 2:** Add padding bytes to the end of the message.

**Step 3:** Use a SHA256 implementation that:

* + For the first block of message uses the standard initial value ( )
  + For the remaining message blocks uses the last hashing value.

Keeping the above procedure in mind, both software and hardware implementations of the procedure are discussed in section ??? .

### Authentication without encryption

In all authentication without encryption approaches, an authentication tag is generated and appended to each message for transmission. The message itself is not encrypted and can be read at the destination independent of the authentication function at the destination.

One authentication technique involves the use of a secret key to generate a small block of data, known as a Message Authentication Code (MAC) that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key . When A has a message to send to B, it calculates the message authentication code as a function of the message and the key: . The message plus code are transmitted to the intended recipient. The recipient performs the same calculation on the received message, using the same secret key, to generate a new message authentication code. The received code is compared to the calculated code (Figure ?????). If we assume that only the receiver and the sender know the identity of the secret key, and if the received code matches the calculated code, then the following statements apply:

1. The receiver is assured that the message has not been altered. If an attacker alters the message but does not alter the code, then the receiver’s calculation of the code will differ from the received code. Because the attacker is assumed not to know the secret key, the attacker cannot alter the code to correspond to the alterations in the message.
2. The receiver is assured that the message is from the alleged sender. Because no one else knows the secret key, no one else could prepare a message with a proper code.

The most important conclusion here is that: **The success of MAC approach relies on the secrecy of the key on both sides. If the key is known, generating fake MACs would be easy.**

Hash functions are popular and have been standardized for generation of MAC, hence the term HMAC (Hash based MAC). Unlike the MAC, a hash function does not take a secret key as input. To authenticate a message, the message digest is sent with the message in such a way that the message digest is authentic.

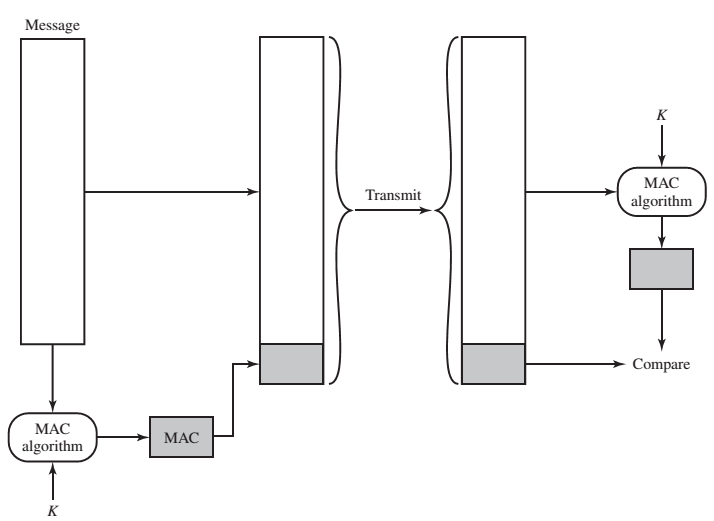


Fig . Authentication using MAC.

Figure ???? shows a technique that uses a hash function but no encryption for message authentication. This technique assumes that two communicating parties, say A and B, share a common secret value .When A has a message to send to B, it calculates the hash function over the concatenation of the secret value and the message: . It then sends to B. Because B possesses , it can recompute and verify . Because the secret value itself is not sent, it is not possible for an attacker to modify an intercepted message. As long as the secret value remains secret, it is also not possible for an attacker to generate a false message.



Fig. Message authentication using a secret value

Now assume that the content of the message box is **the digest of the current running program on an MCU.** Then we have the scenario depicted in Fig. ???.



Fig 1. Authentication protocol using an ATSHA.

As the figure demonstrates, if the host sends the digest of the **current running program** together with the MAC to the client (ATSHA):

1- Because only the host is aware of the secret key, if ATSHA can verify the authenticity of the message, it can ascertain that it was sent by the real host (an MCU in our case). Moreover, it assures the client that the message has not changed.

2- If the message is authenticated, the client can now verify whether the digest sent with the MAC is equal to the original digest by a simple comparison. If this is also true, ATSHA confirms the authenticity of the host software. In other words, it authenticates the host.

Hence, the first step confirms that host is genuine (and not an adversary). The second confirms that the software programmed on a genuine host is correct (and not forged or tampered with). Once again, it is clear that **the success of the above method depends on the secrecy of key on both sides.**

So based on the above, this is what we need to in the MCU:

1. Compute the digest of the current running program.
2. Send the MAC together with the current digest to ATSHA.

ATSHA must do the following:

1. Identify MAC from the digest.
2. Compute the MAC based on the digest part of the message.
3. Compare the calculated MAC with the one received from MCU and decide on the authenticity of the message.
4. If MAC is verified, compare the digest with the one inserted in the message and verify the authenticity of message.

There is a simpler approach that eliminates the need for storing the original program digest in the client and comparing it with the current program digest after message authentication. This is given in Fig. 1.



Fig. Authentication using signature instead of MAC

As the figure demonstrates, when generating MAC in the host, we use the **digest of the original program** instead of the current digest of program. Hence the MAC in this case would actually be the **signature** of the original program. Note that if an adversary alters either the digest or MAC after it is sent to ATSHA, it will not be verified, and ATSHA determines the presence of an adversary (more precisely, man in the middle attack). Further, if the current digest of the program is different from the original one, comparison of MACs in ATSHA will result in a failure. Therefore, the above method is a solid way of determining both the authenticity of a message sent to ATSHA and the program stored in the host.

The above algorithm though powerful has one **major setback**. Assume that the host takes the program hash, computes the signature (MAC) and sends both to ATSHA. Because the hash does not change, an attacker can attack the program as follows: every time the host sends the hash plus MAC, the attacker that is staying between the host and client replaces the current hash with the hash of the original program (which the attacker can easily obtain by observing the communication line). Using this scheme, the client always receives the correct digest, hence it will authenticate the host. Note that this attack is successful regardless of updating the keys for every authentication, given that MAC is always generated with the same hash (that is the original program hash). Hence, we also need to randomize current program hash in the host, so that this attack would be unsuccessful.

A simple way to randomize the program hash is to append a nonce to the current program digest and then compute another digest. Note that the same random number must be appended to the original program digest for computation of signature. Therefore we have what is depicted in Fig. ???. As the figure demonstrates, both the original and current program hash would be randomized in the same manner, using the MAC command of ATSHA which appends data to CPD or OPD as depicted in the figure. From this point on, RCPD refers to Randomized Current Program Digest and ROPD refers to Randomized Original Program Digest.

**Note:** It is vital to securely store OPD (Original Program Digest) safely in the MCU. The procedure for storing OPD is the same as that of storing a secret key (see next section). Thus:

* + OPD is encrypted the same way that a parent key is encrypted.
  + Encrypted OPD is stored in flash memory of MCU same as encrypted parent key.



Fig. Randomizing original and current program digest

### Personalization using ATSHA unique serial number

A nice feature of ATSHA chip is that every available chip in the market has a unique serial number. This serial number can be used for **personalization** of authentication.

Personalization allows the authentication process to be performed with only **one ATSHA chip.** If the authentication algorithm is not personalized and assuming the secret key is equal for all the products of the company, then it is possible to perform authentication with any one of the ATSHA chips programmed. This is the reason why personalization is important (to an extent)!

Personalization could be performed in several ways. For example, if each ATSHA and software possess a unique key, the authentication algorithm would be personal. The problem is that this is difficult to implement in real world scenarios (as is the case for SADR electronics boards). Hence, we use ATSHA’s serial number for the purpose of personalization. In this project, we use two personalization methods:

1. **Signature personalization:** We use ATSHA serial number to generate personal signature,
2. **Secret key personalization:** We use ATSHA serial number in generation of secret key.

Using ATSHA serial number for generation of secret key is called **key diversification** and the resulting key is called a **diversified key**. In what follows, we discuss each method.

* **Signature personalization**

In order to personalize signature for each ATSHA, the signature (or MAC) that is used for program authentication is generated by feeding the data in the following figure to SHA256 engine (Note that this arrangement is exactly like the one used in MAC or CheckMac command of ATSHA):



Fig. Arrangement of data used for generation of program signature

This signature is shaped using the **MAC** command of ATSHA. The signature formed like the above figure has the following properties:

* **Randomness:** It is different for every authentication due to the presence of Nonce that changes the original program digest. Moreover, the secret key would change with every authentication session (as discussed later).
* **Personal**: The presence of ATSHA serial number makes the MAC personal. This means that for every electronic board produced by the company, only a unique ATSHA chip can perform the authentication.

Note that at some point during the connection of ATSHA chip and the host, the serial number must be transmitted from ATSHA to the host. This can happen every time the program tries to authenticate with the chip.

* **Secret key personalization**

It is also possible to combine the secret key with ATSHA serial number using SHA256. This results in a stronger personalization than MAC personalization, given that the secret key is never revealed to any user. A key generated in the above manner is said to be diversified.

In order to generate a diversified key, we feed the data in the following figure to SHA256 engine (Note that this arrangement is exactly like the one used in MAC or CheckMac command of ATSHA):



Fig. Arrangement of data for generation of diversified key

The diversified key generated like above would personalize key generation for every device. **In this project, we use the diversified version of secret key (parent key) to generate the boot secret for copy mode (see SECTION ON CheckMAC command).** It is important to understand that the diversified key must be generated using parent key (see next section). Note that the diversified key is programmed into ATSHA using ATSHA programmer (see appendix II).

**Note:** It is also possible to combine secret key with the serial number using GenDig command of ATSHA. This approach is not discussed in this project.

## Key generation and safe storage

The secret key in symmetric authentication is an integral part and every effort must be made in order to keep it secret both on the host and client. Keeping the 32 byte key secret on the client side which has an ATSHA is guaranteed to be safe. For the host side, however, where there is an unsafe MCU, the problem is a bit complicated. Another desirable property is to have a new random secret key for every authentication session. This would make it a lot harder to determine the secret key and determine it for the next authentication session.

One way to achieve a secret key for every session is to generate a new key based on an **original** secret key. To generate the new key, we combine the **original** key with a Nonce (and perhaps some other data) and then compute SHA256 of the resulting sequence (See Fig.???). Note that the original key remains unchanged and secret throughout this process. The original key in this process is called the **parent key** and the newly produced key is called the **child key**. The child key can be produced from the original key using the DeriveKey command of ATSHA.



Fig. Random new key (child key) generation based on the original key (parent key)

As mentioned earlier, another problem is that the MCU also needs to safely store the secure parent and child key. A nice property of ATmega CPUs is that it is very difficult to extract data from the SRAM (because this data is destroyed after garbage collection or power-down). Moreover, extracting some data scattered randomly in the flash memory of these CPUs is also quite difficult. Hence we do the following for key generation in an MCU:

* The parent key is broken into four blocks of size 8 bytes.
* Each block is encrypted using an encryption algorithm.
* Each block is stored in a different place of flash memory. Hence, the MCU would always have access to the parent key. Moreover, the parent key would be read-only. Therefore the parent key cannot be modified.
* When using the parent key, the decrypted parent key is stored in SRAM.
* In each authentication session, the produced child key is stored in SRAM. It is garbage collected by the end of authentication session. Hence, the child key would disappear after every authentication session.

Based on the above discussion, we would be able to generate a random key per every authentication session and safely store parent and child key in the MCU.

### Generating Diversified Key for Boot Secret

Besides generating the child key, a diversified key is also generated using the parent key. As will be discussed in chapter III, this diversified key is used as boot secret. Note that the diversified key is produced only once during the life of ATSHA and is stored in it using ATSHA programmer. Details are given in appendix II.

### Parent Code Encryption

The encryption and decryption of the parent key follows this simple procedure:

* The four LSB and MSB of each byte of the parent key are swapped and then a logical NOT is performed on every byte.
* The resulting byte is XORed with the byte 0x45[[1]](#footnote-1) and then stored in flash memory.

For decryption, the above algorithm is reversed (first XOR and then swap bytes). **Note that the decrypted parent key is stored in SRAM**.

### Storing Secrets in MCU

The discussion on how data is stored in AVR flash and SRAM is given in appendix I. This is summary of the discussion:

* A data given by keyword *\_\_flash* stored in the flash memory of chip. This data is read only.
* An initialization data is also stored in flash. For example: *int x = 0* defines a value of zero in flash. In runtime, the zero value is fetched from flash and is stored in SRAM.
* All other data that are created during runtime (such as automatic variables and etc.) are stored in SRAM.

