Application Note 157
SHA iButton API Overview

INTRODUCTION

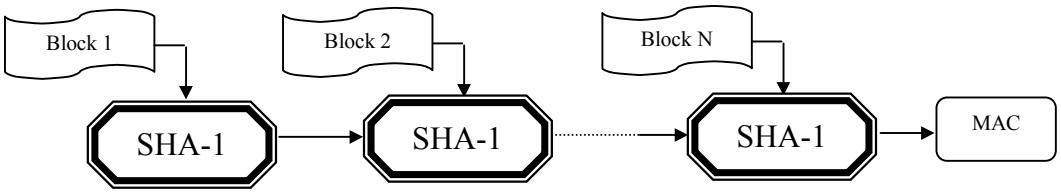
The Dallas SHA iButton® (DS1963S) is a smart token that offers many high security features and supports multiple services. This document provides an overview for implementing applications of digital identification and transactions using the SHA iButtons. A sample service implementation is used to illustrate the detailed steps required for installing the service data and conducting an electronic transaction. The usage of key API calls are discussed and the annotated listing of command and data flows for implementing each API are presented.

The SHA iButtons are suitable for a variety of applications:

- mass transportation
- electronic door locks
- building access controls
- computer and network access controls
- pay phones
- parking meters
- prepay utility meters
- vending machines
- software authorization

WHAT IS SHA?

SHA (Secured Hash Algorithm), through years of development and improvement by cryptographic experts, had been accepted as one of the best secure and widely employed hash algorithm. The SHA algorithm implemented in the SHA iButton (Dallas Semiconductor part number DS1963S) is SHA-1, which conforms to the standards specified in Federal Information Publication Standards 180-1 (FIPS 180-1). In the simplistic sense, a hash function is a process that takes an input string (called pre-image or message) and converts it to a fixed and shorter length output string (called hash value, message digest, or message authentication code). SHA is very efficient for detecting errors or changes in the input string since the inspection is made on the shorter condensed message digest. A one-way hash function has the characteristics of being easy to generate the digest but very hard to reverse compute the input message from a given digest. The security of a one-way hash function is in its strong one-way digest generation operation. Secrecy can be obtained by embedding a secret code in the input string so that no one can generate the correct digest without knowing the secret code. SHA takes input streams in blocks of 512 bits (64 bytes) and produces a 20-byte long output called message authentication code (MAC) or message digest. If the input is not 512 bits long the algorithm pads it to the nearest multiples of 512 bits. The figure below illustrates the chaining of multiple input blocks (in 512 bits) in generating the final MAC.

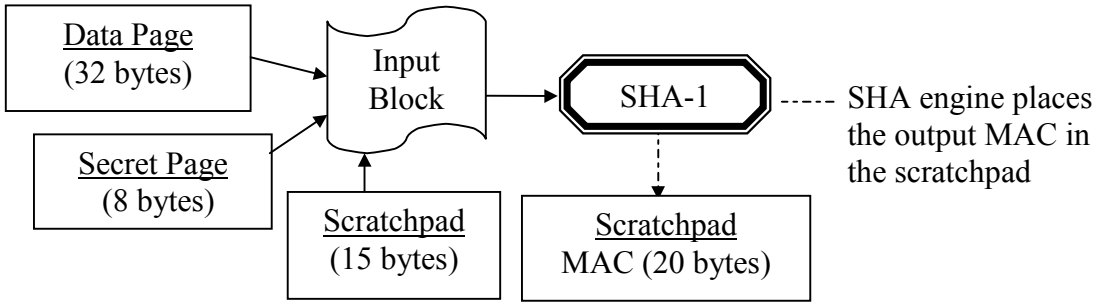


iButton is a registered trademark of Dallas Semiconductor.

SHA IN THE DALLAS SHA iBUTTON (DS1963S)

The Dallas SHA iButtons (DS1963S) implement the SHA-1 computation in a special accelerated circuitry that can complete one SHA-1 computation in under 1 ms. The input to the SHA-1 engine is limited to one block. By design, the input data block to the SHA-1 engine is made up of user data stored in a read/write data page (32 bytes), in the scratchpad (15 bytes), and a secret (8 bytes) that has been pre-installed in a no-read access secret page by the service provider (see Figure 1). The computation result (MAC) is placed in the scratchpad for subsequent operations. Depending upon the specific SHA operation called upon, this output MAC may or may not be hidden in the scratchpad from external access.

MAC GENERATION IN SHA iBUTTON Figure 1



SHA iBUTTON FEATURES

The SHA iButton is a rugged 4K-bit read/write data carrier that can be easily accessed with minimal hardware. An integrated 512-bit SHA-1 engine can be activated to compute a 160-bit message authentication codes (MAC) based on information stored in the device. Data is transferred serially via the 1-Wire® protocol, which requires only a single data contact and a ground return. A single SHA iButton can serve up to eight independent applications each with its own secret and page counter. The SHA iButton may also function as a coprocessor that safe-keeps the system secrets and assists the local transaction host in computing the MACs needed for user token authentication and application data validation. The SHA iButton, like other iButtons, has an additional memory area called the scratchpad that acts as a buffer when writing data to the device memory. The SHA iButton's scratchpad is also used for feeding data segments to the SHA-1 engine or receiving/comparing message authentication codes. When writing to an iButton, data is first written to the scratchpad from where it can be read back and verified for communication errors. After the data has been verified, a copy scratchpad command transfers the data to the target memory location. This process ensures data integrity in an environment that does not provide a reliable electrical contact.

The DS1963S SHA iButton has the following special features:

- 4096 bits of read/write nonvolatile memory organized as 16 pages of 256 bits each
- Eight of the 16 memory pages have individual 64-bit secrets and 32-bit read-only non-rolling-over page write cycle counters
- On-chip 512-bit SHA-1 engine
- As a user token the device can support up to eight independent services
- Can function as a coprocessor for storing system secrets and computing the MACs needed for authenticating user tokens and validating service data

1-Wire is a registered trademark of Dallas Semiconductor.

SHA iBUTTON MEMORY MAP

Data Memory with General Read/Write Access (16 pages, 32 bytes per page)

Page Number	Address Range (TA1 TA2)	Secret Number	Counter Number	Counter Increments
0	0000h to 001Fh	0	0	No
1	0020h to 003Fh	1	1	No
2	0040h to 005Fh	2	2	No
3	0060h to 007Fh	3	3	No
4	0080h to 009Fh	4	4	No
5	00A0h to 00BFh	5	5	No
6	00C0h to 00DFh	6	6	No
7	00E0h to 00FFh	7	7	No
8	0100h to 011Fh	0	0	With write
9	0120h to 013Fh	1	1	With write
10	0140h to 015Fh	2	2	With write
11	0160h to 017Fh	3	3	With write
12	0180h to 019Fh	4	4	With write
13	01A0h to 01BFh	5	5	With write
14	01C0h to 01DFh	6	6	With write
15	01E0h to 01FFh	7	7	With write

Secrets Memory with No Read Access (Eight 64-bit secrets)

Page Number	Address Range (TA1 TA2)	Description
16	0200h to 0207h	Secret 0
	0208h to 020Fh	Secret 1
	0210h to 0217h	Secret 2
	0218h to 021Fh	Secret 3
17	0220h to 0227h	Secret 4
	0228h to 022Fh	Secret 5
	0230h to 0237h	Secret 6
	0238h to 023Fh	Secret 7

Note that only writing to pages 8 to 15 will increment the page counters. Each counter and secret are shared by a matching pair of data pages for certain operations. For example, pages 0 and 8 share counter number 0 and secret number 0. Writing to page 8 will increase counter 0 by 1, but writing to page 0 has no effect on counter 0. Executing Read Authenticated Page command on pages 0 or 8 will both return counter 0 and the MAC computations will use secret 0.

Counter Memory with Read Access Only (Nine 32-bit counters)

Page Number	Address Range (TA1 TA2)	Description
19	0260h to 0263h	Write Cycle Counter of Page 8
	0264h to 0267h	Write Cycle Counter of Page 9
	0268h to 026Bh	Write Cycle Counter of Page 10
	026Ch to 026Fh	Write Cycle Counter of Page 11
	0270h to 0273h	Write Cycle Counter of Page 12
	0274h to 0277h	Write Cycle Counter of Page 13
	0278h to 027Bh	Write Cycle Counter of Page 14
	027Ch to 027Fh	Write Cycle Counter of Page 15
20	0280h to 0283h	Write Cycle Counter of Secret 0
	0284h to 0287h	Write Cycle Counter of Secret 1
	0288h to 028Bh	Write Cycle Counter of Secret 2
	028Ch to 028Fh	Write Cycle Counter of Secret 3
	0290h to 0293h	Write Cycle Counter of Secret 4
	0294h to 0297h	Write Cycle Counter of Secret 5
	0298h to 029Bh	Write Cycle Counter of Secret 6
	029Ch to 029Fh	Write Cycle Counter of Secret 7
21	02A0h to 02A3h	PRNG Counter (SHA engine execution counter)

USING SHA iBUTTON FOR E-PAYMENT APPLICATIONS

The SHA iButton has been designed to function as a user token as well as a coprocessor. The coprocessor would perform all the MAC computations required for device authentication and service data validation during a transaction session, thus eliminating the need for implementing SHA computation code and significantly shortens the development cycle. One of the added benefits of using a SHA iButton as the coprocessor is that the system secrets are stored inside the button and the button cannot be probed to reveal its secrets. Throughout this document we assume that a SHA iButton coprocessor is used in a local host for performing the necessary device authentication and service data validation. A few terms used in this document are defined below:

local host – a hardware unit comprised of the necessary components so it can perform electronic transactions with user SHA iButtons (or other ePurse tokens). Functionally a local host may have three key components: transaction control unit (typically a microprocessor); user interface such as a display and token receptor; and a coprocessor.

address number (AN), ROM ID, registration number, serial number – used interchangeably, refers to a factory lasered 64-bit number that guarantees the global uniqueness of the a DS1963S or any other 1-Wire devices.

transaction control unit (TCU) – a component (typically a microprocessor) that communicates between a coprocessor and the user tokens to perform device authentication and service data validation.

system authentication secret, master authentication secret – used interchangeably, refer to a secret installed in the coprocessor for the purpose of authenticating user devices.

device authentication secret – a secret installed in the user iButton so that the local host could verify if it belongs to the system. Device authentication secret is made unique to the individual user device by binding the system authentication secret with the user device's address number.

system signing secret, master signing secret – used interchangeably, refer to a secret installed in the coprocessor for the purpose of signing and validating service data.

signing signature – a message authentication code (MAC) computed from the service data, system signing secret, and other service and device specific data. The signing signature is usually embedded in the service data page so that the service data can be quickly verified by the local host.

service data, user data, application data, account data, transaction data – data that completely represents a service, such as for an access control or an e-payment application.

user device, user token – a digital carrier of service data for digital authentication or electronic payment applications. In this document the DS1963S SHA iButton is implied whenever user device or user token is referenced.

coprocessor – in the context of this document, a coprocessor is a computing unit capable of performing all the necessary MAC computations for conducting a transaction.

SERVICE INSTALLATION

To use SHA iButtons for device authentication and data validation, proper data and secrets must be installed in both the coprocessor and the user iButtons — a process often referred to as service initialization. Two secrets are installed in the coprocessor: system authentication secret and system signing secret. The system authentication secret is used to create a unique authentication secret for each user device and to verify if a user device belongs to the system. The system signing secret is used to generate the signing signature and to verify if the service data is valid. Only one secret needs to be installed in the user iButton: the unique device authentication secret. It is important to note that the user device (SHA iButton) authentication secret is made unique and different from the system authentication secret by using the user device address number as part of the input for computing the device secret.

There are two types of service data: static and dynamic. Static service data is never changed during a transaction while dynamic service data is always updated at each transaction. For example, the service data for an office or hotel access control application containing timed access privileges information needs to be validated but is not modified by the lock processor (local host). On the other hand, a vending machine or a parking meter needs to debit an appropriate amount from the account and digitally re-sign the reduced balance data so that it can be validated again by other local hosts on the system. Service data protection may not be necessary if the system only needs to know whether a user device belongs to the system (authorized user) and does not make transaction decision based on the contents of the service data

In general, an e-payment application deployment includes the following steps:
For service installation:

(1) Installing system authentication secret into a SHA iButton coprocessor.
(2) Installing system signing secret into a SHA iButton coprocessor (if data protection is needed)

For each user device:

(3) Installing user device authentication secret into the user SHA iButton
(4) Installing the signed service data into the user SHA iButton (if data protection is needed)

For conducting a transaction:

(5) Performing user device authentication using a SHA iButton coprocessor
(6) Performing user data validation using a SHA iButton coprocessor (if data protection is needed)

DEVICE AUTHENTICATION

To authenticate a user SHA iButton, the TCU would ask the coprocessor to compute a random challenge[1] and sends it over to the user iButton and ask it to compute a response MAC using the Read Authenticated Page command. The user iButton takes the challenge, the service data, its own address number, and its authentication secret to compute the response MAC. The response MAC is then read by the local host for comparison with its own computation later. To verify the response MAC, the TCU would ask the coprocessor to first re-compute the user iButton's unique device authentication secret using the user device's address number and the system authentication secret. The TCU would then ask the coprocessor to compute a MAC using the re-computed device authentication secret and the challenge code that it sent over to the user iButton. This coprocessor computed MAC is then compared with the response MAC read from the user iButton to determine if the user device is legitimate. This two-step process is necessary because the local host cannot read the user device's secret and that the user device authentication secret is different from the system authentication secret. Note that device authentication only requires the user device to carry the right authentication secret, it does not care about the service data contents.

SERVICE DATA PROTECTION

Service data protection is achieved by embedding a system generated MAC (signing signature) with the service data. The signing signature is generated using the system signing secret, service data, service data page number and the page counter value, and the user device address number. Using the signing signature one can detect any unauthorized changes and prevent replaying of the service data. To validate the service data, the TCU would ask the coprocessor to re-compute a MAC (signing signature) from the system signing secret and all the service and user device data items obtained at the time of service. This MAC is then compared with the embedded signature to determine if the data is valid. For dynamic data services, after the service data is updated a new signing signature is computed to reflect the data change and the new page counter value. This new signature is then embedded in the service data and saved to the user device.

[1] This challenge is random in the sense that it is affected by the SHA engine counter value, which increments each time the SHA engine is exercised, and that a user iButton would very unlikely get the same challenge twice. See AN152 for more information on generating challenges and secrets.

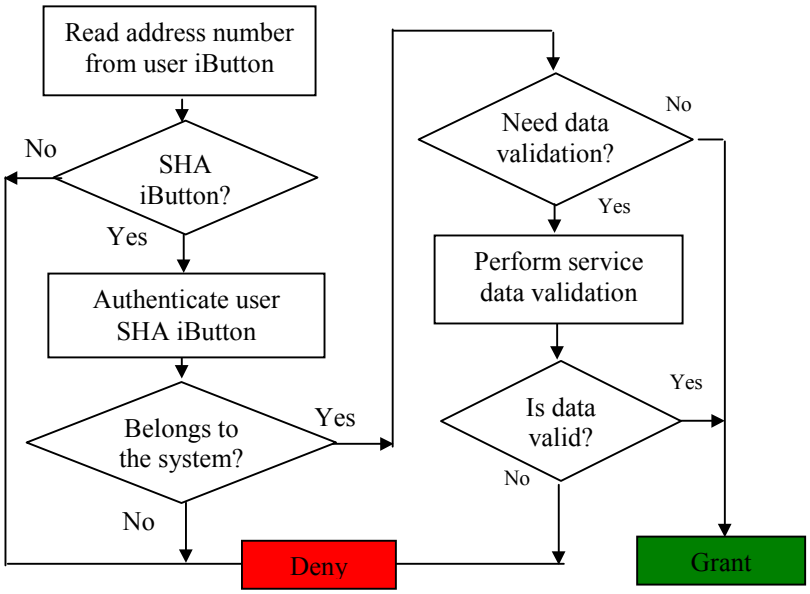
STATIC DATA SERVICES

Static data services only require the device to be authenticated and service data validated in order for a service decision to be made. Device authentication refers to the process that verifies if a user device belongs to the system. Device authentication may be the only procedure required in certain access control applications. There are variety of ways to perform device authentication: (a) reading a device's address number (AN) and searches through a database to see if it belongs to the system; (b) performing a challenge and response procedure to test if the device carries a valid secret; (c) other methods. SHA iButtons use the second method to perform device authentication. In this challenge and response method, the local host asks the user device to compute a response MAC based on its hidden authentication secret, the memory data linked to that secret, and the challenge code that the local host provides. The user device never reveals its secret to the outside world. This authentication mechanism makes the system very secure, particularly beneficial if the communication link between the local host and user device is not secure. Furthermore the local host would issue a different challenge each time a user device requests for a service so that the response MAC would be different each time, thus making intercepting the communication data bits useless.

The typical steps for performing a static data service are as follows (see Figure 2)

(1) Read address number (AN) from the user iButton
(2) If it is not a SHA iButton then branch to other decision process
(3) Authenticate the user iButton using a challenge and response sequence; quit if the user iButton does not belong to the system
(4) Does the service data need to be validated? If no, provide service.
(5) Check if the service data is valid; quit if not
(6) Provide service

STATIC DATA SERVICE DECISION TREE Figure 2

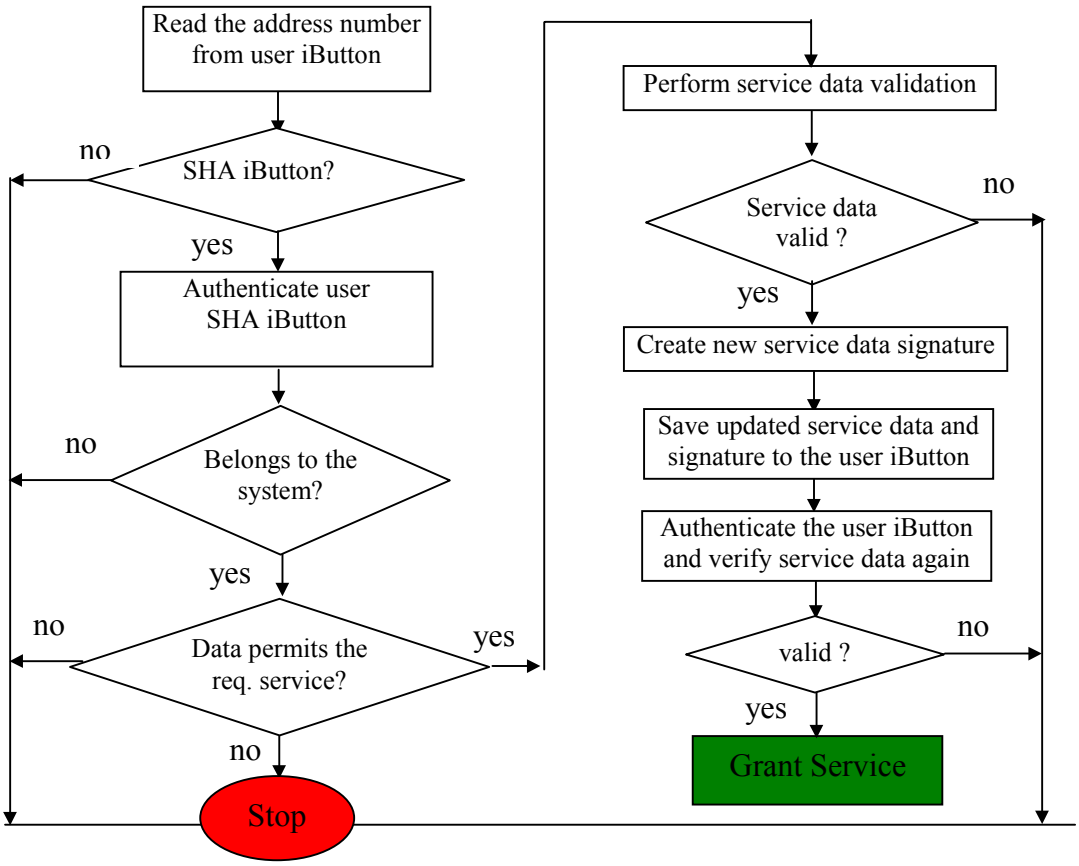


DYNAMIC DATA SERVICES

In addition to the authentication and data validation processes performed for conducting a static data service, a service using dynamic data requires additional steps to modify the service data, recreate a signature and save the re-signed data to the user iButton. The typical steps are outlined below:

(1) Read address number (AN) from the user iButton
(2) If it is not a SHA iButton then branch to other decision process
(3) Authenticate the user SHA iButton using a challenge and response sequence; quit if the user SHA iButton does not belong to the system
(4) Does service data permit the requested service, quit if not.
(5) Perform service data validation; quit if not valid
(6) Modify the service data and create a new signature
(7) Save the new service data with signature to user iButton
(8) Perform user device authentication again to ensure that the service data has been updated in the original user device
(9) Provide service

DYNAMIC DATA SERVICE DECISION TREE Figure 3



HOSTING MULTIPLE SERVICES

The 1-Wire File Structure (OWFS[2]) may be employed to facilitate the coexistence of multiple services in one user iButton — with data for each service stored in its own service file. Multiple files are organized in a directory entry table for efficient access. The local host reads the file entry for a service in the directory table to acquire the actual page location and the number of pages used for that service. The actual service data is read from the service file.

Special files may also be stored in a coprocessor and other iButtons to identify them as special devices. For example we create a file called COPR.0 in a SHA iButton to identify it as the coprocessor and store in that file all the necessary data needed for the TCU to complete a transaction. We may create another file called COLL.102 in a iButton to identify it as a mobile data collector that is authorized to retrieve data from a local host to be dumped into a central database later. When a transaction control unit (TCU) starts up it reads information from the coprocessor iButton to prepare itself ready for conducting transactions. When a TCU detects a data collection button it then performs the necessary authentication and outputs the requested data.