Hence, to store (a block of) the **encrypted parent key** in flash, we use the keyword *\_\_flash* like the example given below:

|  |  |
| --- | --- |
| unsigned char \_\_flash parent\_key1[4] = {x, x, x, x} | (‎0‑1) |

Further, to store the XOR byte of encryption function for key storage, we define:

|  |  |
| --- | --- |
| unsigned char \_\_flash XOR\_Byte = {y} | (‎0‑2) |

Finally, the result of key decryption (or the arent key) and child key is stored as an automatic (perhaps static) variable in SRAM. Hence, the keys would be destroyed at the desired points of time and never eternally available. Note that the use of static keyword does not change the fact that data is stored in SRAM.

**Note:** OPD is stored in the MCU in the same manner as the parent key.

### Choosing the Parent Key

Note that when choosing the secret key, the exact content of the key is not important, so long as it is random enough. The more important problem is that it is impossible (for the time being) to provide a unique parent key for every software and hardware produced in the company. This is because:

* The secret key has to be stored in the flash memory of the MCU. This means that the key must be programmed in the software. Hence in order to change the key for every software, we must write a new version of the software.
* Programming a different parent key in ATSHA for every software is also quite difficult. There is also the matter of synchronizing the ATSHA chip with the corresponding software.

The above discussion implies that we will have to use the same parent key for every product. This has the following setback:

**If the secret key is compromised on one board, then all products are at risk.**

Hence, it is best to at least choose one parent key for every line of products (for example a parent key for product A, a parent key for product B and so on). It would be very helpful to find ways of implementing different parent keys for all products in the future.

## The Proposed Authentication Algorithm (Basic Scheme)

Based on the discussion given above, the following authentication algorithm is proposed



Fig. The basic authentication algorithm

As the figure demonstrates, authentication requires three separate steps which are implemented at various time points:

* **Step one – Before production:** OPD must be computed. Moreover, an appropriate parent key must be chosen.
* **Step two – On the production line:** The parent key must be stored on both the host and client. Moreover, OPD must be programmed on the host. Data storage on ATSHA is performed using ATSHA programmer (see the report on phase I).
* **Step three – Authentication protocol (During product consumption)**: The authentication protocol is executed at appropriate points during product consumption.

The authentication program works as discussed in the previous sections. During each authentication, an equivalent new child key is produced on the host and client. Next, the random program digest is generated on the host side and is sent to ATSHA together with the signature. This signature is compared with the one produced on the client side using the child key, and is compared with the transmitted MAC. This comparison determines whether or not the program is authentic.

The verify ACK box depicted in the above figure is an important part of the authentication procedure and is discussed in the following section.

### Verifying ACK/NACK After Authentication

A major issue of the above algorithm is that after performing CheckMac, ATSHA only returns a value of 0x00 to indicate that authentication was successful. Hence, an adversary can attack the algorithm by replacing ATSHA with a chip that sends this 0x00 at the right time.

To overcome the aforementioned issue, ATSHA provides an operation mode called copy mode. In copy mode, in addition to returning the boolean to the system, the device copies a separate secret to a temporary key storage location only if the comparison succeeds. This separate secret can be combined with a unique challenge from the system to generate a unique response. The secret produced in the copy mode is referred to as the boot secret.

This is how the boot secret is created. After performing the authentication, the child key is stored the temporary storage location. Next, it is combined with a Nonce, and then it is sent to the host. In the host, the same nonce is combined with the child key. Next, the value sent from ATSHA is compared with the value generated on the host side and if the results are equal, the authentication is considered successful.

## Bootloader

How does bootloader provide access to the flash memory? If this is in form of blocks of certain size, how does one compute the digest of data?



# ATSHA204A CHIP

## Introduction

This section takes a look at the features of ATSHA that are needed in order to implement the authentication algorithm discussed in the previous chapter (see also the report on phase I). The chapter is divided into two main parts. In the first part, we discuss the memories and storage units of ATSHA. In the second part, we discuss the commands needed to implement the authentication algorithm.

## ATSHA Data Storage Units and Buffers

ATSHA provides three data storage units:

* **Configuration Zone (CZ):** Used for general settings of ATSHA.
* **Data Zone (DZ)**: Used for key and data storage.
* **One Time Programmable zone (OTP zone):** Used for storing read only data or fuse type data.

Moreover, ATSHA has two data buffers:

* **I/O buffer:** A buffer of size 84 bytes used for sending command, getting results and etc. **Hence, no command of ATSHA can have a length longer than 84 bytes.**
* **Tempkey register:** A register that is used to store the temporary data in ATSHA. Given that it is a RAM type register, its content is destroyed if power is cut-off or device goes to sleep mode. However, the content of this register is preserved if the device goes to idle mode. **At no point during the operation of ATSHA could the content of Tempkey register be read**.

Fig ??? demonstrates the data flow between the data zones, SRAM, I/O buffer and the ALU unit (command unit) of ATSHA. As the figure demonstrates, DZ and CZ could be accessed via I/O buffer. Moreover, the result of ATSHA commands can be read from I/O buffer. However, Only ALU can put data in SRAM, or read from it. On the other hand, ATSHA can ask that some data in DZ be put in SRAM or vice versa. The most important point here is that the user does not have direct access to SRAM.



Fig. Data flow between ATSHA buffers and zones

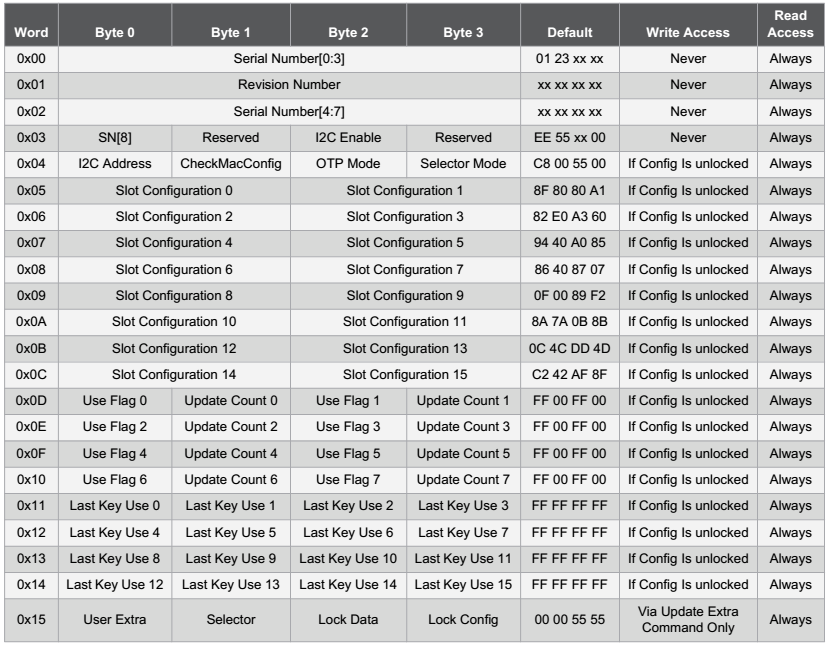
In what follows, first we discuss how to program CZ for various needs in this project. Then we briefly discuss DZ and Tempkey.

### Configuration Zone (CZ)

A brief discussion of CZ was given in phase I of this project and all parameters of this zone were set assuming no encrypted read or writes and no application of DeriveKey command. In this report, once again we review how to configure this zone for the authentication algorithm required in this report. An overview of CZ is given in Fig. ????.

**Note:** Note that CZ is programmed using ATSHA programmer. Hence, no operation is performed on this zone during authentication. Moreover, the way the data zones are locked follows the same procedure as the one discussed in the previous phase.

Fig. Overview of CZ



Various data sections of this zone are discussed below.

* **CheckMacConfig**

This byte applies only to the CheckMac, Read, and Write commands:

* + **Read and Write:**

Bit 0 controls Slots 0 and 1, bit 1 controls Slots 2 and 3, and so on. Any encrypted Read/Write command will fail if the value in TempKey.SourceFlag does not match the corresponding bit in this byte. This byte is ignored for clear text reads and writes.

* + **CheckMac:**

Bit 0 controls slot 1, bit 1 controls Slot 3 and so on. The copy function will only be enabled if the CheckMacSource value corresponding to the target slot matches the value of Mode bit 2 of the CheckMac command. The command will fail if Mode bit 2 does not match TempKey.SourceFlag, so this is equivalent to requiring the corresponding bit in this byte to match TempKey.SourceFlag.

* **SlotConfiguration**

As discussed in phase I of this projects, these slots determine the access level to each slot of DZ. These slots were set to 0x0000 in phase I, which implied that all data could be read and written with no limit. In this project, however, we would employ some other aspects of these bytes.

Configuration of SlotConfig depends on which of the following is intended for that slot:

* Encrypted, plain text or no Read/Write
* The slot is used for DeriveKey command
* The slot is used for CheckMac or GenDig command

SlotConfig field is interpreted according to Table ??? when the Data zone is locked. When the Data zone is unlocked, these restrictions do not apply, and all DZ slots may be freely written and none may be read.

**Note:** Each set of bits in this slot are referred to using a dot. Moreover, the slot for which SlotConfig is configured is shown by a bracket. Hence, SlotConfig[0].ReadKey means ReadKey bits for slot zero.

Table. . SlotConfig bits

|  |  |  |
| --- | --- | --- |
| Bit | Name | Description |
| 0-3 | ReadKey | Slot of the key to be used for encrypted reads.  If 0x0, then this slot can be used as the source slot for the CheckMac Copy Command. |
| 4 | CheckOnly | 0 = This slot can be used for all crypto commands.  1 = This slot can only be used for CheckMac and GenDig followed by CheckMac Commands. |
| 5 | SingleUse | 0 = No limit on the number of time the key can be used.  1 = Limit on the number of time the key can be used based on the UseFlag (or LastKeyUse) for the slot |
| 6 | EncryptRead | 0 = Clear reads are permitted.  1 = Requires the slot to be Secret and encrypted read to access. |
| 7 | IsSecret | 0 = The slot is not secret and allows clear read, clear write, no MAC check, and no Derivekey Command.  1 = The slot is secret. Reads and writes if allowed, must be encrypted. |
| 8-11 | WriteKey | Slot of the key to be used to validate encrypted writes.  **Note**: The LSB of SlotID is set in bit 8. |
| 12-15 | WriteConfig | Specific configuration for DeriveKey and Write command are required. These are given below. |

The IsSecret bit controls internal circuitry necessary for proper security for slots in which reads and/or writes must be encrypted or are prohibited altogether. It must also be set for all slots that are to be used as keys, including those created or modified with DeriveKey. Specifically, to enable proper device operation, this bit must be set unless WriteConfig is “Always”. 4-byte accesses are prohibited to/from slots in which this bit is set.

* + **Read restrictions:** The following table demonstrates the type of data read can be performed based on the values of IsSecret, EncryptRead and ReadKey.

**Note:** This table belongs to the Read command of ATSHA and is brought here for convention.

Table. Restrictions on writing to a DZ slot based on SlotConfig values.

|  |  |  |
| --- | --- | --- |
| IsSecret | EncryptRead | Description |
| 0 | 0 | Clear text reads are always permitted from this slot.  Slots set to this state should never be used as key storage.  Either 4 or 32 bytes may be read at a time. |
| 0 | 1 | Prohibited. No security is guaranteed for slots using this mode. |
| 1 | 0 | Reads are never permitted from this slot.  Slots set to this state can still be used for key storage. |
| 1 | 1 | Reads from this slot are encrypted using the encryption algorithm of Read command.  The encryption key is in the slot specified by ReadKey. 4-byte reads and writes are prohibited. |

Hence, when trying to configure a slot to hold a secret key, IsSecret = 1 and EncryptRead = 0, so that the secret (parent or child keys) keys can never be read. However, it can be used for all other crypto-authentication tasks (MAC, DeriveKey, and etc.)

* + **Write restrictions:** The following table demonstrates the type of data read can be performed based on the values of IsSecret, WriteConfig and WriteKey.

Table. WriteConfig Bits — Write command

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bit 15 | Bit 14 | Bit 13 | Mode Name | Description |
| 0 | 0 | 0 | Always | Clear text writes are always permitted on this slot. Slots set to “always” should never be used as key storage. Either 4 or 32 bytes may be written to this slot.  **Note**: IsSecret should be zero for this mode to function. |
| X | 0 | 1 | Never | Writes are never permitted on this slot **using the Write command**  Slots set to “never” can still be used as key storage  **Note**: IsSecret should be one for this mode to function.  **Note**: DeriveKey command can still use this slot. |
| 1 | 0 | X | Never | Writes are never permitted on this **slot using the Write command**  Slots set to “never” can still be used as key storage.  **Note**: IsSecret should be one for this mode to function.  **Note**: DeriveKey command can still use this slot. |
| x | 1 | X | Encrypt | Writes to this slot require a properly computed MAC, and the input data must be encrypted by the system with WriteKey using the encryption algorithm documented in the Write command description (which requires “UpdateExtra Command”). 4-byte writes to this slot are prohibited.  **Note**: IsSecret should be one for this mode to function. |

Hence, when using a slot for secret key storage, IsSecret = 1 and Bits 13 to 15 should either be 0b10x or 0bx01, so that the slot with secret key can never be overwritten. However, it can be used for all other crypto-authentication tasks (MAC, DeriveKey, and etc.)