THE COPROCESSOR FILE

A DS1963S SHA iButton is configured to function as a coprocessor by installing the appropriate system secrets and data into it. For static data services without data validation, only the system authentication secret is installed in the coprocessor. For dynamic data services or static data services requiring data validation both system authentication secret and system signing secret must be installed in the coprocessor. The coprocessor also performs all the necessary MAC computations for authenticating the user iButton and validating the service data. In our sample implementation, a coprocessor a file (COPR.0) is created in a SHA iButton to identify it as the coprocessor. COPR.0 contains all the data needed by a transaction control unit in order to perform a transaction.

COPROCESSOR FILE DATA STRUCTURE[3] Table 1

Structure of Coprocessor File COPR.0			
	No. of Bytes	Sample Data	Remarks
Name of the service data file	5	DLSM.102	Only the basename (DLSM) and extension are stored, not the separator(.). The extension byte is stored in hex (66h=102 d)
Signing page number	1	8	Legal choices are page 0 or 8. Page 0 is used for file entry table.
Authentication page number	1	7	Starting at a high enough page number to give enough pages for this file's storage.
Workspace page number	1	9	Needed by the coprocessor for recreating user device unique authentication secret

[2] The 1-Wire File Structure (OWFS) provides a directory structure for data residing in 1-Wire devices including iButtons. It allows named files to be randomly accessed as they are on other file systems. The definitions and rules of the OWFS are sufficient to store multiple files in nested directories using device capacities up to 16M bytes. These devices may be organized as 2...65535 pages of 32...256 bytes. See Dallas Semiconductor/Maxim Application Note 114 for more information on OWFS structure and implementation support.

[3] In the sample service implementation all data bytes are written least significant bytes first

Structure of Coprocessor File COPR.0			
			during a transaction.
Version number	1	1	Version number is used by the service provider to track service configuration changes.
Installation date code	4	04 0E 00 63 h	installation code stored in M D YY format. YY=number of years since 1900. For example 4/14/1999 = 04 0E 00 63
Device authentication secret binding data	39	39 FFh bytes	Used by the service provider to bind authentication secret to a user SHA iButton. The first 32 bytes of this block is written to a data page and the remaining 7 bytes written to the scratchpad when computing the device authentication secret. The scratchpad actually contains 15 bytes that are input to the SHA engine. The other 8 bytes are the page number (1 byte) and the device address number without the CRC byte.
Service sign code	3	3 00h bytes	This data is part of the 15 bytes written to the scratchpad for creating service data signature. The other 12 bytes are page number (1 byte), page counter (4 bytes), and the device address number without the CRC (7 bytes).
Provider name length (svcNameLen)	*1*	*20*	Length of the service provider name, adjustable.
Signature length (signLen)	*1*	*20*	Length of the signature block in user account page. This value may be reduced to allow space for other data (auxiliary data)
Auxiliary data length (auxDataLen)	*1*	*0*	Length of the auxiliary data block, adjustable. A zero length provides no additional data
Name of service provider	20	Dallas Semiconductor	A service provider name or description. The length is limited to the *Provider name length* entry.
Signature initial value	*20*	20 00h bytes	Length of initial block must match the *Signature length* entry above. These bytes are set in place of the signature block in user data page for creating the data signature.
Auxiliary data	*0*		Length of auxiliary data must match the *Auxiliary data length* entry above.
Encryption algorithm code	1	0	A flag indicating the type of encryption or encoding mechanism employed for securing the contents of this file.
DS1961S flag	1	0	A boolean flag indicating that the secret installation for this coprocessor was performed with appropriate padding for

Structure of Coprocessor File COPR.0			
			use with a DS1961S.
Total number of bytes	**100**		

Note that the length of COPR.0 varies with the values of the three length parameters: Provider Name Length, Signature Length, and Auxiliary Data Length. These parameters are provided to allow for additional custom features that the software supports but need to be enabled by the data in COPR.0 file.

THE SERVICE DATA FILE

In the sample implementation, we create a service file DLSM.102 in each user SHA iButton to identify it as a user iButton. The file DLSM.102 contains the following data:

SERVICE FILE DATA STRUCTURE Table 2

Structure of Service File DLSM.102			
	No. of bytes	Sample Data	Remarks
Data type code	1	0	This field may be used to further indicate the type of transaction data. For example 0 for dynamic and 1 for static data etc.
Service data signature	20	[computed]	To be computed by the local host.
Multiplier /conversion factor	2	8B48h	This data may be used to convert one quantity to another. In the sample service we combine the international currency code for USD and 0.01 conversion factor for converting from cents to dollar. The specific format is usually service dependent.
Account balance	3	01 86 A0 h	100000 cents ($1000)
Transaction ID	2	1234h	Transaction ID may be used to link more reference data to a transaction.
Total bytes	**28**		

Note that the file content is 28 bytes long but 32 bytes are stored in the page in the following format (see AN114 for more information):

File length (1 byte)	File contents (28 bytes)	File continuation pointer (1 byte)	CRC-16 (2 bytes)

When the local host detects a user SHA iButton it reads from the directory entry table of the user SHA iButton the page number (13 in our sample service) for file DLSM.102 and performs device authentication using the secret associated with page 13 (secret number 5).

INSTALLING SERVICE DATA

Proper service data and secrets must be installed into the coprocessor and a user iButton before any transaction can be conducted with them — a process also referred to as device initialization. Device authentication is needed for both static and dynamic data services. If the service uses dynamic data (such as e-payment applications), the service data needs to be verified and updated during each transaction. The coprocessor would contain two system secrets: one for authentication and one for verifying and re-signing the service data. The steps for installing the two system secrets are identical except that the system signing secret must be installed in secret number 0, and is usually computed using different input data (partial phrases). On the user device side, protection of service data is achieved by embedding in the service data page a signature computed from the service data and the system signing secret. The basic steps for installing a service into a coprocessor and on a user SHA iButton are as follows, with the detailed description provided in the sections below.

For services requiring no data validation:

(1) Install the system authentication secret into the coprocessor
(2) Install a unique device authentication secret into the user iButton by binding the system authentication secret with the user iButton address number and the service data page number
(3) Write service data to the user iButton (if needed)

For services requiring data validation:

(1) Install the system authentication secret into the coprocessor
(2) Install the system signing secret into the coprocessor
(3) Install a unique device authentication secret into the user iButton by binding the system authentication secret with the user iButton address number and the service data page number
(4) Compute a signing signature from the system signing secret, service data, page number and write cycle counter value of the service data page, and the user device address number
(5) Embed the signing signature with service data and write it to the user iButton

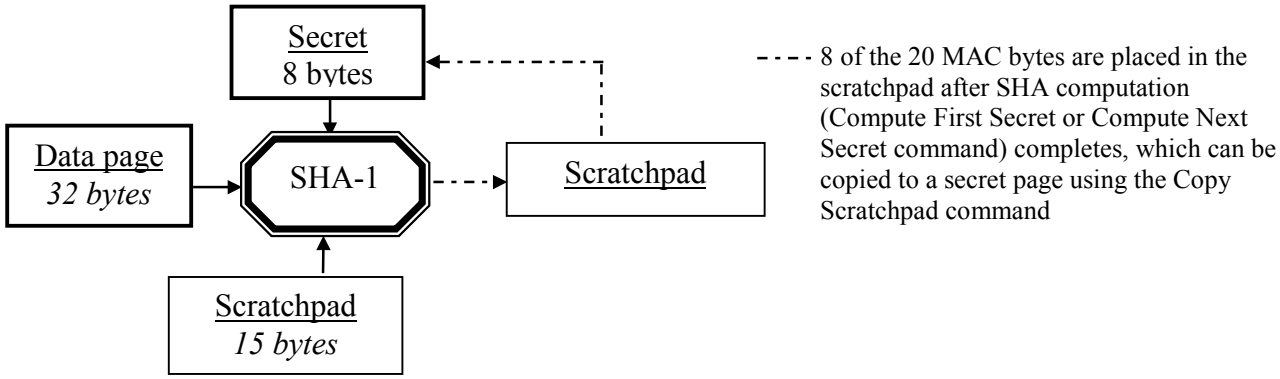
SECRET GENERATION IN SHA iBUTTON

Installing a secret into a SHA iButton should be done by writing data (partial phrases, see AN152 for more information) to the designated page and scratchpad, computing a MAC from the data, and copying the selected bytes of the MAC to the target secret number (see Figure 4). Directly writing to the secret should be avoided. Generating a secret via the on-chip SHA engine also allows multiple parties to participate in the secret installation process (called secret sharing), yet no single party can reproduce the system secret without the cooperation of others. This process significantly reduces the risk of secret exposure. The sequence for generating a system secret (authentication or signing) from *N* partial phrases (*partials[k], k=0 to N-1*) are outlined in Figure 5. At each iteration the MAC computation uses input from two sources: 47 bytes of input data (32 bytes written to a data page and 15 bytes written to the scratchpad), and the current contents of the secret (8 bytes). The secret is updated using the new MAC just computed before the next iteration. At the start of the loop (*k=0*) input from the secret assumes a null value. Since each step produces a new secret that becomes the input for the next calculation, the final system secret is a function of all the preceding input partial phrases.

The above sequence is equally applicable to the computation of system authentication secret and system signing secret except that the system signing secret must be installed in secret number 0

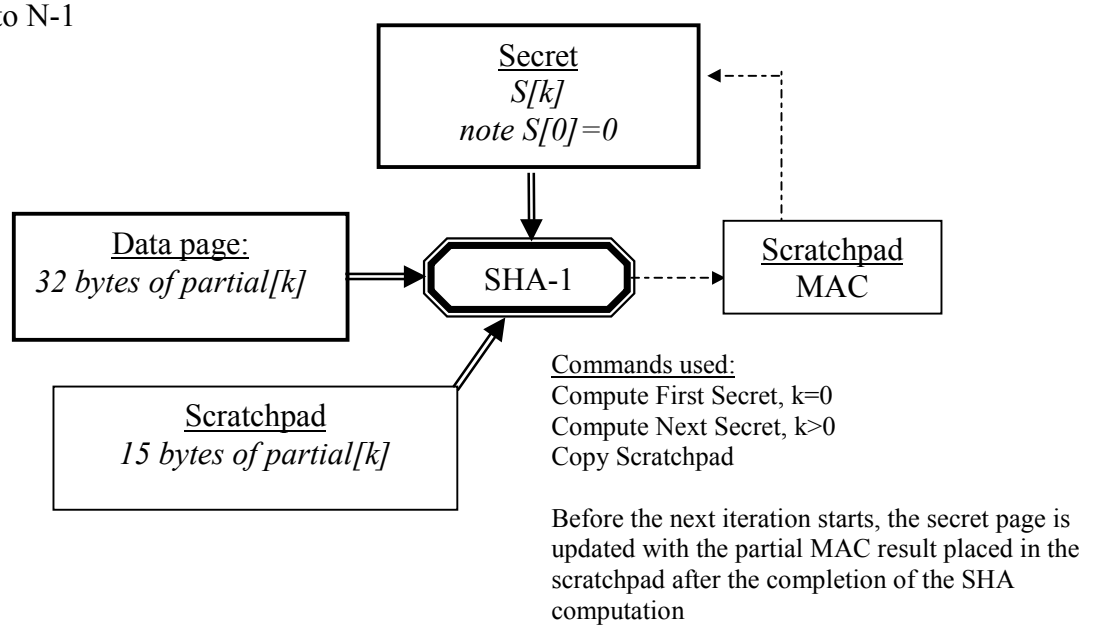
(*cSignSecretNum=0*) of the coprocessor. A general purpose function is implemented for installing a system secret from multiple partial phrases into a SHA iButton: *installSeystemSecret*, see section 0 for implementation details.