Now assume that a slot is going to be used by DeriveKey command. The following table shows the configuration of WriteConfig with respect to DeriveKey command. Note that Bit 13 must be set, or DeriveKey command would not be allowed on this slot. Note also that Bit 15 determines the need for MAC when using DeriveKey (see DeriveKey command).

Table. WriteConfig Bits — Derivekey command

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Bit 15(1)  (MAC bit) | Bit 14 | Bit 13(2)  (DeriveKey allowed) | Bit 12(3)  (Roll-key or create key) | Source Key(4) | Description |
| 0 | X | 1 | 0 | Target | DeriveKey command can be run without authorizing MAC (Roll). |
| 1 | X | 1 | 0 | Target | Authorizing MAC required for DeriveKey command (Roll). |
| 0 | X | 1 | 1 | Parent | DeriveKey command can be run without authorizing MAC (Create). |
| 1 | X | 1 | 1 | Parent | Authorizing MAC required for DeriveKey command (Create). |
| X | X | 0 | X |  | Slots with this value in the WriteConfig field may not be used as the target of the DeriveKey command. |

**Note 1:** MAC bit denotes whether or not MAC checking is required for DeriveKey command on this slot.

**Note 2:** This bit determines whether or not DeriveKey command is allowed to be performed on this slot.

**Note 3:** This bit determines whether roll-key or create key operation can be performed on this slot.

**Note 4**: The source key for the computation performed by the DeriveKey command can either be the key directly specified in Param2 (the “Target”) or the key at slotConfig[Param2].WriteKey (the “Parent”). Hence, a slot on which DeriveKey is performed is called “Target” and the slot denoted by slotConfig[Param2].WriteKey is the parent of this slot. See the DeriveKey command section for more details.

To explain the configuration mentioned in the table, note that DeriveKey command can be used in two different ways to create new keys:

* **Roll keys**: combine a (parent) key with ATSHA serial number to create a new (child) key
* **Create keys**: Combine the current parent key with a Nonce to generate a new child key.

To allow each of the above operations to take place, we can also ask the user to provide a MAC. That way, an unauthorized change of the keys would be impossible (see DeriveKey command section). Note that DeriveKey accepts as input a parameter denoted by Param2, which holds a four bit address of a slot.

In this project, we intend to use the “create key” approach. When using this approach, we need to specify the parent key and the target slot in which the key is stored. Setting SlotConfig.WriteConfig to either 0b0x11 or 0b1x11 for bits 12 to 15 would imply that the parent key used for DeriveKey command is slotConfig[Param2].WriteKey, and the newly generated child key would be stored in the slot denoted by Param2. The configuration of slots for use with DeriveKey command is discussed in the corresponding section.

**Note:** The SinlgeUse field is typically used for one time used keys. This topic is not discussed in this project.

**Note:** The CheckOnly field is not used in this project and its value is always set to zero as well.

## DZ Configuration

A detailed discussion of DZ configuration is given in the report of phase I. The purpose of this section is to illustrate the specific use of each slot of DZ. This is given in the following table. Note that the main difference between limited use keys and single use key is that single use key can only be used 128 times and after that the key is disabled. Limited use keys can however be reset and again used 8 times. A detailed discussion on single use and limited use keys is given at section 13 of [ATSHADatasheet]. Moreover, a brief discussion is given on DeriveKey command description of this report.

Table. Usage of each data slot of DZ.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Slot | Encrpted Read/ Write | Copymode of CheckMac command | Single use key storage | Limited use key |
| 0x00 | **✓** |  | **✓** |  |
| 0x01 | **✓** | **✓** |  |
| 0x02 |  | **✓** |  |
| 0x03 | **✓** | **✓** |  |
| 0x04 |  | **✓** |  |
| 0x05 | **✓** | **✓** |  |
| 0x06 |  | **✓** |  |
| 0x07 | **✓** | **✓** |  |
| 0x08 |  |  |  |
| 0x09 | **✓** |  |  |
| 0x0A |  |  |  |
| 0x0B | **✓** |  |  |
| 0x0C |  |  |  |
| 0x0D | **✓** |  |  |
| 0x0E |  |  |  |
| 0x0F | **✓** |  | **✓** |
| Note: CheckMacConfig is not defined for the even slots when using CheckMac command.  Note: SingleUse bit is ignored for slots 0x08 to 0x0E. Hence, single use property is not defined. | | | | |

### TempKey (ATSHA SRAM)

The device includes an SRAM Array that is used to store the input command or output result, intermediate computation values, and/or an ephemeral key. The entire contents of this memory are always invalidated whenever the device goes into sleep mode or the power is removed. The ephemeral key is named TempKey, and can be used as an input to the MAC, HMAC, CheckMac, GenDig, and DeriveKey commands. It is also used as the Data protection (Encryption or Decryption) key by the Read and Write commands.

TempKey is a storage register in the SRAM array that can be used to store an ephemeral result value from the Nonce, GenDig, CheckMac, or SHA commands. The contents of this register can never be read from the device (although the device itself can read and use the contents internally). The following table demonstrates the fields of this register.

Table . The SRAM registers of ATSHA

|  |  |  |
| --- | --- | --- |
| Name | Bit Length | Description |
| TempKey | 256  (32-bytes) | Nonce (from Noncecommand) or Digest (from GenDigcommand). |
| SlotID | 4 | If TempKey was generated by GenDig (see the GenData and CheckFlag bits), these bits indicate which key was used in its computation. The four bits represent one of the slots of the Data zone. |
| SourceFlag | 1 | The source of the randomness in TempKey (see Nonce command):  0 = Internally generated random number (Rand).  1 = Input seed only, no internal random generation (Input). |
| GenData | 1 | 0 = TempKey.SlotID is not meaningful, and is ignored.  1 = The contents of TempKey were generated by GenDigusing one of the slots in the Data zone (and TempKey.SlotID will be meaningful).  **Note:** Not relevant to this project. |
| CheckFlag | 1 | If one, the contents of TempKey were generated by the GenDig command and at least one of the keys used in that generation is restricted to the CheckMac command (SlotConfig.CheckOnly is one); otherwise, this bit will be zero. |
| Valid | 1 | 0 = The information in TempKey is invalid.  1 = The information in TempKey is valid. |

The name “TempKey” refers to the contents of the 32-byte (256-bit) Data register. The remaining bit fields are referred to as TempKey.SourceFlag, TempKey.GenData, and so on. The TempKey.Valid bit is cleared to zero under any of the following circumstances:

* Power-up, sleep, brown-out, watchdog expiration, or tamper detection. The contents of TempKey are however retained when the device enters idle mode.
* After the execution of any command other than Nonce or GenDig, regardless of whether or not the command execution succeeds. It may be cleared by the CheckMac command unless a successful copy takes place. It is not cleared if there is a communications problem, as evidenced by a Cyclic Redundancy Check (CRC) error.
* An error during the parsing or execution of a GenDig and/or Nonce command.
* Execution of GenDig replaces any previous output of the Nonce command with the output of the GenDig command. Execution of the Nonce command likewise replaces any previous output of the GenDig command.

**Note that the only exception to the law of data invalidation in TempKey is when CheckMac copy mode is performed. In this case, setting ATSHA even to idle mode causes the boot secret in TempKey to be invalidated. This must be considered when implementing the copy mode.**

## ATSHA Commands

A common feature of all commands that use SHA256 of ATSHA (except for SHA command itself) is:

**They all provide a message of size 88 byte as input to SHA command, and they provide the padding bytes automatically.**

Hence when using any command (other than SHA command) we must make sure that the input to the SHA256 engine has exactly 88 bytes.

### Implementation of SHA256

As explained in section ‎1-1-, implementation of SHA256 is iterative. This means that we break the message into blocks of 512 bits and using the results of the previous hash stage, we compute the new hash output with the current message block.

The ATSHA chip offers two commands that employ such a technique, namely SHA, MAC. Both these commands work in almost the same fashion: first they are initialized to produce an starting point for computation of digest (or as mentioned in section ‎1-1-). Next, they get either blocks of length 32 or 64 byte to iteratively compute the hash. The output value of each step would be available in Tempkey, and the final value can be read from the output buffer. Of the two commands mentioned here we prefer to use MAC (as discussed in section ), which is suitable for our implementation.

#### Software Implementation of SHA256

To compute the same hash in software, ATMEL provides a library function that works exactly like the MAC command of ATSHA. Calling this function ATSHA\_MAC() (it is defined under the name uint8\_t\_sha204h\_mac() in the original definition, see [ATSHA204 helper library]), the function input parameter is a pointer to a structure of type sha204h\_mac\_in\_out defined as follows:

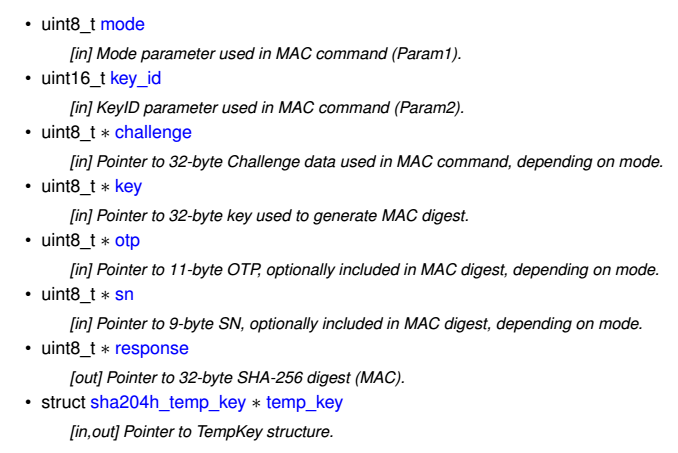


Fig. sha204h\_mac\_in\_out data structure fields

The last item is pointer to a structure of type sha204h\_temp\_key, which is defined as follows:

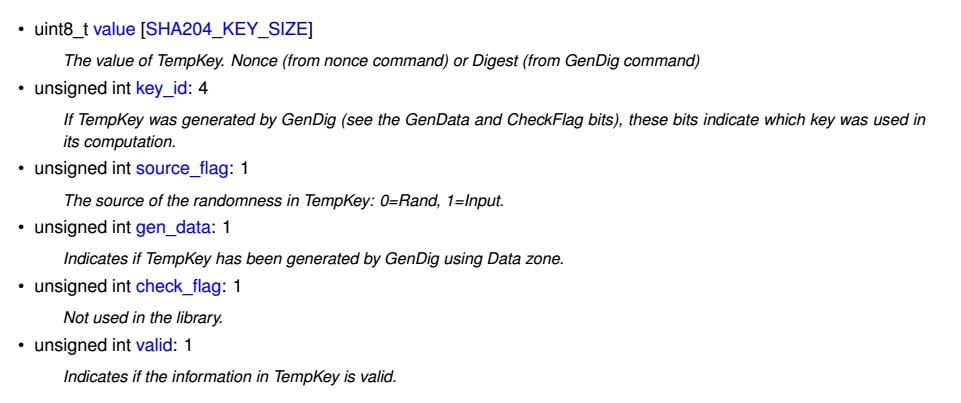


Fig. sha204h\_temp\_key data structure fields

Now to employ the ATSHA\_MAC() to compute the digest of a message (that is a complete padded message), we must do as follows:

The goal of the remainder of this section is to define an algorithm that computes the digest of the program stored in MCU in an iterative manner. This algorithm has the following properties:

* It adds the required padding to the final blocks of the message as necessary.
* It sends a 32 byte block of the message to ATSHA\_MAC() in each step to iteratively compute the hash.

In what follows we assume that a message to be hashed is presented as 8 bit unsigned integers (or **uint8**). We have the following definitions:

|  |  |
| --- | --- |
| Definition | Description |
| message[32] | An array denoting a block of input message |
|  | The length of message (not including padding).  Note: should be of format uint64, to accommodate for the length of very large programs. Note that the maximum message length for SHA256 is bits. |
|  | ( denotes the total number of 256 bit message blocks) |
|  | ( denotes the total number of 512 bit message blocks) |
|  | and ( denotes the length of the remainder of message that does not form a 512 bit size block) |
|  | and (Number of 8 bit zeros that are appended to the message, after appending 0x80)  **Note:** 0x80 is always added to the final block of message to pad a one at the end of message. This why we subtract one in the above equation, to account for adding these 8 bits.  **Note:** is computed as follows:   * If : then * Else: |

(The padding might be wrong, most probably 0x80 must not be used in all cases and in cases perhaps length bytes must be added to it as well!)

Recall that a message that is to be hashed by SHA256 should be divided into blocks of length 512 bits, denotes how many of these blocks are present in the message. denotes the number of uint8 at the end of the message that do not constitute a block of 512 bits. Note also that 0x80 is always added to the end of the message to attach the “1” bit required to form the message. Given that this is always attached to the end of the message, we must determine how many uint8s are left to the 448th bit of this 512 bit block after 0x80 is attached to it. This number is determined by k. Note that if after appending 0x80 we are past the 448th bit, we need to generate a new block of size 512, and pad enough zeros at its beginning to reach the 448th bit of this new block.

Now given the above definitions, the following pseudo code is used to compute the hash of a message M (to which we only have a pointer of uint16 type) in the standard form using ATSHA\_MAC() function.

**PSEUDO CODE IS IN LATEX**

### MAC Command

The MAC command of ATSHA uses the internal SHA256 engine of ATSHA to compute the MAC, given certain input data to the command. In this section first we discuss the implementation of this command in ATSHA chip, then we briefly discuss the standard software implementation function provided by ATMEL called sha204h\_mac().

To briefly discuss the logic behind MAC calculation algorithm, consider Fig. 1. As the figure demonstrates, when calculating MAC the input to the SHA256 is 128 bytes (instead of only 64 bytes comprising the key and challenge). As was explained in the section … (the section in the previous chapter on how MAC should be generated), when generating the MAC, ATSHA also uses several of its internal data as denoted in the figure to randomize the MAC. Hence for example, ATSHA either uses SN[2:3] or two bytes of zero instead for the MAC command. Therefore, MAC command uses 88 bytes to produce an output. Note that since ATSHA computes SHA256 using the standard implementation, the padding bytes are added to the data array (which makes the sum of data to 128 bytes). Note however that these data bits need not to be provided as input for the command, given that ATSHA chip automatically adds them to the end of the MAC message.

Finally note that the bytes Mode, Opcode and Param2 refer to the values used for MAC command (see next section).

To demonstrate two simple and most important use cases, consider the following:

* **MAC command used to generate MAC:**

To generate the MAC in ATSHA using key[SlotID], the host asks ATSHA to put key[SlotID] in the first 32 bytes, and the challenge (current program hash) in the second 32 bytes. Host may also ask that ATSHA put any additional data it desires from the remainder of Fig. . Now, MAC command computes the digest of the above vector and puts the result in the output buffer.

Note that if we wish to use key[SlotID] as a parent key and derive other keys based on it, first we run Nonce or GenDig command to derive a new key based on this key and save it in TempKey, and then used this new key instead of key[SlotID].

* **MAC command used with nonce to create new keys (authenticate chip):**

In this mode, the host sends a nonce to ATSHA and ATSHA produces a random number based on it and combines this random number with the key and sends the corresponding digest to the host. In order to do so, the host first runs the Nonce command and save the result in TempKey. Next, the host asks ATSHA to put key[SlotID] in the first 32 bytes and TempKey in the second 32 bytes. The host can ask ATSHA to put any additional data from the remainder of Fig. . Now, MAC command computes the digest of the above vector and puts the result in the output buffer.

**Note:** The auxiliary data in Fig. are constant (as mentioned in [ATSHADATASHEET]). This means that these data (including OTP data, serial number) would not change during the entire operation of authentication protocol. The only bytes that are subject to change are Mode, and Param2.



Fig. Data array used for calculation of MAC (input to SHA256 algorithm) used in ATSHA chip.

#### MAC Command in ATSHA

The following is from [ATSHA data sheet].The MACcommand computes a SHA-256 digest of a key stored in the device, a challenge, and other information on the device. The output of this command is the digest of this message. If the message includes the serial number of the device, the response is said to be “diversified”.

The normal command flow to use this command is as follows:

1. Run the Nonce command to load input challenge and optionally combine it with a generated random number. The result of this operation is a nonce stored internally on the device within tempkey.
2. Optionally run the GenDig command to combine one or more stored EEPROM locations in the device with the nonce. The result is stored internally in the device within tempkey. This capability permits two or more keys to be used as part of the response generation.
3. Run this MAC command to combine the output of Step 1 (and Step 2 if desired) with an EEPROM key to generate an output response (or digest).

Note that the first two steps of the above are optional.

Table . shows the input data to ATSHA chip when using MAC command. The mode parameter is very important in this command, in that it determines the data to be used for calculation of MAC. This parameter is illustrated in Table .

**Note:** Based on the software implementation of this command (next section), the data in TempKey is invalidated after this process.

Table . MAC command Input Parameters

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Size | Note |
| Opcode | MAC | 1 | 0x08. |
| Param1 | Mode | 1 | Controls which fields within the device are used in the message. |
| Param2 | SlotID | 2 | Which internal key is to be used to generate the response.  Bits 0:3 only are used to select a slot but all 16 bits are used in the SHA-256 message. |
| Data | Challenge | 0 or 32 | Input portion of message to be digested, ignored if Mode:0 is one |

Table . MAC command Input Parameters

|  |  |  |
| --- | --- | --- |
| Name | Size | Note |
| Response | 32 | SHA-256 digest. |

Table . MAC command mode parameter

|  |  |
| --- | --- |
| Bits | Note |
| 7 | Must be zero. |
| 6 | If set, include the 48 bits SN[2:3] and SN[4:7] in the message; otherwise, the corresponding message bits are set to zero. |
| 5 | Include the first 64 OTP bits (OTP[0] through OTP[7]) in the message; otherwise, the corresponding message bits are set to zero.  If Mode[4] is set, the value of this mode bit is ignored |
| 4 | Include the first 88 OTP bits (OTP[0] through OTP[10]) in the message; otherwise, the corresponding message bits are set to zero. |
| 3 | Must be zero. |
| 2 | If either Mode:0 or Mode:1 are set, Mode:2 must match the value in TempKey.SourceFlag or the command will return an error. |
| 1 | If zero, then the first 32 bytes of the SHA message are loaded from one of the data slots  If one, then the first 32 bytes are filled with TempKey. |
| 0 | If zero, then the second 32 bytes of the SHA message are taken from the input Challenge parameter.  If one, then the second 32 bytes are filled with the value in TempKey. **This mode is recommended for all use.** |

#### MAC command software implementation

|  |  |
| --- | --- |
| **Function name** | sha204h\_mac() |
| **Input** | struct \*sha204h\_mac\_in\_out |
| **Output** | A uint8\_t denoting command execution status |

As stated earlier, MAC command is also implemented by the function sha204h\_mac() for software purposes. The input parameter of this function is a pointer to a structure of type sha204h\_mac\_in\_out, whose fields are as follows:

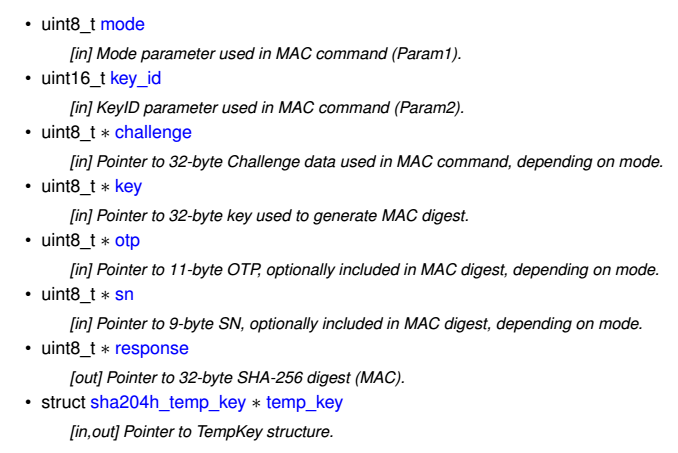


Fig. sha204h\_mac\_in\_out data structure fields.

As the figure demonstrates, the structure takes the mode and key\_id parameters used in calculation of MAC, as well as pointer to challenge data and the key. It also takes pointers to otp and sn data that are (perhaps) stored in the host software. The pointer to response denotes the output of this function, which is the 32 byte MAC. The last input of this structure is pointer to another structure called sha204h\_temp\_key. This structure models the tempkey register of ATSHA, whose fields are given in the following figure:

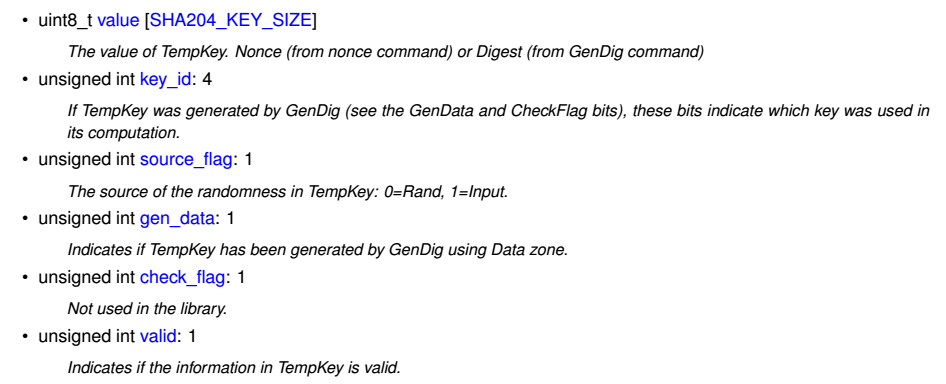


Fig. sha204h\_mac\_in\_out data structure fields.

It is clear that the fields of this structure correspond to that of ATSHA SRAM. Note the initialization values of each one of the parameters in this structure.

The flowchart of sha204h\_mac() function is depicted in Fig. . As the figure demonstrates, this function performs three basic steps: 1- checking the input parameters of MAC command, 2- filling the (88 byte) challenge array, 3- SHA256 calculation, 4- updating tempkey.

1. **Check input parameters:** In this stage, the function checks whether or not the mode given to the function is chosen properly, and whether the necessary data has been provided for the corresponding mode as well. For example, if mode parameter is set so that serial number is used for calculation of MAC, if serial number is not provided as input to the command it will return an output flag SHA204\_BAD\_PARAM. Further if the data from Tempkey is going to be used, the function check to see if the valid flag in this register is one and that check flag and source flag are both zero. The conditions that are checked and the corresponding reason are commented in the code
2. **Filling the challenge array:** After checking the input parameters, the function begins to fill the 88 byte array that is used to compute the MAC. This array is denoted by temporary[], and the pointer p\_temp is a pointer to this array which is used for all calculations. The order in which the data is put in this array is the same as Fig. (MAC command array).
3. **SHA256 calculation:** After filling the challenge array, data is sent to standard SHA256 calculation function. Note that the padding bytes should be provided for this function???.
4. **Updating TempKey:** After computation of digest, Temkey.valid is set to zero to denote that the data available in TempKey register is invalid.



Fig. Flowchart of sha204h\_mac()

### Nonce command

To illustrate this command, we should point out that ATSHA has an internal random number generator, whose outcome random number is referred to as RNG. Fig. illustrates the function of NONCE command. The output of this command is always stored in Tempkey. It may also put some data in data buffer. Nonce command has two modes:

1. **Random mode:** This mode generates a Nonce. For this end it combines a 20 byte input number with an internal RNG and computes SHA256 of this data. The input to SHA256 engine for generation of Nonce is depicted in Fig. . The Nonce generated is saved in TemKey, TempKey.Source is set as Random and TempKey.valid is set to 1. Moreover, the command puts RNG used to create Nonce in data buffer. Therefore, the system that uses this command (e.g., the host microcontroller) reads this RNG and must generate Nonce on its own.
2. **Pass through mode:** This mode is used for writing a 32 byte data in the TempKey register. When using this mode, TempKey.Source is set as input and TempKey.Valid is set to 1. No data is put in data buffer.

Note that according to [ATSHA DATA SHEET]), replay attacks against the host are prevented when using the input 20 bytes to generate Nonce. Note also that the data written to TempKey can be used for GenDig. DevKey, MAC and CheckMac commands. This way, a desired input (such as a challenge or etc.) would be available for all these commands in the TempKey register.