SECRET GENERATION IN SHA iBUTTON Figure 4



SYSTEM SECRET GENERATION USING SECRET SHARING Figure 5

Loop k=0 to N-1



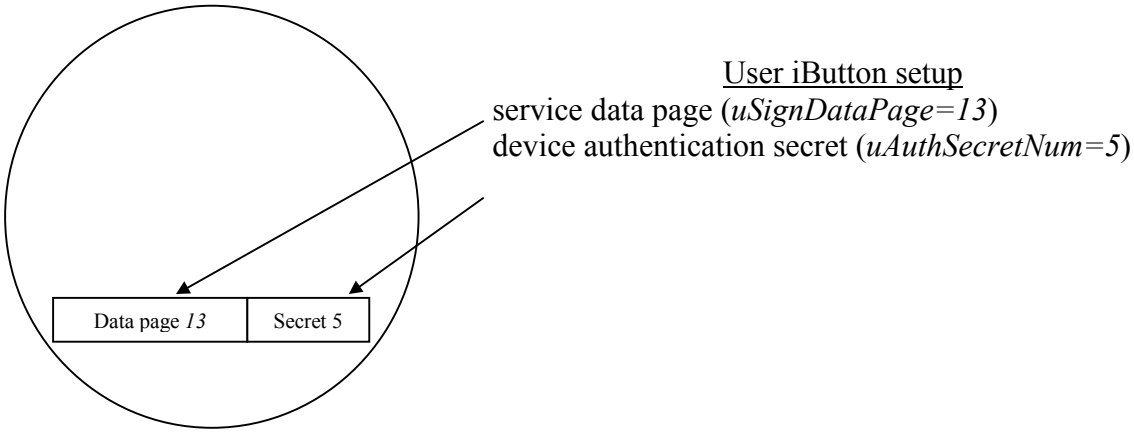
End loop

THE SAMPLE SERVICE

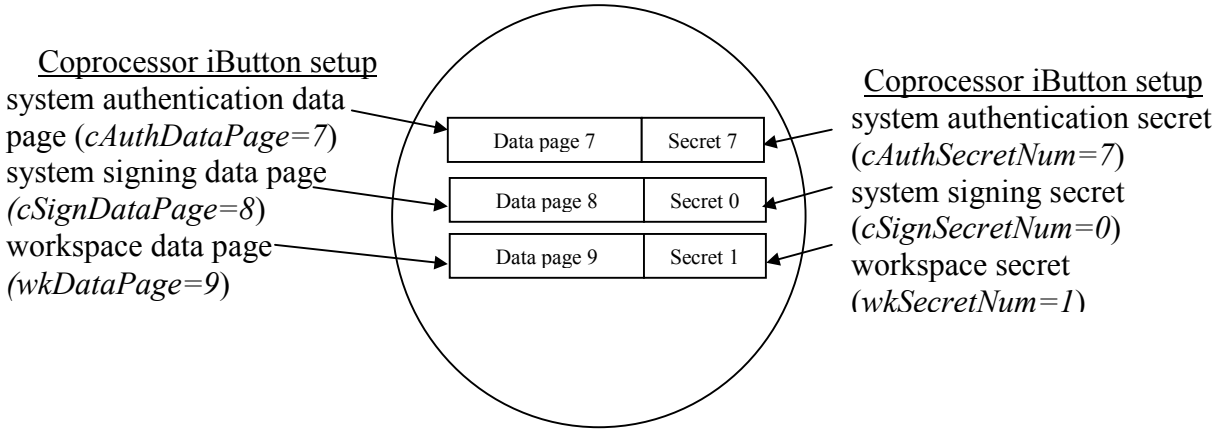
Our sample service is an e-payment service with its account balance debited and re-signed with a new signing signature at each transaction. Note that device authentication does not verify the contents of the service data page, but only involves the device authentication secret associated with the service data page in the user iButton. For service data signing, the system wide signing secret is stored in coprocessors, not in user iButtons. Therefore, we are able to use only one data page and one secret to host an e-payment application: a data page for service data and its matching secret for device authentication. Symbols and variables used in our sample service are summarized below for quick reference. Note that in a multiple services hosting device, the service data page number

(*uSignDataPage*) and authentication secret number (*uAuthSecreNum*) could be different in each user iButton.

The pages used in user SHA iButton are as follows:



The pages used in coprocessor SHA iButton are as follows:



LIST OF VARIABLES AND VALUES Table 3

List of Variables		
Variable	**Sample Service Value**	**Description**
cAN		coprocessor iButton address number
uAN		user iButtonaddress number
cAuthDataPage	7	system authentication data page, in coprocessor
cAuthSecretNum	7	system authentication secret number, in coprocessor
cSignDataPage	8	system signing data page, in coprocessor
cSignSecretNum	0	system signing secret number, in coprocessor
uAuthDataPage	13	user device authentication data page
uAuthSecretNum	5	user device authentication secret number
uSignDataPage	13	service data page number in user iButton
svcName	Dallas Semiconductor	service provider name, padded on the right with 00h bytes if needed

List of Variables		
Variable	**Sample Service Value**	**Description**
nAuthPartials	1	number of partial phrases for computing system authentication secret
nSignPartials	1	number of partial phrases for computing system signing secret
authPartial[j] j=0 to nAuthPartials-1	47 FFh bytes	partial phrases for computing the system authentication secret - 47 bytes in each array element: 32 bytes written to a data page and 15 bytes to scratchpad
signPartial[j] j=0 to nSignPartials-1	47 FFh bytes	partial phrases for computing the system signing secret - 47 bytes in each array element: 32 bytes written to a data page and 15 bytes to scratchpad
authBind	39 00h bytes	device authentication secret binding data - a 39-byte block used for binding the system authentication secret to a user device. The first 32 bytes are written to a data page and the remaining 7 bytes written to the scratchpad for computing the device authentication secret.
signCode	3 00h bytes	a 3-byte code used for creating the service data signing signature
signInitial	20 00h bytes	initial value for computing the service data signing signature
uData		service data page contents
TA1, TA2, ES		device address and status registers
uSignDataPageWCC		write cycle counter of the service data page, in user iButton *(=uAuthDataPageWCC*)
cFileName	COPR.0	coprocessor file name, stored in coprocessor
uFileName	DLSM.102	service file name, stored in user iButton
wkDataPage	9	workspace data page number in coprocessor
wkSecretNum	1	workspace secret number, in coprocessor

INSTALLING SYSTEM AUTHENTICATION SECRET IN COPROCESSOR

The command sequence and data flow for installing the system authentication secret in a SHA iButton are summarized below. The API function (*installSystemSecret*) implementation details are presented in section Appendix A.

INSTALLING SYSTEM AUTHENTICATION SECRET IN COPROCESSOR

Figure 6

Install System Authentication Secret in Coprocessor

Loop k=0 to nAuthPartials-1

- write first 32 bytes of authPartial[k] to scratchpad [Write Scratchpad]
- copy data from scratchpad to page cAuthDataPage [Copy Scratchpad]
- write the remaining 15 bytes of authPartial[k] to scratchpad [Write Scratchpad]

❶

- compute the secret; MAC is placed in scratchpad
  [Compute First Secret], if k=0
  [Compute Next Secret], if k>0

❷

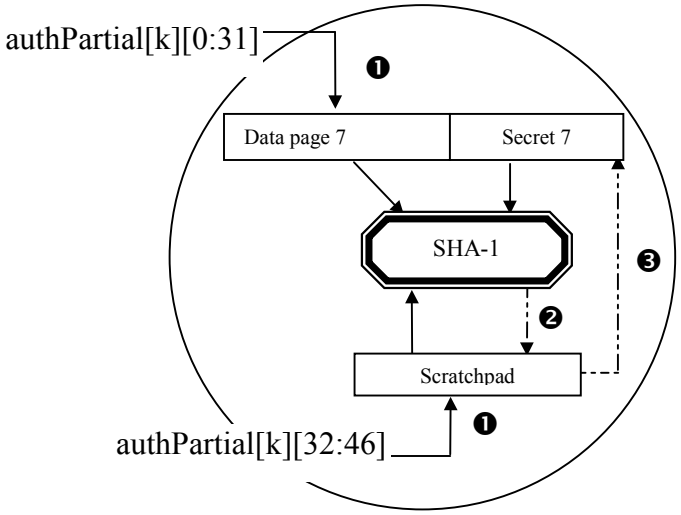
- prepare for copying the secret by writing dummy bytes to the scratchpad using the target secret address [Write Scratchpad]
- copy secret from scratchpad to secret cAuthSecretPage [Copy Scratchpad]

❸

End loop

To install the system authentication secret in a coprocessor using APIs:
s=installSystemSecret(cAN,cAuthDataPage, cAuthSecretNum, nAuthPartials,authPartial)
s=eraseDataPage(cAN,cAuthDataPage)

The call to *eraseDataPage* (implemented in section 0) erases the last partial phrase written to the coprocessor to prevent it from being exposed.



Coprocessor
System authentication secret is installed in secret *cAuthSecretNum* (=7) of coprocessor.
Partial phrases are written to page *cAuthDataPage* (=7) and scratchpad for computing the secret.

Please note that in this document for brevity reasons the API calls listing ignores any error checking and retry iterations. In an actual implementation each call return status should be checked for errors and appropriate iteration loops be wrapped around the call. For example the code snippet for installing the system authentication secret in a coprocessor with appropriate error checking and retries my look like below:

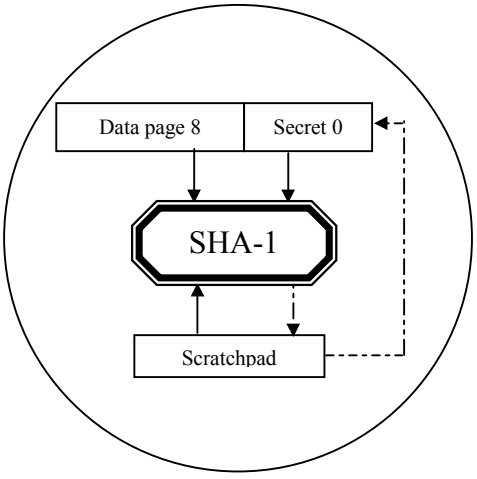
```
// iteration count limit
loopLimit = 5
```

```
s=1
loop=0
do while (loop<loopLimit and s<>0)
    // the call returns 0 if no error
    s=installSystemSecret(cAN,cAuthDataPage, cAuthSecretNum, nAuthPartials,authPartial)
    loop=loop+1
end loop

// quit if we still have errors after preset number of tries
if(s<>0) then
        exit
end if

s=1
loop=0
do while (loop<loopLimit and s<>0)
    // the call returns 0 if no error
    s=eraseDataPage(cAN,cAuthDataPage)
    loop=loop+1
end loop

// quit if we still have errors after preset number of iterations
if(s<>0) then
        exit
end if
…continue …
```

INSTALLING SYSTEM SIGNING SECRET IN COPROCESSOR

The command sequence and data flow for installing the system signing secret in a SHA iButton are identical to that for installing system authentication secret, except with different partial phrases and target data page and target secret (system signing secret must be installed in secret 0).