Fig. The block diagram of Nonce command



Fig. Input to SHA256 for generation of Nonce

In what follows, first we discuss the hardware implementation of this command on ATSHA, then briefly discuss the emulating software implementation

#### Nonce command in ATSHA

The following is from [ATSHA data sheet]. The Nonce command generates a nonce for use by a subsequent GenDig, MAC, HMAC, Read, or Write command by combining an internally generated random number with an input value from the system. The resulting Nonce is stored internally in TempKey and the generated random number is returned to the system.

The input value is designed to prevent replay attacks against the Host, and it must be externally generated by the system and passed into the device using this command. It may be any value that changes consistently, such as a nonvolatile counter, current real time of day, and so forth; or it can be an externally generated random number.

To provide a nonce value for subsequent crypto commands, the input number and output random number are hashed together according to the information listed below. The resulting digest (nonce) is always stored in the TempKey register, TempKey.Valid is set, and TempKey.SourceFlag is set to “Rand”. The nonce can be used by a subsequent GenDig, Read, Write, HMAC, or MAC command, thus the system must **externally** compute this digest value and store it externally to complete the execution of those commands.

Alternatively, this command can also be run in a pass-through mode if a fixed nonce is required for subsequent commands. In this case, the input value must be 32 bytes long, and it is passed directly to TempKey without modification. No SHA-256 calculation is performed, and TempKey.Source Flag is set to “Input.” The nonce value in TempKey may not be used with Read or Write commands. If operated in this mode and with a repeated input number value, the device provides no protection against replay attacks.

Prior to the configuration section being locked, the RNG produces a value of 0xFF FF 00 00 FF FF 00 00 to facilitate testing. This test value is combined with the input value in the manner described above.

Table ??? shows the input parameters to this command. Moreover, Table ??? demonstrates the output values of the command. Finally Table ??? shows the operation modes of this command.

Table . Nonce command Input Parameters

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Size | Note |
| Opcode | Nonce | 1 | 0x16. |
| Param1 | Mode | 1 | Controls the mechanism of the internal RNG and seed update. |
| Param2 | Zero | 2 | Must be 0x0000. |
| Data | Numln | 20, 32 | Input value from system. |

Table . Nonce command Input Parameters

|  |  |  |
| --- | --- | --- |
| Name | Size | Note |
| RandOut | 1 or 32 | The output of the RNG or a single byte with a value of zero if Mode[0:1] is three. |

Table . Nonce command mode parameter

|  |  |
| --- | --- |
| bits | Meaning |
| 2-7 | Must be zero. |
| 0-1 | If zero, then combine the new random number with NumIn, store in TempKey. Automatically update EEPROM seed only if necessary prior to random number generation. Recommended for highest security.  If one, then combine the new random number with NumIn, store in TempKey. Generate random number using existing EEPROM seed, do not update EEPROM seed (not used in this project).  If two, then is it invalid (i.e, this mode is undefined and should not be used).  If three, then operate in the pass-through mode and write TempKey with NumIn. |

#### Nonce command software implementation

|  |  |
| --- | --- |
| **Function name** | sha204h\_nonce() |
| **Input** | struct \*sha204h\_nonce\_in\_out |
| **Output** | A uint8\_t denoting command execution status |

Sha204h\_nonce() is the software realization of Nonce command in the sense that:

* It emulates the pass-through mode of ATSHA Nonce command
* It generates the same Nonce as produced by ATSHA, if RNG and input random number are provided for the function.

This function accepts a pointer to a structure of form sha204h\_nonce\_in\_out as input. The fields of this structure are given in the following figure. Note that \*rand\_out is the pointer to an array that holds the RNG generated by ATSHA chip.

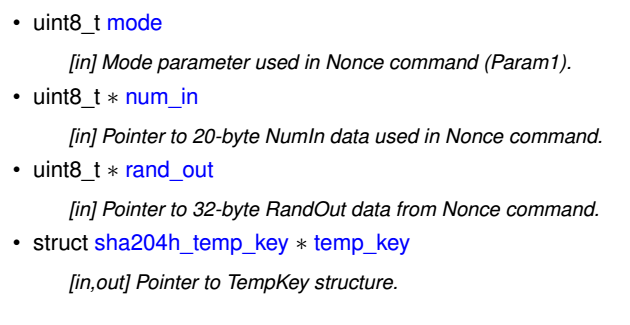


Fig. . sha204h\_nonce\_in\_out data structure fields

The flow-chart of Sha204h\_nonce() is depicted in Fig. . As the figure demonstrates, the function performs four separate actions:

1. **Check input parameters:** The function checks whether or not the mode given to the function is chosen properly (i.e., a valid mode of Nonce command), and whether the necessary data has been provided for the corresponding mode as well. This means that \*num\_in, \*rand\_out and \*tempkey must all point to an object based on the input mode, or else the function returns a SHA204\_BAD\_PARAM flag.
2. **Computation:** Based on the chosen mode, performs either of the following actions:
   1. **Pass through mode (Mode = 0x03):** In this mode, the function simply puts the content of \*num\_in in TempKey. Moreover, TempKey.SourceFlag is set to zero to denote pass through mode.
   2. **Random modes (Mode = 0x00 or 0x01):** In these two modes, the function forms temporary[] array (which is input to SHA256 function) according to the data array of Fig. (Nonce Data Structure). Then, SHA256 of temporary[] is calculated to return the resulting Nonce. Note that Nonce would be available in the input structure of SHA256 function. Moreover, TempKey.SourceFlag is set to one to denote random mode.
3. **Updating TempKey:** After computation of digest, Temkey.valid is set to one to denote that the data available in TempKey register is invalid. The other parameters of except source flag are set to zero



Fig. . Flow-chart of sha204h\_nonce() command

### DeriveKey command

Derive key command is used for generation of new keys from the old ones present in EEPROM.

Before discussing this command, we should remember that access to the data in DZ is limited by proper setting of SlotConfig fields in CZ. Any slot in the DZ of the EEPROM can be used to store a key, however, the value will be secret only if the read and write permissions are properly set within SlotConfig (including the IsSecret bit) [DATASHEET]. In what follows, we discuss the methods for updating the secret keys within DZ. Updating the secrets using GenDig command is discussed in the corresponding section. Before proceeding with this section, refer to the discussion on SlotConfig of CZ in section ????.

The DeriveKey command uses SHA256 engine of ATSHA to create new keys by combining a data present in a DZ slot with some other data available. Fig. . demonstrates how these data are combined together to form the new key using SHA256.



Fig. Arrangement of data passed to SHA256 in DeriveKey command

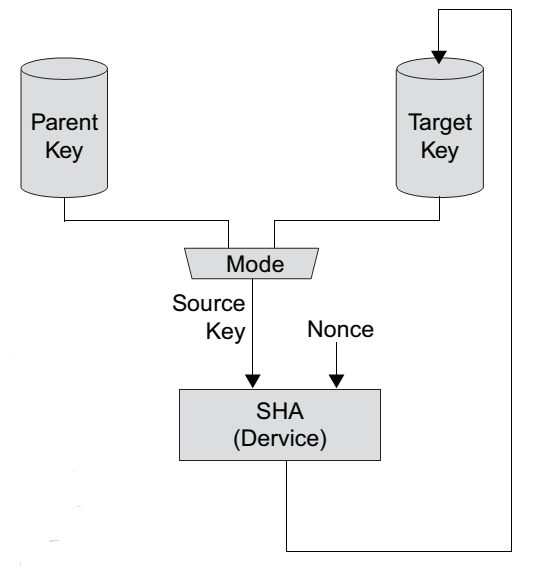
Based on the value assigned to TempKey and target or parent key, four different types of key can be made using DeriveKey. These are discussed below (the following is from [ATSHADATASHEET]).

* **Rolled key:**

In order to prevent repeated use of the same key value, the ATSHA204A supports key rolling. Normally, after a certain number of uses (perhaps as few as one), the current key value is replaced with the SHA-256 digest of its current value combined with some offset, which may either be a constant, something related to the current system (for example, a serial number or model number), or a random number. The value with which the key is going to be rolled should be loaded in TempKey using Nonce command. Note that a failure of DeriveKey command through power failure may cause the key to be permanently lost. Rolled keys are not used in this project.

* **Created Key:**

In order to support unique ephemeral keys for every Client, the ATSHA204A also supports key creation. In this mechanism, a “parent” key (**specified by** slotConfig.writeKey) is combined with a fixed or random nonce to create a unique key, which is then used for any cryptographic purpose. Creating child keys from a parent key that has limited (read/write access) is a good way to create different keys for each session. Note that each time the create operation is performed on slots 0 thru 7; the UpdateCount field for that slot is incremented. The difference between created and rolled keys is that created key operation takes the key determined by SlotConfig[TargetKey].WriteKey (parent key), creates the new key using SHA256, and stores it in SlotConfig[TargetKey]; Whereas in rolled key, ATSHA takes the key stored in SlotConfig[TargetKey], rolls the key and once again stores it in SlotConfig[TargetKey]. This is depicted in the following figure. Hence when rolling a key, SlotConfig[TargetKey].WriteKey does not need to be equal to TargetKey; This means that DeriveKey command saves the rolled key on the previous key slot by default! However, **it is best that** **SlotConfig[TargetKey].WriteKey for the slot used in roll-key operation points to the same slot (that is SlotConfig[TargetKey].WriteKey = TargetKey).** This allows the MAC check for the DeriveKey command to be performed based on the current key value.

****

**Fig. . Operation mechanism of DeriveKey command. Source key denotes either parent key (create key) or target key (roll-key).**

* **Sinlge use keys:**

For the SlotID values corresponding to slots 0 thru 7 in the data section of the EEPROM, repeated usage of the key stored in the slot can be strictly limited. This feature is enabled if the SingleUse bit is set in the SlotConfig field. The SingleUse bit is ignored for slots 8 thru 14. The number of remaining uses is stored as a bit map in the UseFlag byte corresponding to the slot in question. Single use keys are not discussed in this project. For a detailed discussion refer to section 13.3.4 [ATSHADATASHEET].

* **Limited use keys**

This mechanism allows the key stored in slot 15 of DZ to be used only 128 times. The limited use key of ATSHA is not discussed in this project. For a detailed discussion refer to section 13.3.5 [ATSHADATASHEET].

Note that the main difference between limited use keys and single use key is that single use key can only be used 128 times and after that the key is disabled. Limited use keys can however be reset and again used 8 times.

Another point to discuss about DeriveKey command is authorization using MAC. This authorization only allows a valid user to create new keys, and no unauthorized updating of keys would be allowed. This MAC is created based on the fact that only the host and client are aware of the secret key. How this MAC is created is discussed in the following section. The point here is that if the MAC is incorrect, key update operation would not be performed.

#### DeriveKey command in ATSHA

The following is from [ATSHADATASHEET]. The device combines the current value of a key with the Nonce stored in TempKey using SHA-256, and places the result into the target key slot. SlotConfig[TargetKey].Bit 13 **must** be set or DeriveKey will return an error.

If SlotConfig[TargetKey].Bit12 is zero, the source key that will be combined with TempKey is the target key specified in the command line (Roll-Key operation). If SlotConfig[TargetKey].Bit12 is one, the source key is the parent key of the target key, which is found in SlotConfig[TargetKey].WriteKey (Create Key operation). Hence when trying to create a key, setting the TargetKey parameter of DeriveKey command causes the following:

* *SlotConfig[TargetKey].WriteKey* is chosen is parent key.
* The key that is created is stored in *SlotConfig[TargetKey].*

Prior to execution of the DeriveKeycommand, the Nonce command must have been run to create a valid nonce in TempKey. Depending upon the state of bit two of the input mode, this nonce would have been created with the internal RNG, or it would have been fixed.

If SlotConfig[TargetKey].Bit15 is set, an input MAC must be present that is computed as follows:

|  |  |
| --- | --- |
| SHA256(ParentKey, Opcode, Param1, Param2, SN[8], SN[0:1]) | (‑) |

where the ParentKey ID is always SlotConfig[TargetKey].WriteKey. This means that when computing this MAC on the ATSHA side, ParentKey value is always chosen from SlotConfig[TargetKey].WriteKey. Hence, for the roll key operation (as mentioned above), it is best that SlotConfig[TargetKey].WriteKey = TargetKey. Moreover, for create key operation only the parent key is used for MAC authentication.

**Note:** If the source and target keys are the same, there is a risk of permanent loss of the key value if power is interrupted during the Write operation. If the configuration bits permit it, then the key slot may be recovered using an authenticated and encrypted write based upon the parent key.