To install the system signing secret in a coprocessor using the APIs:
s=installSystemSecret(cAN,cSignDataPage, cSignSecretNum, nSignPartials,signPartial)
s=eraseDataPage(cAN,cSignDataPage)

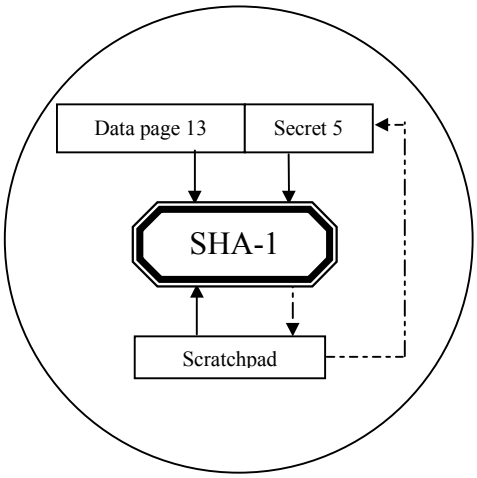


Coprocessor
System signing secret is installed in secret *cSignSecretNum* (=0) of coprocessor. Partial phrases are written to page *cSignDataPage* (=8) and scratchpad for computing the secret.

Note that system signing secret must be installed in secret 0 (*cSignSecretNum*=0), and its matching data page *cSignDataPage* can be either 0 or 8. In our sample service page 0 is used for OWFS directory, so we use *cSignDataPage*=8.

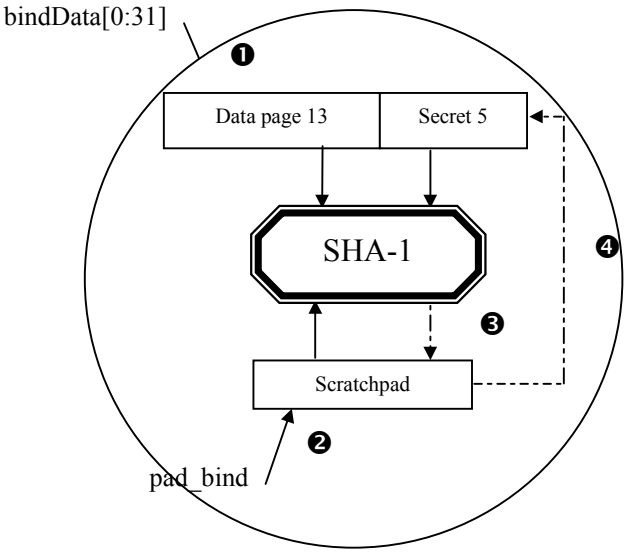
INSTALLING DEVICE AUTHENTICATION SECRET IN USER iBUTTON

To install the user iButton's device authentication secret, the system authentication secret is installed into the user iButton first (*secretAuthSystem*), then a unique device secret is computed from the system binding data (*bindData*), the system authentication secret (*secretAuthSystem*), the service data page number (*uSignDataPage*) and the address number (*uAN*) of the user iButton. It is the user iButton's address number that makes the device authentication secret unique to the iButton. The two step process is outlined below and an API (*bindSecretToiButton*) is implemented (see section 0 for details) for the second step.



User iButton Auth Secret Step 1
System authentication secret (*secretAuthSystem*) is installed in secret *uAuthSecretNum* (=5) of user iBuitton. Partial phrases are written to page *uAuthDataPage* (=13) and scratchpad for computing the secret.

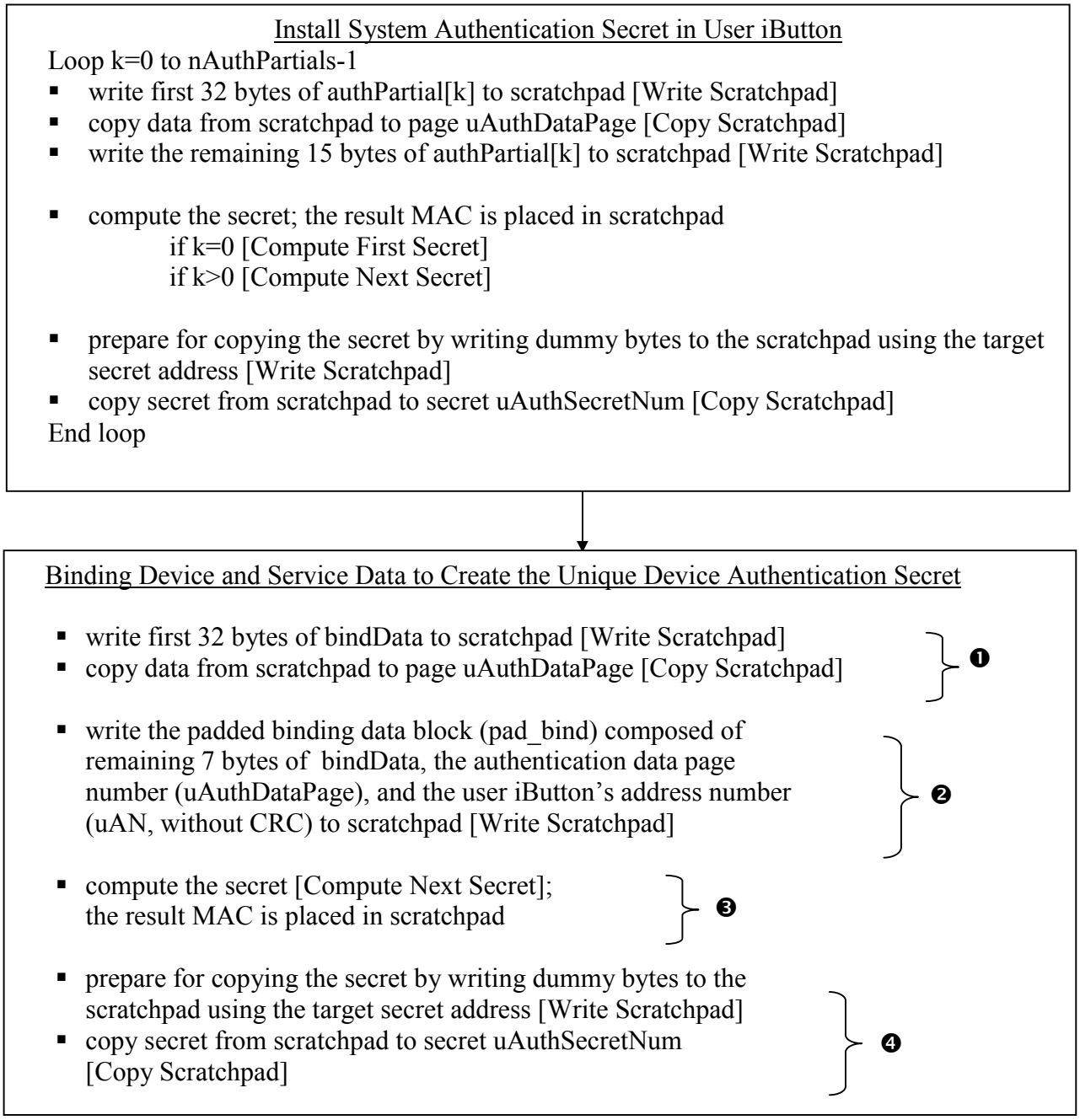
BINDING DEVICE AND SERVICE DATA TO CREATE THE DEVICE AUTHENTICATION SECRET



User iButton Auth Secret Step 2
Device authentication secret (*secretAuthDevice*) is installed in secret *uAuthSecretNum* (=5) of user iButtonby binding the system authentication secret (*secretAuthSystem*) with service and user iButtonspecific data (*uAuthDataPage, uAN*).

Pad the binding data, service and user device data in scratchpad for computing the unique device authentication secret (*pad_bind*):

offset	0:7	8:11	12:12	13:19	20:22	23:31
# of bytes	8 bytes	4 bytes	1 byte	7 bytes	3	9 bytes
data	00h	bindData[32:35]	uAuthDataPage	uAN[0:6]	bindData[36:38]	00h



To install a device authentication secret using the APIs:
s=installSystemSecret(uAN,uAuthDataPage,uAuthSecretNum,nAuthPartials,authPartial)
s=bindSecretToiButton(uAN,uAuthDataPage,uAuthSecretNum,bindData,uAuthDataPage,uAN)
s=eraseDataPage(uAN,uAuthDataPage)

CREATING SERVICE DATA SIGNATURE IN COPROCESSOR

The service data signature is computed from the service data and the system signing secret. The signature may be embedded[4] in the service data page and verified by a local host during each transaction. For example, in our sample service we embed the signature in the service data page as follows (the leading length byte, continuation pointer, and CRC-16 bytes are needed for OWFS):

Embed Data Signature in Service Data Page

offset	0:0	1:1	**2:21**	22:23	24:26	27:28	29:29	30:31
# of bytes	1 byte	1 byte	**20 bytes**	2 bytes	3 bytes	2 bytes	1 byte	2 bytes
data	length	data type	**signature bock**	conversion factor	account balance	trans ID	cont. pointer	CRC-16

Since the signature occupies part of the service data page, the signature block must be initialized to some known value (*signInitial*)[5] for computing the signature. Note that in order to prevent unauthorized copying and reuse of the service data (creating "money") the signature computation should take as input the user device specific data such as address number of the device (*uAN*), page number (*uSignDataPage*) and the write cycle counter value (*uSignDataPageWCC*) of the service data page. It is also important that the service data be installed into a page whose write cycle counter increments with each write operation if copying of the service data is to be prevented. The process for creating a service data signature is implemented in API *createDataSignature*, see section 0 for implementation details.

// create signature, the passed uData_ini has the signature block initialized to signInitial
sig = createDataSignature(cAN,cSignDataPage,cSignSecretNum,uData_ini,signCode, uAN,uSignDataPage,uSignDataPageWCC)

Padded Signing Data Block (pad_signCode):

offset	0:7	8:11	12:18	19:19	20:22	23:31
# of bytes	8 padding bytes	4 bytes	7 bytes	1 byte	3 bytes	9 padding bytes
data	*8 00h bytes*	*uSignDataPageWCC+1*[6]	*uAN[0:7]*	*uSignDataPage*	*signCode*	*9 00h bytes*

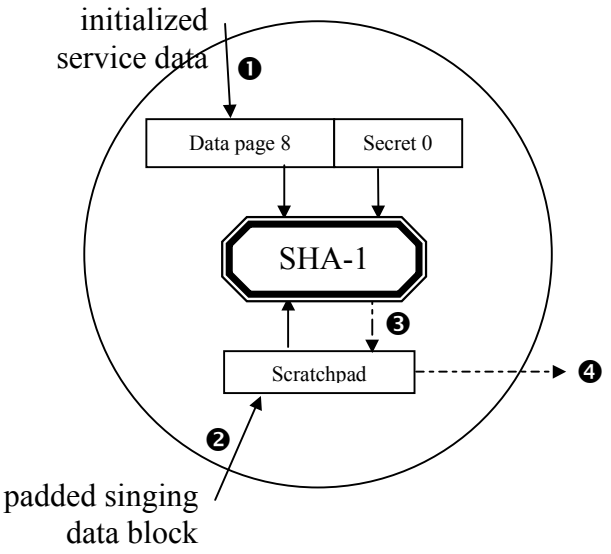
[4] Note that the signing signature does not have to be embedded in the service data page. It could be saved in a separate page and thus leave the full 32-byte space for raw service data.

[5] If signature is not embedded as part of the 32-byte service data page then no initialization is needed: *sig=createDataSignature(cAN,cSignDataPage,cSignSecretNum,uData,signCode,uAN,uSignDataPage, uSignDataPageWCC)*

[6] WCC+1 becomes the new write cycle counter of service data page after the signed service data is written to it because each writing to the page increments the cycle counter by 1 (on pages 8 to 15).

Create Service Data Signing Signature in Coprocessor

- write uData_ini to scratchpad [Write Scratchpad]
- copy data from scratchpad to page cSignDataPage [Copy Scratchpad] ❶
- write the padded signing data (pad_signCode) consisted of *signCode, uSignDataPageWCC, uAN (without the CRC), and uSignDataPage* to scratchpad [Write Scratchpad] ❷
- compute the signature [Sign Data Page]; the result MAC is placed in scratchpad ❸
- read the signature MAC from scratchpad [Read Scratchpad] ❹



Coprocessor

Service data signature is computed from:

- the initialized service data (*uData_ini*),
- the system signing secret (at *cSignSecretNum*=0)
- the system signing code (*signCode*)
- the user iButton address number *uAN* (without the CRC)
- the service data page number (*uSignDataPage)* and its page write cycle counter value *(uSignDataPageWCC)*

SERVICE INSTALLATION SUMMARY

The key API calls for installing a service are summarized below.

STATIC DATA SERVICES WITHOUT DATA VALIDATION

Installing service data into the coprocessor
// install the system authentication secret into the coprocessor
s=installSystemSecret(cAN,cAuthDataPage, cAuthSecretNum, nAuthPartials,authPartial)

// erase the partial phrase so it is not accidentally exposed
s=eraseDataPage(cAN,cAuthDataPage)

Installing service data into the user iButton
// install the system authentication secret into the user iButton
s=installSystemSecret(uAN,uAuthDataPage,uAuthSecretNum,nAuthPartials,authPartial)

// bind service data, user device ID with system secret to create a unique device auth secret
s=bindSecretToiButton(uAN,uAuthDataPage,uAuthSecretNum,bindData,uAuthDataPage,uAN)

s=eraseDataPage(uAN,uAuthDataPage) // erase the binding data from the data page

SERVICES WITH DATA VALIDATION

Installing service data into the coprocessor
// install the system authentication secret into the coprocessor
s=installSystemSecret(cAN,cAuthDataPage, cAuthSecretNum, nAuthPartials,authPartial)

// install the system signing secret into the coprocessor
s=installSystemSecret(cAN,cSignDataPage, cSignSecretNum, nSignPartials,signPartial)

s=eraseDataPage(cAN,cSignDataPage) // erase the partial phrase so it is not accidentally exposed
s=eraseDataPage(cAN,cAuthDataPage)

Installing service data into the user iButton
// install the system authentication secret into the user iButton
s=installSystemSecret(uAN,uAuthDataPage,uAuthSecretNum,nAuthPartials,authPartial)

// bind service data, user device ID with system secret to create a unique device auth secret
s=bindSecretToiButton(uAN,uAuthDataPage,uAuthSecretNum,bindData,uAuthDataPage,uAN)

// create the service data signature in coprocessor
sig=createDataSignature(cAN,cSignDataPage,cSignSecretNum,uData_ini,signCode,
uAN,uSignDataPage,uSignDataPageWCC)

// embed the signature in service data page
sData= ...

// write the service data with the embedded signature to user iButton
s=writeDataPage(uAN,uSignDataPage,sData)

CONDUCTING TRANSACTIONS

Conducting transactions with SHA iButtons may require two steps: the local host authenticating the user SHA iButton and updating the service data. A local host may consist of three logical components: coprocessor, transaction control unit (TCU) and a user interface such as display, token receptor, and service dispenser. A TCU maybe a personal computer or a microcontroller. It is important that the coprocessor and other elements containing secret and confidential information be physically and electronically secured inside an appropriate security boundary.

AUTHENTICATING THE USER iBUTTON

The user SHA iButton authentication process is depicted in Figure 7, which can be expressed in API calls as follows:

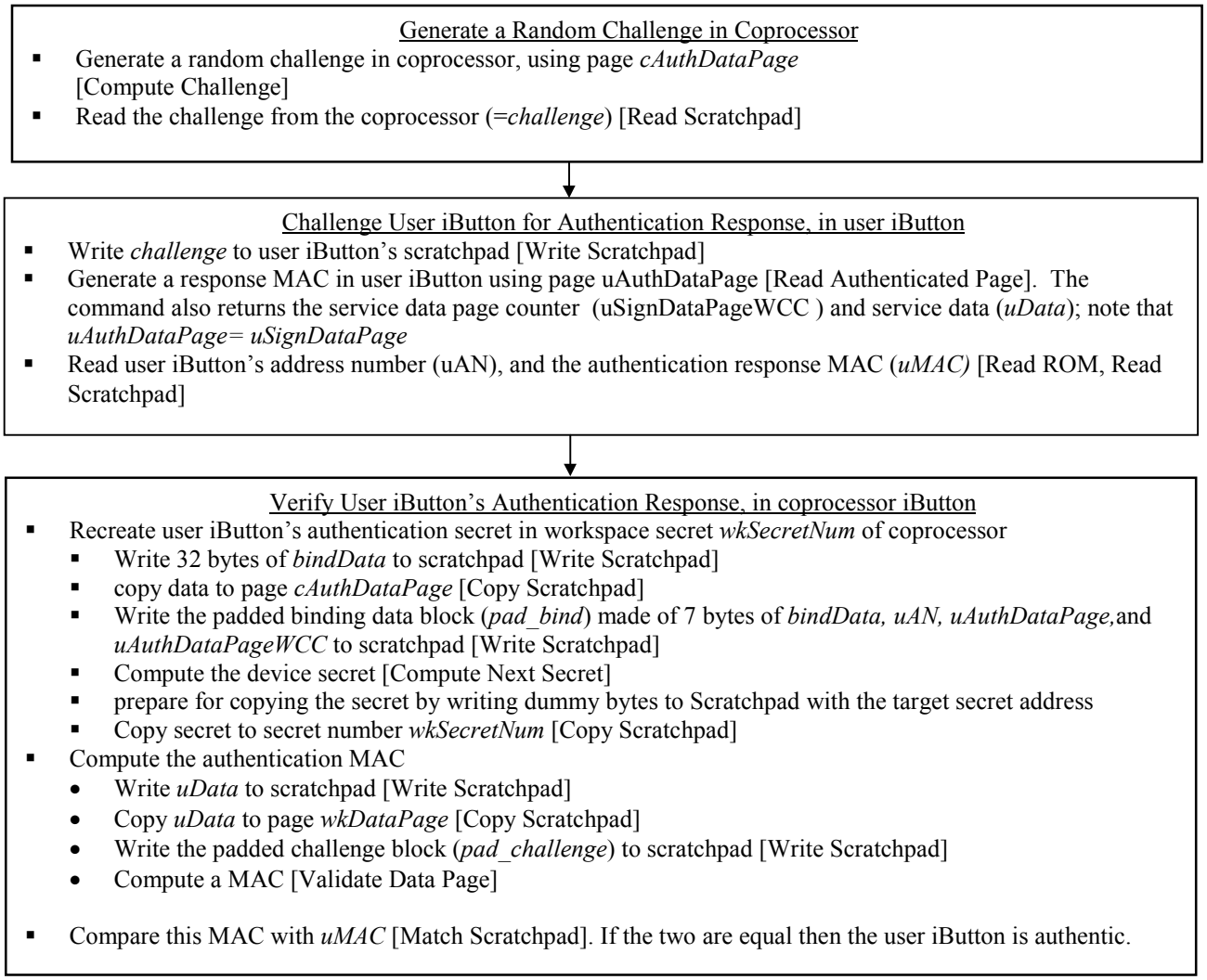
challenge=createChallenge(cAN,cAuthDataPage) // compute a challenge in the coprocessor

// challenge the user iButton for an authentication response
// the API call returns the service page data, page counter, and the response MAC
// uData=uResp[0:31]; uSignDataPageWCC=uResp[32:35]; uMAC=uResp[36:55]
uResp=answerChallenge(uAN,uAuthDataPage,challenge)

// recreate user iButton's device authentication secret in coprocessor
s=bindSecretToiButton(cAN,cAuthDataPage,wkSecretNum,bindData,uAuthDataPage,uAN)

// verify the response MAC in the coprocessor
s=verifyAuthResponse(cAN,wkDataPage,uData,uAN,uAuthDataPage,
uSignDataPageWCC,challenge,uMAC)

USER SHA iBUTTON AUTHENTICATION PROCESS Figure 7



padded binding data block (pad_bind)

8 bytes	4 bytes	1 byte	7 bytes	3	9 bytes
8 00h bytes	bindData[32:35]	uAuthDataPage	uAN[0:6]	bindData[36:38]	9 00h bytes

padded challenge data block (pad_challenge)

8 padding bytes	4 bytes	1 byte	7 bytes	3 bytes	9 padding bytes
8 00h bytes	*uAuthDataPageWCC*	*uAuthDataPage*	*uAN[0: 6]*	*challenge*	*9 00h bytes*

GENERATING A RANDOM CHALLENGE IN COPROCESSOR

To generate a random challenge in the coprocessor, the TCU would simply ask the coprocessor to perform a Compute Challenge operation. TCU can pick any 3 of the 20 MAC bytes placed in the scratchpad by the SHA engine upon completion of the command. The API function is called as follows, implementation details are presented in Appendix A.

challenge = createChallenge(cAN, cAuthDataPage)

CHALLENGING THE USER iBUTTON FOR AUTHENTICATION RESPONSE

TCU sends the challenge (*challenge)* to the user iButton and asks it to compute a response MAC based on the challenge (written to scratchpad offsets 20 to 22) and its device authentication secret. The API is implemented in Appendix A.

uResp=answerChallenge(uAN,uAuthDataPage,challenge)

Upon return uResp contains the service data (*uData*), the page write cycle counter (*uAuthDataPageWCC*), and the response MAC (*uMAC*).

uData=uResp[0:31]
uAuthDataPageWCC=uResp[32:35]
uMAC=uResp[36:55]

VERIFYING THE USER iBUTTON'S AUTHENTICATION RESPONSE

To verify an user iButton's response one must first recreate the user iButton's device authentication secret in the coprocessor. Note that the recreated authentication secret is not readable by the TCU, nor does it need to be. Authentication is not achieved by direct comparison of authentication secrets, rather of the results of SHA computations based on the authentication secret and other service and device specific data. After the user authentication secret has been recreated in the coprocessor, the Validate Data Page command is executed to compute the authorization MAC for comparison. The coprocessor stores its computation result (*cMAC*) in the scratchpad, hidden from external read. Match Scratchpad command is issued to compare the readout from the user iButton (*uMAC*) with *cMAC* in the scratchpad of the coprocessor.