For slots 0 thru 7 only, if input parsing and the optional MAC check succeed, UseFlag[TargetKey] gets set to 0xFF and UpdateCount[TargetKey] is incremented. If UpdateCount currently has a value of 255, then it wraps to zero. If the command fails for any reason, these bytes will not be updated. The value of UpdateCount may be corrupted if power is interrupted during the execution of DeriveKey.

If SlotConfig[TargetKey].Bit12 (roll-key or create key bit) or SlotConfig[TargetKey].Bit15 (Mac bit) is set and SlotConfig[ParentKey].SingleUse is also set, DeriveKey returns an error if UseFlag[ParentKey] is 0x00. DeriveKey ignores SingleUse and UseFlag for the target key if SlotConfig[TargetKey].Bit12 and SlotConfig[TargetKey].Bit15 are both zero.

Table ??? shows the input parameters to this command. Moreover, Table ??? demonstrates the output values of the command.

Table . DeriveKey command input parameters

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Size | Note |
| Opcode | DERIVEKEY | 1 | 0x1C. |
| Param1 | Random | 1 | Bit 2: The value of this bit must match the value in TempKey.SourceFlag  Bit 0:1 and 3:7 Must be zero. |
| Param2 | TargetKey | 2 | Key slot to be written. |
| Data | Mac | 0 or 32 | Optional MAC used to validate operation. |

Table . DeriveKey command output parameters

|  |  |  |
| --- | --- | --- |
| Name | Size | Note |
| RandOut | 1 | Upon successful completion, the ATSHA204A returns a value of zero. |

* **Use case example:**

As discussed before, we intend to use a parent key in this project to generate a child key for every session of authentication. Both the parent and child keys have read and write restrictions, meaning they cannot be read or written; hence, they can be accessed only via DeriveKey command. Assume that slot1 is chosen as parent and slot2 as child. Moreover, assume that no MAC authentication is considered. Therefore, SlotConfig0 and SlotConfig1 in CZ must be set as follows:

Table . Suggested SlotConfig value for slot1 and slot2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Bit |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | SlotConfig1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | SlotConfig2 |

The reason why we chose slot2 for child key is that now CheckMac command copies the boot secret in Slot3 (refer to the section on CheckMac command).

Now assume that a Nonce (random number type) has been uploaded in TempKey, DeriveKey must be run as follows to generate the child key in Slot2:

Table . DeriveKey parameters

|  |  |
| --- | --- |
|  | Value |
| Opcode | 0x1C. |
| Param1 | 0x00 (Bit 2 must be 0 to match TempKey.Sourceflag = Rand). |
| Param2 | 0x0002 |

#### DeriveKey command software implementation

|  |  |
| --- | --- |
| **Function name** | sha204h\_derive\_key() |
| **Input** | struct \*sha204h\_derive\_key\_in\_out |
| **Output** | A uint8\_t denoting command execution status.  The created key is returned via the pointer \*target\_key. |

Sha204h\_derive\_key() is the software realization of DeriveKey command in the sense that it accepts a parent key and either rolls it or creates a new key that is stored as child key.

This function accepts a pointer to a structure of form sha204h\_ derive\_key\_in\_out as input. The fields of this structure are given in the following figure. Note that \*rand\_out is the pointer to an array that holds the RNG generated by ATSHA chip.

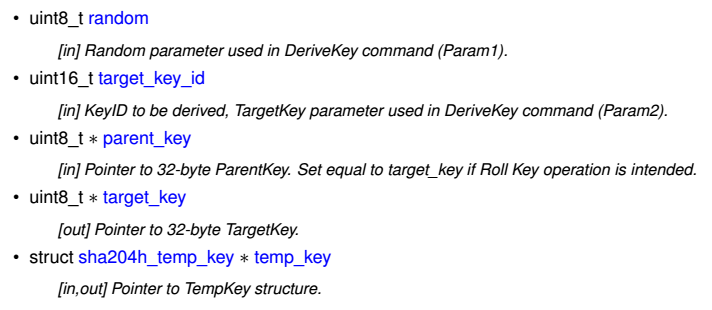


Fig. sha204h\_derive\_key\_in\_out data structure fields

Note that \*parent\_key pointer points to the key that will be used in either roll key or create key operation. Moreover, \*target\_key pointer points to the outcome of DeriveKey operation. As always, a pointer to temp\_key is required for this command to take place.

The function performs two separate actions (see the commented implementation):

1. **Check input parameters:** The function checks whether or not pointers have been assigned to sha204h\_ derive\_key\_in\_out fields. Moreover, it checks whether Param1 and Param2 of the command are chosen properly, else the function returns a SHA204\_BAD\_PARAM flag.

The function also checks whether Tempkey register parameters are valid for the command. That is TempKey.checkflag = 0, TempKey.Valid = 1, and that the random flag in Param1 equals TempKey.Sourceflag. If one of the above conditions are incorrect, the code returns SHA204\_CMD\_FAIL flag.

1. **Computation:** Based on the chosen mode, the function forms the array depicted in Fig. (the data array of command). After forming this array, SHA256 is calculated and the result is returned via target\_key pointer. Finally, Temkey.valid is set to zero and a SHA204\_SUCCESS is returned.

Note that no MAC checking is performed in this implementation.

### CheckMac command

The CheckMac command is the command that is used by ATSHA to perform authentication, that is, calculate the MAC based on the input challenge and compare it with an input MAC. When providing data for computation of MAC, this commands adds exactly the same auxiliary data as the one used by MAC command. This is illustrated in Fig. . As the figure demonstrates, data used for CheckMac command MAC calculation are:

* Data provided by the host: The black boxes in the figure.
* Data provided by client (ATSHA chip or emulating software): The red boxes in the figure.

Note that it is obvious that the slot corresponding to key should be provided by the ATSHA. Also note that SN[8] and SN[0:1] are the same in all ATSHA chips. The only other data that the client uses which is not provided by the host is OTP[0:7]. This point demonstrates the importance of data stored in this zone, as it provides an information that can used to generate a MAC that prevents the attack discussed in the previous chapter (that is, the attack in which the same program digest is sent to ATSHA chip).

In what follows, first we discuss the hardware implementation of this command on ATSHA, then briefly discuss the emulating software implementation.

**Note:** CheckMac command can also be used for password checking, although this is not discussed in this project.

**Note:** CheckMacConfig byte of plays a crucial role in implementation of CopyMode of CheckMac command. See the CZ section. Note that based on the definition of this byte, only slots 0 to 7 can be used for the copy mode of ATSHA command.



Fig. Arrangement of data passed to SHA256 in CheckMac command

A main issue of the CheckMac command is that it only produces an ACK/NACK after comparing the MACs. Hence, an adversary can forge this ACK and the MCU may never find this out. To overcome this problem, an extra feature of CheckMac is useful. Using copy mode, ATSHA copies only one of the (odd numbered) slots of DZ to TempKey. The content of this slot is referred to as boot secret. This boot secret is read from ATSHA by a challenge (most commonly using a MAC command) and is compared with the same secret at the MCU. Therefore, the MCU can ascertain whether the target ATSHA has sent the ACK. Copy mode is discussed in further detail below.

#### CheckMac command in ATSHA

The following is from [ATSHADATASHEET]. The CheckMaccommand calculates a MAC response that had been generated on a Crypto Authentication device (or perhaps emulating software as in our implementation) and compares the MAC response with some input value. It returns a Boolean result to indicate the success or failure of the comparison.

Prior to running this command, the Nonce and/or GenDig commands may have been optionally run to create and load a key or nonce value in TempKey. The mode parameter determines the source of the “key” (the first 32-bytes of the SHA message) and “challenge/nonce” (the second-32 bytes of the SHA message).

Mode[2] controls the requirement for a random nonce if TempKey is part of the computed value. If the bit is one, then TempKey must be generated using Nonce(Fixed); if it is zero, then TempKey must be generated using Nonce(Random).

**Note:** Setting the bit to one may enable replay attacks in some situations.

If the comparison matches, then the target slot value may be copied into TempKey (copymode of CheckMac command). This process is depicted in Fig. ????. If SlotID is even, then the target slot is SlotD+1, or else the target slot is SlotID. For the copy to take place, the following conditions must be true. If they are not all true, then the ATSHA204A will return the comparison result but not copy the key value:

1. The mode parameter to CheckMac must have a value of 0x01or 0x05.
2. SlotConfig.ReadKey for the target key must be zero.
3. The bit in Config.CheckMacSource corresponding to the key slots must have a value that matches Mode[2].

**Note:** Based on the above discussion about the target key of CheckMac command and also the definition of CheckMacConfig in CZ, only odd slots of DZ can be the target of copy mode.



Fig. Copy mode of CheckMac command

Table ??? shows the input parameters to this command. Moreover, Table ??? demonstrates the output values of the command.

Table . CheckMac command Input Parameters

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Size | Note |
| Opcode | CHECKMAC | 1 | 0x28. |
| Param1 | Mode | 1 | Bit 0: Source of the **second** 32-bytes of the SHA message.  0: ClientChal parameter  1: TempKey  Bit 1: Source of the **first** 32-bytes of the SHA message.  0: Slot[SlotID]  1: TempKey  Bit 2: If TempKey is used, this bit must match the value of TempKey.SourceFlag.  Bit 3-4: Must be zero.  Bit 5: 8-bytes of SHA message.  0: zeros  1: OTP zone  Bits 6-7: Must be zero. |
| Param2 | SlotID | 2 | Which internal key is to be used to generate the response. Bits 0:3 only are used to select a slot. |
| Data1 | ClientChal | 32 | Challenge sent to Client. (Must appear in the input stream). |
| Data2 | ClientResp | 32 | Response generated by the Client (*i.e, the MAC corresponding to this challenge*). |
| Data3 | OtherData | 13 | Remaining constant data needed for response calculation (otherdata[] in Fig. (above figure)???? and the following table). |

Table . Output parameter of CheckMac command

|  |  |  |
| --- | --- | --- |
| Name | Size | Note |
| SUCCESS | 1 | Upon successful completion, ATSHA204A returns a value of zero |

To clarify the content of otherdata[], the following table is used:

Table. The content of OtherData array.

|  |  |  |  |
| --- | --- | --- | --- |
| Size | CheckMac | MAC | Note |
| 1 | OtherData[0] | OpCode | MAC OpCode = 0x08 |
| 1 | OtherData[1] | Mode | Mode used for MACcommand. |
| 2 | OtherData[2:3] | SlotID | SlotID used for MACcommand. |
| 3 | OtherData[4:6] | OTP[8:10] | OTP[8:10] used for MACcommand. (Useful for Legacy.) |
| 4 | OtherData[7:10] | SN[4:7] | SN[4:7] used for MACcommand. (Unique per Client.) |
| 2 | OtherData[11:12] | SN[2:3] | SN[2:3] used for MACcommand. (Unique per Client.) |

* **Use case example (authentication with copy mode – Approach I):**

Assume that DeriveKey command has been run to produce a child key in slot2 as mentioned in the corresponding section. Further assume that the host loads its current digest into TempKey register using Nonce command pass-through mode (mode = 3). The mode parameter for CheckMac is chosen as follows:

Table. Mode parameter of CheckMac command

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2(1) | 3 | 4 | 5 | 6 | 7 | Bit |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Mode |
| Note 1: Given that the pass mode of Nonce command is used, source flag is 1 (input). | | | | | | | | |

Further, SlotID is set to 0x0002. Despite uploading the digest into TempKey register, it is also written in ClientChal input of this command (because challenge must appear in the input). Moreover, the computed MAC on the host side is also uploaded to ClientResp. OtherData array is also filled as desired (which must include ATSHA serial number for personalization). After execution of CheckMac command and upon success, ATSHA copies the content of slot3 into TempKey register. Note that the CheckMac bit for slot3 in CheckMacConfig of CZ must match bit two of CheckMac mode parameter. This implies that for this implementation, CheckMacConfig = 0x08. **Note also that at no point should the device be put to idle mode, or the copy mode content copied to TempKey would disappear.**

Now to check the CopyMode secret that is loaded in TempKey, first a random number is generated **by the host[[2]](#footnote-2)**. Then using the MAC command, the value in TempKey is combined with this challenge to produce a MAC. This MAC is then sent from ATSHA to host, where it is compared with the same MAC produced on the host side (Note that the boot secret is very much like the secret key, that is both the host and client are aware of it). If these two MACs are equal, then the host is verified.

#### CheckMac command software implementation

The software implementation of CheckMac command follows the same approach as those of the previous commands such as MAC and etc. This means that first the necessary conditions for execution of the command are checked. Next, the input array to the SHA256 engine is parsed, and finally the corresponding SHA256 is calculated. Note that this implementation of CheckMac command **does not** compare the constructed MAC with the input MAC. This comparison needs to be implemented separately.

Given that the software implementation of this command is not used in this project, it is not discussed any further.



# Algorithm and Abstract Implementation

## Introduction

In this chapter, we discuss the authentication algorithm we are going to implement in more detail, using the functions of ATSHA chip. As with phase I, the authentication algorithm has two aspects:

1. Pre-programming the ATSHA chip (i.e., DZ and CZ configuration).
2. Algorithm (software) implementation in the host (microcontroller).

In the following sections, we first discuss how to configure CZ and DZ and then proceed to discuss abstract software implementation using state diagrams.

## CZ and DZ configuration

In order to perform an authentication session which employs a child key and also involves the copy mode of ATSHA, we assign three slots of DZ: one to the parent key, one to child key and another one to boot secret (diversified parent key). Note also that child key must be stored in an even slot, so that the boot secret is read from the odd slot right after this slot. Based on this assumption:

* Slot 0x01 (Slot1) of DZ holds the parent key.
* Slot 0x02 (Slot2) of DZ holds the child key of each session.
* Slot 0x03 (Slot3) of DZ holds the diversified key.

The other slots in DZ are of no relevance to this project. Note that using the above configuration, copymode of ATSHA copies slot3 into TempKey after authentication. In order to use each of these slots, we must configure *SlotConfiguration1* to *SlotConfiguration3* in CZ. Moreover, we must configure *CheckMacConfig* byte of CZ for Slot3.

* ***SlotConfiguration***

The most important feature of the two slots must be that their read and write should be prohibited. Moreover, Slot1 must be the parent of slot2. Therefore, the following table summarizes the value of *SlotConfig1* to *SlotConfig3.* As a precaution, the parent of Slot3 is set to Slot3 itself.

Table . The content of SlotConfig bytes in CZ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Bit |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | **SlotConfig1** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | **SlotConfig2** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | **SlotConfig3** |

* ***CheckMacConfig***

Given that we use the fixed mode of Nonce to upload the current digest into TempKey and since we perform the copy mode of CheckMac using slot3, we must set  
*CheckMacConfig = 0x02.* Therefore, bit one in this byte will match TempKey.SourceFlag (or bit two of CheckMac command) which is one because of using pass through mode.

**A final note**: configuration of CZ and storing the parent key in DZ slot1 and generating and storing the diversified key in slot3 is performed using ATSHA programmer. See appendix II.

## Authentication protocol state diagram

Based on Fig. ??? (the algorithm figure in chapter one), the algorithms discussed in this section only refers to the authentication session. The discussion on how OPD is calculated and programmed together with the secret key in the host and how ATSHA is programmed using the programmer is given in Appendix II. When saying a state is performed in host/ client, we mean that the corresponding operation is performed in each party. Note however that at all times, the host is responsible for sending the right command to client in order for a command to take place.



Fig. . Authentication protocol state diagram

As the figure demonstrates, the entire authentication protocol is broken into an authentication session and a preparation state:

* **Preparation state:** The preparation state takes place before the authentication session. This state is responsible for computing CPD in MCU. As long as this digest is not computed, no authentication can take place.
* **Authentication session states:** After the completion of preparation state, authentication session states take place as follows:
  + **Create child key:** A child key is created on both the host and client. Moreover, ROPD is computed as well as RCPD. Note that the **same** Nonce is used for generating all three parameters to reduce authentication time.
  + **Compute signature:** The program signature is created in the host using child key and ROPD.
  + **Authentication:** Authentication is performed in the client by comparing the signature with the computed MAC in the client using child key and RCPD. The host determines the success or failure of algorithm based on the flag received from the client.
  + **Check ACK authenticity:** The authenticity of the ACK (if received from the client) is verified using the copy mode of CheckMac command.

Note that initialization and finalization are intermediate states which denote the beginning and the end of authentication process. It is clear from the above figure that one main task of finalization state is to declare the outcome of authentication to the outside of protocol.

In what follows, we discuss the detail of how each state is implemented. But first, we must discuss the implementation approach used for implementing the authentication algorithm.

### Implementation approach

The implementation approach considered for implementing the authentication protocol in this project follows the same logic as the one implemented in the previous phase. All authentication session states must be implemented in PL1. However, the preparation state is implemented in the **asynchronous state of the main program of MCU**. This allows the protocol to compute CPD without imposing a significant computational burden on the MCU.



Fig. Implementation logic of the authentication protocol (See phase I report).

Besides implementing PL1 in this report, we must decide on the placement of PL1 and PL2 in the main program of MCU. The preferred approach is method I as discussed in section 3.3.2 of the report of phase I. This implies that:

* PL2 is implemented in the asynchronous loop of MCU main program,
* PL1 is implemented in the synchronous loop of MCU main program,
* Two auxiliary states named “AUTHENTICATION STATE” and “BREACH STATE” and a failure threshold “AUTHENTICATION\_FAILURE\_THRESHOLD” must be defined in the main program of MCU.

Using the above implementation, the MCU:

* must wait until CPD is computed in order to perform authentication.
* performs authentication only if allowed by the main program.
* leaves authentication session if needed, and resumes the protocol the next time it is asked to perform authentication (if necessary).
* Invalidates current program if the number of failures exceeds AUTHENTICATION\_FAILURE\_THRESHOLD.
* is not allowed to leave the authentication session when checking the authenticity of MAC (copy mode of CheckMac command).

In this phase, PL1 sends out the following flags:

* **PL1\_BUSY:** PL1 is busy performing authentication steps.
* **PL1\_SUCCESSFUL:** PL1 has finished successfully, meaning that authentication has been successful.
* **PL1\_FAILURE:** PL1 has failed, meaning that authentication has not been successful.
* **PL1\_CHECK\_ACK\_AUTHENTICITY:** PL1 is in copy mode of CheckMac command, and is checking ACK authenticity (when PL1 output flag is PL1\_CHECK\_ACK\_AUTHENTICITY, we are not allowed to leave the authentication algorithm).

The flow chart of the program implemented based on the above assumptions is given in Fig. ???. Note that the MCU enters the breach state in the same manner as the one discussed in phase I. Moreover, the function implementing CPD computation returns a flag “CPD\_COMPUTATION\_FINISHED” denoting end of digest computation. Therefore, we are not allowed to enter the “AUTHENTICATION STATE” from hypothetical “NORMAL PROGRAM STATE” unless this flag is set. An implementation of the above discussion in C language is given in Fig.????.

One thing that should be pointed out based on the proposed implementation is that PL1 is solely responsible for determining the outcome of authentication process; whatever action that needs to be performed based on the result of authentication is left to MCU main program (in BREACH state, as discussed in the following sections).



Fig. Flow chart of MCU program assuming the implementation of authentication protocol.

Fig. A sample implementation of PL1 in the synchronous loop by employing an auxiliary “AUTHENTICATION STATE” defined in the main program

|  |
| --- |
| switch (current\_state) // current\_state variable denotes the states of the main program.  {  case (NORMAL\_PROGRAM\_STATES) // Assume that we want to enter normal program states.  if (CPD\_claculation\_outputflag == CPD\_COMPUTATION\_FINISHED)  {  current\_state = AUTHENTICATION\_STATE; // We are allowed to perform authentication only If CPD calculation is   // finished  }  else  {  current\_state = NORMAL\_PROGRAM\_STATES;  }  case (AUTHENTICATION\_STATE): // The state in which authentication function is called.  PL1\_output\_flag = ATSHA\_Authentication\_PL1(); // Call the authentication protocol level one function.  if (PL1\_output\_flag == PL1\_CHECK\_ACK\_AUTHENTICITY)  {  next\_state = AUTHENTICATION\_STATE;  }  else if (PL1\_output\_flag == PL1\_BUSY)  {  if (normal\_program\_flag) // If a flag shows that we need to continue with normal program states  {  next\_state = ...; // Proceed to normal program states.  }  else  {  next\_state = NORMAL\_PROGRAM\_STATES; // Proceed to normal program states.  }  }  else if (PL1\_output\_flag == PL1\_FINISHED)  {  authentication\_failure\_counter = 0;  next\_state = NORMAL\_PROGRAM\_STATES; // Proceed to normal program states.  }  else if (PL1\_output\_flag == PL1\_FAILURE)  {  authentication\_failure\_counter++;  if (authentication\_failure\_counter > AUTHENTICATION\_FAILURE\_THRESHOLD) // If authentication failure  // counter exceeds a threshold  {  next\_state = BREACH\_STATE; // There has been a breach, so stop the normal function of program.  }  else  {  next\_state = AUTHENTICATION\_STATE;  }  }  else // In case the output flag of PL1 is unknown  {  next\_state = AUTHENTICATION\_STATE;  }  break;    case (BREACH\_STATE): // The state in which the necessary tasks are performed to stop divulging the information due to   // an attack .  if (authentication\_failure\_counter > AUTHENTICATION\_FAILURE\_THRESHOLD) // Check to see if failure  // counter is actually above  // the threshold so that we  // are not in this state by accident.  {  ... // Do the necessary tasks to stop the normal function of the software.  next\_state = BREACH\_STATE;  }  else // Else we need to exit this state, because the program is here due to a mistake.  {  next\_state = AUTHENTICATION\_STATE;  }  break;    } |

### Preparation State (CPD computation)

As pointed out earlier, an authentication session takes place only when CPD has been computed. CPD computation is performed iteratively as discussed in section ???. After computation of CPD, the corresponding function (namely ????? in section), sends out an output flag named CPDOutputFlag. Before entering the AUTHENTICATION STATE in main program, the main program ensures that this flag is CPD\_COMPUTATION\_FINISHED. Only then the main program is allowed to invoke the authentication session states. This is illustrated in the following state diagram.



Fig. A hypothetical state diagram of authentication protocol preparation state.

### PL1 Initialization State

This is the first state of PL1. In this state, the output flag of PL1 is set to PL1\_BUSY. Moreover, the command buffer of PL1 is emptied to ensure that no incorrect command is sent to the ATSHA chip. The algorithm then transits to key generation states.

### Create Child Key, RCPD and ROPD States

The purpose of these states is to create a child key from parent key in host and client using a nonce, and then compute RCPD from CPD and ROPD from OPD, respectively, using the same nonce. This state is broken into three major states:

* **GENERATE NONCE:** The states in which a nonce is produced in client.
* **GENERATE CHILD KEY:** The states in which a child key is generated in the host and client using none.
* **GENERATE RCPD\_ROPD:** The states in which RCPD and ROPD are generated in the client using nonce.

A top to bottom implementation of these states, the state diagram of states comprising this state together with a pseudo code describing the function of each state is given in Fig. ????. Before discussing each sub-state, recall from phase I implementation that PL2 BUSY is a state in PL1 where a command is sent to ATSHA. This state invokes PL2 implementation function (ATSHA\_PL2()). Hence, for sending a command to ATSHA, PL1 transits to this state and waits until PL2 finishes sending the command and reading the response from ATSHA. Then it can access the emulated command response buffer of PL2 for reading the response from ATSHA. Moreover, we need to determine what happens to ATSHA after it is woken up and before WD expires; either it is sent to sleep or idle mode or kept awake. In the remaining sections, we determine the situation when entering PL2\_BUSY state by three key words for each scenario: SLEEP, IDLE, AWAKE. These are written before entering PL2 BUSY state in the state diagrams and are written in the pseudo codes.



Fig. State diagram and pseudo code for generating nonce, RCPD and ROPD states.

As is clear, each one of the major states constitute a number of sub-states. A transition between these sub-states as shown in the figure accomplishes the required tasks. Considering the above mentioned points, each sub-state is discussed below:

* **GENERATE NONCE:**
  + **GENERATE NONCE:** A 20 byte random number is generated in host using data such as current tick time and other random parameters.
  + **GENERATE NONCE+1:** Nonce command is sent to ATSHA. The resulting random number would be read from ATSHA buffer in PL2. ATSHA is sent to IDLE mode to keep TempKey content. Nonce is stored in the host.
  + **GENERATE NONCE+2:** Nonce is put in a register that emulates TempKey in the host. TempKey.Source is set to zero, given that Nonce command produces a random number; also set TempKey.CheckFlag = 0. Moreover, a copy of current nonce is kept, which can be used later on to generate a new nonce for copy mode of ATSHA.
* **GENERATE CHILD KEY:**
  + **GENERATE CHILD KEY:** DeriveKey command is sent to ATSHA. The child key is generated and stored in Slot2 of DZ.Device is sent to SLEEP.
  + **GENERATE CHILD KEY+1:** The encrypted parent key which is stored in host flash memory is read and decrypted.
  + **GENERATE CHILD KEY+2:** The decrypted parent key together with nonce is fed to DeriveKey emulating software to generate the child key in host.
* **GENERATE RCPD\_ROPD:**
  + **GENERATE RCPD\_ROPD:** The generated CPD is combined with the current nonce using MAC command to produce RCPD.
  + **GENERATE RCPD\_ROPD+1:** OPD is read and decrypted from flash of host.
  + **GENERATE RCPD\_ROPD+2:** ROPD is produced in the same manner as is RCPD.