The first step of the authentication process, recreating the user device authentication secret in the coprocessor, is the same as installing a device authentication secret in the user iButton during the service installation process except that this time the target device is the coprocessor and the secret is installed into an unused secret number of the coprocessor. The second step of verifying an user iButton's response MAC (*uMAC*) is implemented in Appendix A.

// recreate user device's unique authentication secret in coprocessor – to an unused
// secret number (wkSecretNum)
s=bindSecretToiButton(cAN,cAuthDataPage, wkSecretNum, bindData, uAuthDataPage, uAN)

// verify the user device's response MAC in coprocessor
s=verifyAuthResponse(cAN,wkDataPage,uData,uAN,uAuthDataPage,
uAuthDataPageWCC,challenge,uMAC)

VERIFYING THE EMBEDDED SERVICE DATA SIGNATURE

A service data page is verified by checking its embedded signature (*sign*). First the signature block of the service data page is set to a known initial value (*signInitial*) and a signature is re-computed in the coprocessor (*cSign*). TCU then compares *sign* with *cSign*, if the two are the same then the service data is valid. Note that we obtained the service data, write cycle counter of service data page while we authenticated the user iButton.

To recreate the signature in a coprocessor, one first saves the signature (*sign)* and then initialize the signature block of *uData* with *signInitial* to recover the original block (*uData_ini*) that was used to

generate the signature. We then use the user device and service parameters (address number and service page number and counter) to recreate the signature:

cSign=createDataSignature(cAN,cSignPage, cSignSecretNum, uData_ini, signCode, uSignDataPage, uAN, uSignDataPageWCC-1)

RE-SIGNING THE UPDATED SERVICE DATA

In dynamic data services, the service data is updated and a new signature is computed for re-signing the service data. The process is identical to that of installing the initial service data in user iButton. If *uData_Update* is the updated service data with the signature block initialized to *signInitial*, then the new signature is created as follows:

sign=createDataSignature(cAN, cSignPage,cSignSecretNum, uData_Update, signCode,uSignDataPage, uAN, uSignDataPageWCC)

SUMMARY OF CONDUCTING A TRANSACTION WITH THE SHA iBUTTONS

The key API calls for conducting a transaction are summarized below.

STATIC DATA SERVICES WITHOUT DATA VALIDATION

// create a random change code in coprocessor
challenge = createChallenge(cAN, cAuthDataPage)

// ask the user iButton to respond to the challenge
uResp=answerChallenge(uAN,uAuthDataPage,challenge)
uData=uResp[0:31]
uSignDataPageWCC=uResp[32:35]
uMAC=uResp[36:55]

// recreate user device's authentication secret in coprocessor
// in an unused secret number (wkSecretNum)
s=bindSecretToiButton(cAN,cAuthDataPage, wkSecretNum, bindData, uAuthDataPage, uAN)

// verify the user device's response MAC in coprocessor
status=verifyAuthResponse(cAN,wkDataPage,uData,uAN,uAuthDataPage, uAuthDataPageWCC,challenge,uMAC)
…

STATIC DATA SERVICES WITH DATA VALIDATION

// create a random change code in coprocessor
challenge = createChallenge(cAN, cAuthDataPage)

// ask the user iButton to respond to the challenge
uResp=answerChallenge(uAN,uAuthDataPage,challenge)
uData=uResp[0:31]
using=uData[4:23]
uSignDataPageWCC=uResp[32:35]
uMAC=uResp[36:55]

// recreate user device's unique authentication secret in coprocessor
// in an unused secret number (wkSecretNum)
s=bindSecretToiButton(cAN,cAuthDataPage, wkSecretNum, bindData, uAuthDataPage, uAN)

// verify the user device's response MAC in coprocessor
status=verifyAuthResponse(cAN,wkDataPage,uData,uAN,uAuthDataPage,
uAuthDataPageWCC,challenge,uMAC)

// recreate the data signature in coprocessor
// initialize uData with *signInitial*
uData_ini=…
cSign=createDataSignature(cAN,cSignPage, cSignSecretNum, uData_ini, signCode,
uSignDataPage, uAN, uSignDataPageWCC-1)

// compare the uSign and cSign
….

DYNAMIC DATA SERVICES

// create a random change code in coprocessor
challenge = createChallenge(cAN, cAuthDataPage)

// ask the user iButton to respond to the challenge
uResp=answerChallenge(uAN,uAuthDataPage,challenge)
uData=uResp[0:31]
uSignDataPageWCC=uResp[32:35]
uMAC=uResp[36:55]

// recreate user device's unique authentication secret in coprocessor
// in an unused secret number (wkSecretNum)
s=bindSecretToiButton(cAN,cAuthDataPage, wkSecretNum, bindData, uAuthDataPage, uAN)

// verify the user device's response MAC in coprocessor
s=verifyAuthResponse(cAN,wkDataPage,uData,uAN,uAuthDataPage,
uAuthDataPageWCC,challenge,uMAC)

// recreate the data signature in coprocessor
cSign=createDataSignature(cAN,cSignPage, cSignSecretNum, uData_ini, signCode,
uSignDataPage, uAN, uSignDataPageWCC-1)

// compare the uSign and cSign
…

// update the service data and recompute signature
sign=createDataSignature(cAN, cSignPage,cSignSecretNum, uData_Update,
signCode,uSignDataPage, uAN, uSignDataPageWCC)

// write the new service data and signature (*upData*) to the user iButton
s=writeDataPage(uAN,uAuthDataPage,upData)

// authenticate the user iButton and verify service data again
challenge = createChallenge(cAN, cAuthDataPage)

// ask the user iButton to respond to the challenge
uResp=answerChallenge(uAN,uAuthDataPage,challenge)
uData=uResp[0:31]
uSignDataPageWCC=uResp[32:35]
uMAC=uResp[36:55]

// make sure the user iButton received the updated data, compare uData and upData
…

// verify the user device's response MAC in coprocessor
status=verifyAuthResponse(cAN,wkDataPage,uData,uAN,uAuthDataPage,
uAuthDataPageWCC,challenge,uMAC)

APPENDIX A: IMPLEMENTING THE APIS

The implementation details of various APIs are discussed in this section. A developer should be able to adopt them to the computer language of his choice. Please note that these listings are not actual or functional computer codes. They are only intended to demonstrate the basic command and data flow sequences. For brevity the listing ignores error checking and retry loops. In the actual implementation error checking and retries are always needed.

BASIC 1-WIRE DEVICE I/O OPERATIONS

The necessary 1-wire device access functions are assumed to be provided by a driver for the appropriate platform. The actual names of functions may differ from those listed below, however the functionalities should be the same. We assume that the following helper functions have been provided.

BASIC 1-WIRE I/O OPERATIONS Table 4

S=select(devAN)	Selects the device whose address number (*devAN*) is given in the argument. The call returns 0 if device is selected successfully.
S=select()	Selects the device whose *devAN* is the same as the last one accessed. The call returns 0 if device is selected successfully.
S=resume()	Resumes communication with the last communicated device on the network. The call returns 0 if device is selected successfully.
S=reset()	Resets the 1-wire net, call returns 0 if operation is successful.
Data=readBytes(len)	Reads *len* bytes from the selected device
S=writeBytes(block,from,len)	Writes *len* bytes (starting at offset *from* of the data array *block*) to the selected device. The call returns 0 if operation is successful.
TA1=lowAddress(page)	Returns the low address byte of a data page number
TA2=highAddress(page)	Returns the high address byte of a data page number

TA1S=lowSecretAddress(secretNum)	Returns the low address byte of a secret number
TA2S=highSecretAddress(secretNum)	Returns the high address byte of a secret number
C=CRC16(block,from,len,seed)	Computes the CRC-16 of the given *len* data elements (starting at offset *from* of array *block*), with the given seed value *seed*.
C=invCRC16(block,from,len,seed)	Computes the inverted CRC-16 of the given len data elements (starting at offset *from* of array *block*) with the given seed value *seed*

WRITE TO A DATA PAGE (WRITEDATAPAGE)

This API writes a 32-byte data block to a data page.

int status = writeDataPage(byte[] devAN, byte pageNum, byte[] data)

variable	type	Description
devAN	byte[8]	target iButton address number
pageNum	byte	target page number
data	byte[32]	the data block to be written
status	int	0 = no error status<>0 means error occurred. Depending on the actual implementation, the values of status may represent various sources of errors.

TA1=lowAddress(pageNum)
TA2=highAddress(pageNum)

Command	Data stream	Notes
select(…)	devAN	selects the target iButton for communication
Erase Scratchpad	[W]: {C3h, TA1, TA2}	this clears the HIDE flag for subsequent I/O
readBytes(len)		read enough (for example len=5) status bytes to check if operation has completed. A value of AAh indicates completion.
reset()		resets the above operation
resume()		resumes communication with the target device
Write Scratchpad	[W] {0F, TA1, TA2, data}	writes the data block to scratchpad
readBytes(2)		reads the inverted CRC-16 of the above data stream. TCU should compute its version of the CRC-16 for the same data stream and compare with the value read from the target device. A mismatch indicates errors in the I/O.
reset()		
resume()		
Read Scratchpad	[W] AAh [R] {TA1, TA2, ES}	reads address and ES registers. TCU should check if these readings are correct.
reset()		
resume()		
Copy Scratchpad	[W] {55h, TA1, TA2, ES}	copies scratchpad data to target page.
readBytes(len)		check for operation status. A value of AAh byte in the returned block indicates the completion of the copy operation.
reset()		resets the network and call returns

A simplistic way of trapping errors and iterating a preset number of loops would be to put the above steps in a loop and let the code flow jump to the beginning of the loop each time an error occurs. This

approach is easy to implement but sacrifices on execution speed. A more tighter error trapping would be to check each command call for error and only iterate on the offending call when an error occurs. The table below illustrates a simplistic approach.