### Compute Signature States

The purpose of these states is to create a signature (or MAC) using ROPD, child key and ATSHA serial number. The state diagram together with a pseudo code describing the function of each state is given in Fig. ????.



Fig. State diagram and pseudo code for generating program signature (or MAC) states.

As stated earlier, we need to include ATSHA serial number in computation of MAC in order to make the authentication algorithm personal. Considering this point, each sub-state is discussed below:

* **COMPUTE SIGNATURE:**
  + **COMPUTE SIGNATURE:** ROPD is loaded in TempKey register of host software. Given that this equals to using this register for fixed data storage, TempKey.SourceFlag is set one and TempKey.CheckFlag is set to zero.
  + **COMPUTE SIGNATURE+1:** Slot zero of DZ (which contains ATSHA serial number) is read and serial number bytes are extracted. Client is sent to SLEEP.
  + **COMPUTE SIGNATURE+2:** Program signature is generated using MAC command. Mode parameter is set to 45, given that the first slot is filled with key[slotID], second slot is with TempKey, serial number should be included and that TempKey.SourceFlag was set to one in COMPUTE SIGNATURE state.

### Authentication States

The purpose of these states is to authenticate the current running program. The state diagram together with a pseudo code describing the function of each state is given in Fig. ????.



Fig. State diagram and pseudo code for authentication states

In order to perform authentication, we use the CheckMac command of ATSHA. Authentication must be performed in such a way that we could also use the copy mode of CheckMac command. Considering the above mentioned points, each sub-state is discussed below:

* **AUTHENTICATION**
  + **AUTHENTICATION:** RCPD is loaded into the TempKey of ATSHA so that we use mode 0x05 of CheckMac command. ATSHA is kept IDLE.
  + **AUTHENTICATION+1:** A CheckMac command is sent to ATSHA. The target slot for copy mode (slot3) is also given to the algorithm. ATSHA is kept AWAKE so that the boot secret does not disappear from TempKey.
  + **AUTHENTICATION+2:** The response from ATSHA is read from PL2 (recall that this response has 4 bytes, the second byte of which is ACK/NACK). If response is ACK, the algorithms transits to CHECK AUTHENTICTY state, else the algorithm refers to AUTHENTICATION FAILURE state. If authentication fails, the internal flag of PL1 determining success or failure of algorithm is set to failure (still a busy flag is returned by PL1). This internal flag is **only** returned in PL1 FINALIZATION STATE.

### Check ACK Authenticity States

The purpose of these states is to use the copy mode of ATSHA to determine whether the received ACK is authentic. The state diagram together with a pseudo code describing the function of each state is given in Fig. ????.



Fig. State diagram and pseudo code for check ACK authenticity states

As mentioned earlier, the secret in the target slot of CheckMAC command (slot3 in this implementation) is copied into TempKey register after successful execution of the command. At this point we need to create a MAC to extract this secret from ATSHA. This MAC is created as depicted in the following figure:



Fig. State diagram and pseudo code for authentication states

The creation of this MAC implies Mode = 0x46. Also given that no slot of DZ is used, SlotID could optionally be set to 0x00. Considering the above mentioned point, each sub-state is discussed below:

* **CHECK ACK AUTHENTICTY**
  + **CHECK ACK AUTHENTICTY:** A random number for creation of boot secret MAC is generated and stored. For this end, we use the nonce generated by ATSHA in GENERATE NONCE+1 state.
  + **CHECK ACK AUTHENTICTY+1:** A challenge is sent to ATSHA for reading the content of TempKey. For this end, Mode parameter is set to 0x46. Note that by doing so, the first 32 bytes of MAC command are filled with TempKey value and the second with input challenge (random number generated by host). This choice also means that ATSHA serial number is once again used for creation of MAC. Note also that SlotID = 0x00 is optionally chosen and that TempKey.SourceFlag is one when copy mode loads the data slot into TempKey.
  + **CHECK ACK AUTHENTICTY+2:** The MAC received from ATSHA is stored in host. A read command is sent to ATSHA to read CZ0, whose data will be used to generate the (diversified) boot key in the host.
  + **CHECK ACK AUTHENTICTY+3:** Diversified boot key is generated in the host. For this end, the same process performed by the programmer is repeated’ TempKey is loaded with parent key and the corresponding TempKey registers are set. The key is generated using the emulating MAC command software. Note that setting   
    SlotID = 0x0000 is once again optional.
  + **CHECK ACK AUTHENTICTY+4:** BootMAC is generated in the host using MAC command with all inputs and parameters the same as CHECK ACK AUTHENTICITY+1 state.
  + **CHECK ACK AUTHENTICTY+5:** The MAC produced in the previous section is compared with the MAC generated by ATSHA. If comparison is successful, the internal flag of PL1 determining success or failure of algorithmis set to success and the algorithm transits to PL1 FINALIZATION STATE.

### PL1 Finalization State

PL1 FINALIZATION state is the last state visited in the authentication protocol. The state diagram together with a pseudo code describing the function of each state is given in Fig. ????.



Fig. State diagram and pseudo code for finalization state.

Also in this state, CPD computation flag must be invalidated. This allows the preparation state to begin calculating CPD.

Perhaps the most important task performed in the finalization state is to send out the success/failure flag of PL1. As illustrated in section (section on PL1 implementation), PL1 sends out four different flags; While the state of PL1 is declared either PL1 BUSY or PL1\_CHECK\_ACK\_AUTHENTICITY in all other states, the flag sent out the finalization state determines whether the authentication has failed, or succeeded. The flag that holds the success or failure of the algorithm is assigned either in AUTHENTICATION+2 or CHECK ACK AUTHENTICITY+5. Table ???? shows the output flag assignment of PL1 in each state of PL1. Note that success/failure flag must be established in the program using a separate variable in the program.

Table. Output flag of PL1 corresponding to each state of PL1

|  |  |  |
| --- | --- | --- |
| State | Success/Failure flag | Output flag |
| AUTHENTICATION+2 and  CHECK ACK AUTHENTICITY+5 | PL1\_FAILURE/ PL1\_SUCCESSFUL | PL1\_BUSY |
| CHECK ACK AUTHENTICITY to CHECK ACK AUTHENTICITY+4 | ----- | PL1\_CHECK\_ACK\_AUTHENTICITY |
| PL1\_FINALIZATION | ----- | Authentication flag |
| Other states of PL1 | ---- | PL1\_BUSY/ |

### Other States of the Protocol

In this section, with discuss the scenarios pertaining to the improper states in the protocol. These states are time-out state, unknown state and so on. Also in this section we discuss what happens of the microcontroller is reset or we have power shut-down.

* **PL2 FAILURE**

This is an isolated state of the program and only happens when PL2 denotes a failure in sending a command to ATSHA. The state diagram for this state is depicted in the following figure. As this diagram indicates, upon entering PL2 FAILURE state, the internal success failure flag is set to PL1 FAILURE, and the algorithm transits to PL1 FINALIZATION state.



Fig. State diagram for PL2 failure state

* **Unknown state**

In the event of an unknown state in PL1, the same process as that of the previous phase is performed. This means that (see Fig. ?????):

* + The algorithm makes sure that if PL2 is busy, it terminates properly.
  + If PL2 output flag is failure, a failure flag is returned and algorithm transits to PL2 FAILURE state.
  + If PL2 is ready or finished, PL1 transits to PL1 INITIALIZATION state.
  + If PL2 flag is unknown, the algorithm first determines PL2 output flag.



Fig. State diagram for the unknown state

## Impact of Unwanted MCU Reset

A main issue of the algorithm implemented in phase I was that unwanted resets could have caused problems for the algorithm. The algorithm implemented in this phase is immune to such MCU reset because there is no critical data for the algorithm that changes per reset. For example the parent key, boot secret and all other data are constant, and the only data that changes in time are the data that are created in every authentication session, such as RCPD, ROPD, child key and so on. Hence, the algorithm is immune to unwanted MCU reset.



# Software Implementation

## Introduction

A 32 byte array is defined to always keep the nonce generated either by client or host. This vector is referred to *nonce[].*

In order to increase the readability of the implementation and reduce the function overhead, we use two functions to assign command parameters:

* ***Assign\_CommandParams():*** Assigns all parameters of the command (name, word address, opcode and etc.) other than data variables.
* ***Assign\_CommandData():*** Assigns the data variables of a command.

Note that *Assign\_CommandParams()* **must always** be performed prior to *Assign\_CommandData().* This is because the latter computes the CRC bytes of the command, for which it needs the parameters of the command as well.

The TempKey structure is defined as a static for PL1, which allows constant access to the register without its content getting destroyed after each call of PL1.

* **GENERATE NONCE + 1:**

To send the nonce command to ATSHA in this state, the corresponding parameters of the Nonce command defined in *ATSHA\_Commands\_Module.h* are assigned to command buffer. Moreover, a 20 bytes of the nonce[32] generated in the previous state are assigned as Data1 of this command (see Fig. ). Note that the next state after this state is PL2\_BUSY\_CYCLE and the one after that is GENERATE NONCE + 2, which are assigned to state variables current\_PL1\_state and next\_PL1\_state, respectively, using Assign\_PL1\_States().



Fig. Nonce command and the corresponding response in GENERATE NONCE +1 state

* **GENERATE NONCE + 2:**

The content of response vector in PL2 (which is the nonce produced by ATSHA) is copied to TempKey.value as well as nonce[]. Copying this content to nonce[] allows the current nonce to be used for generation of nonce in copy mode.

* **GENERATE CHILD KEY**

To generate the child key in ATSHA, the DeriveKey command depicted in Fig. is sent to ATSHA. Note that the response to this command has the four byte template, the second byte of which is the success/failure byte whose value must be zero (success). Also note that the device is put to sleep after sending this command.



Fig. DeriveKey command and the corresponding response in GENERATE CHILD KEY state

* **GENERATE CHILD KEY+1**

The encrypted parent key is read from flash in two sub-states so as to compensate for the slow speed of reading data from flash.

* **CREATE ROPD\_RCPD and CREATE ROPD\_RCPD+2**

In order to generate RCPD and ROPD using MAC command based on Fig. ?????, the command is run with the following input parameters. Note that in order to simplify the codes, we use mode 0x45 of the MAC command. However, instead of using ATSHA serial number, we feed a vector of all zeros to the command so that it would be consistent with Fig. ?????. Moreover, the hypothetical key slot to be used for command calculation in this command is filled with CPD or OPD.



Fig. MAC command parameters used for generation of RCPD and ROPD

This is how the function should be used:

* ­It should first be called with a null pointer to both message and digest, so that the initialization state takes place.
* Each 64 byte of a (long) message should be passes to the function with null pointer to the digest.
* The final block whose length is less than 64 bytes (even a zero byte message is acceptable) is sent to the function. The output pointer to the digest now holds the pointer to digest.



# Appendix I: Data Storage in AVR CPUs

## I-1 Introduction

It is very important to remember that the discussion given in this appendix is only applicable to AVR CPUs.

The question that may arise here is that how should data be stored in flash memory and SRAM? Before this is illustrated, we briefly discuss the memory structure of AVR based on IAR C/C++ compiler reference guide [IARREFERENCE] (available in help menu of IAR software).

The AVR microcontroller is based on the Harvard architecture—thus code and data have separate memory spaces and require different access mechanisms. Code and different type of data are located in memory spaces as follows:

* The internal flash space, which is used for code, *\_\_flash* declared objects, and initializers
* The data space, which can consist of external ROM, used for constants, and RAM areas used for the stack, for registers, and for variables
* The EEPROM space, which is used for variables



# Appendix II: ATSHA Programmer

## II-1 Introduction

OPD calculation and ATSHA programmer programming!

## II-2 Implementation of ATSHA Programmer

The boot secret must be generated using MAC command (diversified key) in programmer. Add this function to programmer. It is implemented exactly like CHECK ACK AUTHENTICITY+3. Could I compute that MAC before locking CZ?

1. The year of birth of the algorithm designer! [↑](#footnote-ref-1)
2. Note that at this point TempKey holds the data in slot3. Hence, we cannot use the Nonce command of ATSHA to produce a nonce. This means that to randomize the challenge in copy mode, the host should provide the nonce! [↑](#footnote-ref-2)