Command	Data stream	Notes
loop=0 ***start:*** *loop=loop+1* *do while loop<=loopLimit*		
k=select(…)	devAN	selects the target iButton for communication
if(k<>0) goto start		
Erase Scratchpad	[W]: {C3h, TA1, TA2}	this clears the HIDE flag for subsequent I/O
readBytes(len)	{status bytes}	read enough (for example len=5) status bytes to check if operation has completed. A value of AAh indicates a success.
if (the last status byte is not AAh) goto start		
s1=reset()		resets the above operation
s2=resume()		resumes communication with the target device
if(s1<>0 or s2<>0) goto start		
Write Scratchpad	[W] {0F, TA1, TA2, data}	writes the data block to scratchpad
readBytes(2)	{inv CRC-16 bytes}	reads the inverted CRC-16 of the above data stream. TCU should compute its version of the CRC-16 for the same data stream and compare with the value read from the target device. A mismatch indicates errors in the I/O.
if (CRC bytes do not match) goto start		
s1=reset()		
s2=resume()		
if(s1<>0 or s2<>0) goto start		
Read Scratchpad	[W] AAh [R] {TA1, TA2, ES}	reads address and ES registers. TCU should check if these readings are correct.
if (TA1, TA2, and ES do not match) goto start		
reset()		
resume()		
Copy Scratchpad	[W] {55h, TA1, TA2, ES}	copies scratchpad data to target page.
readBytes(len)	{status bytes}	check for operation status. At value of AAh byte in the returned block indicates the completion of the copy operation.
if (the last status byte <>AAh) goto start		
s1=reset()		resets the network
end loop: *if(loop>loopLimit)* *status=1* *else* *status=0*		

ERASING A DATA PAGE (ERASEDATAPAGE)

This API erases the contents of a data page by filling it with 32 FFh bytes.

int status = eraseDataPage(byte[] devAN, byte pageNum)

variable	type	description
devAN	byte[8]	target iButton address number
pageNum	byte	target page number
status	int	see above description for the return *status* variable

set data[32]={32 FFh bytes}
status=writeDataPage(devAN, pageNum, data)

COPY MAC TO A SECRET PAGE (COPYMACTOSECRET)

This API copies 8 bytes of a result MAC in the scratchpad to a secret address memory. This function is often used after the Compute First Secret and Compute Next Secret commands to copy the partial MAC placed in the scratchpad to a secret memory.

int status = copyMACtoSecret(byte[] devAN, byte secretNum)

variable	type	description
devAN	byte[8]	target iButton address number
secretNum	byte	the target secret number

TA1S=lowSecretAddress(secretNum)
TA2S=highSecretAddress(secretNum)
set tmp_data[32]={32 00h bytes}

Command	Data stream	Notes
select(devAN)		selects the target iButton for communication
Write Scratchpad	[W] {0Fh, TA1S, TA2S, tmp_data} [R] {inverted CRC-16 bytes}	sets up the address flags for next copying operation. The returned CRC bytes are used for error detection
reset()		
resume()		
Read Scratchpad	[W] AAh [R] {TA1S,TA2S,ES}	reads back the address and E/S registers. TCU checks if these registers are correct
reset()		
resume()		
Copy Scratchpad	[W] {55h, TA1S, TA2S, ES}	copies secret from scratchpad to secret number *secretNum* Note that only 8 bytes from scratchpad are copied into the secret memory.
readBytes(len)		read enough status bytes to check if the copying operation has completed
reset()		

INSTALLING A SYSTEM SECRET (INSTALLSYSTEMSECRET)

Installing system secret in a SHA iButton is performed with both the coprocessor and user iButtons, during service installation and conducting transactions. The process for installing system authentication secret and installing system signing secret are identical except that the partial phrases are usually different and that the system signing secret must be installed in secret number 0 of the coprocessor. The API implemented for this general purpose is *installSystemSecret.*

int status= installSystemSecret(byte[] devAN, byte pageNum, byte secretNum, int numPartials, byte[] partial)

variable name	type	description
devAN	byte[8]	address number of the target iButton
pageNum	byte	target data page number of device to write partial phrases to
secretNum	byte	target secret number, for the intermediate and final secrets
numPartials	int	number of partial phrases
partial	byte[size]	partial phrase array. It is defined as a single dimension array. *size=47*numPartials*

TA1=lowAddress(pageNum)
TA2=highAddress(pageNum)
TA1S=lowSecretAddress(secretNum)
TA2S=highSecretAddress(secretNum)

PADDING PARTIAL PHRASE (*PAD_PARTIAL[J]*) Table 5

8 padding bytes	*15 bytes*	*9 padding bytes*
8 00h bytes	*partial[j][32:46]*	*9 00h bytes*

Command	**Data stream**	**Notes**
Loop j=0 to numPartials-1		
		write partial[j] to data page pageNum
writeDataPage(…)	devAN, pageNum, partial[j][0:31]	write the first 32 bytes of partial[j] to data page pageNum
		write padded partial (pad_ partial) to scratchpad
resume()		
Write Scratchpad	[W] {0Fh, TA1, TA2, pad_partial[j]} [R] {inverted CRC-16 bytes}	writes the padded binding data code to scratchpad. The returned inverted CRC bytes are used for error detection
reset()		
resume()		

If (j=0)
ComputeFirstSecret
else
ComputeNextSecret
	If (j=0)
	[W]{33h,TA1,TA2,0F}
	else
	[W]{33h,TA1,TA2,F0}
	
	[R] {inverted CRC-16 bytes}
		compute MAC (secret)
		launches the Compute First Secret or Compute Next Secret function depending on the value of j. Partial result of MAC is placed in scratchpad.

readBytes(len)		read enough (for example len=5) status bytes to check if operation has completed. A value of AAh indicates success.
reset()		
copyMACtoSecret(…)	devAN, secretNum	**copy 8 bytes of the MAC to target secret**
reset()		
End of Loop		

BINDING A SECRET TO THE USER iBUTTON (BINDSECRETTOIBUTTON)

An unique device authentication secret is created by binding the system authentication secret with the service data page number and the device address number.

int status = bindSecretToiButton(byte[] devAN,byte pageNum,byte secretNum,
byte[] bindData, byte uAuthDataPage, byte[] uAN)

variable	type	description
devAN	byte[8]	target device address number
pageNum	byte	target data page number
secreNum	byte	authentication secret number
bindData	byte[39]	a 39-byte block system wide binding data
uAuthDataPage	byte	data page number for the binding data
uAN	byte[8]	address number of the user device

TA1=lowAddress(pageNum)
TA2=highAddress(pageNum)
TA1S=lowSecretAddress(secretNum)
TA2S=highSecretAddress(secretNum)

PADDING BINDING DATA (PAD_BIND) Table 6

8 padding bytes	4 bytes	1 byte	7 bytes	3 bytes	9 padding bytes
8 00h bytes	*bindData[32:35]*	*uAuthDataPage*	*uAN[0:6]*	*bindData[36:38]*	9 00h bytes

Command	**Data stream**	**Notes**
writeDataPage(…)	devAN, pageNum, bindData[0:31]	**writes binding data bindData[0:31] to page pageNum**
		write padded binding data (*pad_bind)* to scratchpad
resume()		
Write Scratchpad	[W] {0Fh, TA1, TA2, pad_bind} [R] {CRC-16 bytes}	writes the padded binding data to scratchpad TCU checks the returned operation status bytes CRC-16 to detect any errors.
reset()		
		compute the MAC (device secret)
resume()		
Compute Next Secret	[W]{33h,TA1,TA2,F0} [R] {bytes of inverted CRC-16}	launches the Compute Next Secret function. Partial result of MAC is stored in scratchpad for next copying operation
readBytes(len)		read enough status bytes to check if operation has completed.
reset()		
copyMACtoSecret(…)	devAN, secretNum	**copy the device secret from scratchpad to target secret**

CREATING THE SERVICE DATA SIGNATURE (CREATEDATASIGNATURE)

This API computes a signature for the service data using the service data, the system signing secret, the service data page number and its write cycle counter, and the address number of user iButton.

byte sig[]=createDataSignature(byte[] devAN, byte pageNum, byte secretNum, byte[] uData, byte[] signCode, byte uSignDataPage, byte[] uAN, byte[] uSignDataPageWCC)

The call returns a 20 byte MAC if successful.
Parameters and data used for creating a service data signature are listed below:

variable	type	description
devAN	byte[8]	target iButton address number (coprocessor)
pageNum	byte	the data page number to which the service data is written for computing the signature
secretNum	byte	the secret number where the system signing secret is stored (must be 0)
uData	byte[32]	the user data block to be signed
signCode	byte[3]	a system wide code used for computing the signature
uSignDataPage	byte	the service data page number of user iButton, used in computing the signature
uAN	byte[8]	the user device address number used for computing the signature
uSignDataPageWCC	byte[4]	write cycle counter of service data page (uSignDataPage) in user iButton

TA1 = lowAddress(pageNum)
TA2=highAddress(pageNum)

For I/O efficiency the *signCode* and other service parameters are padded into a 32-byte block and written to the scratchpad at once.

PADDING SIGNING DATA (*PAD_SIGNCODE*) Table 7

8 padding bytes	4 bytes	7 bytes	1 byte	3 bytes	9 padding bytes
8 00h bytes	*uSignDataPageWCC+1*[7]	*uAN[0:7]*	*uSignDataPage*	*signCode*	*9 00h bytes*

Command	**Data stream**	**Notes**
writeDataPage(…)	devAN, pageNum, uData	**write uData to page pageNum**
		write padded sign data (pad_signCode) to scratchpad
resume()		
Write Scratchpad	[W] {0Fh, TA1, TA2, pad_signCode} [R] {inv. CRC bytes}	writes the padded signing data (pad_signCode) to scratchpad
reset()		
		compute the signature
resume()		
Sign Data Page	[W]{33h,TA1,TA2, C3} [R]{inv CRC-16 bytes}	launches the Sign Data Page command. The result MAC is stored in scratchpad
readBytes(len)		read enough status bytes to check if operation has completed.
reset()		

[7] WCC+1 becomes the new write cycle counter of service data page after the signed service data is written to it because each writing to the page increments the cycle counter by 1 (on pages 8 to 15).

resume()		
Read Scratchpad	[W] AAh [R]{TA1S, TA2S, ES, scratchpad data, and CRC bytes}	reads the address, ES registers and scratchpad data. The 20-byte signature block is stored starting at offset 8 of the scratchpad.
reset()		resets 1-wire network and returns

CREATING THE CHALLENGE BYTES (CREATECHALLENGE)

This API returns a 3-byte challenge, often generated in the coprocessor. The challenge has a random nature in the sense that the SHA computation uses the SHA engine counter as one input parameter (the counter increments each time when a SHA computation is carried out in the device and the user would not get the same challenge bytes twice. Since this SHA computation does take input data from a data page and the scratchpad, one could make the challenge truly random by using environmental conditions (such as temperature, humidity, noise level, and the reader probe force etc.) as input to the computation. This API implementation simply replies on the SHA engine counter value and whatever contents in the memory page and scratchpad to provide an ever changing MAC.

byte[] challenge=createChallenge(byte[] devAN, byte pageNum)

variable	type	description
devAN	byte[8]	target iButton address number
pageNum	byte	target data page number. Only the secret associated with this page is used in this simple implementation.

TA1 = lowAddress(pageNum)
TA2 = highAddress(pageNum)

Command	**Data stream**	**Notes**
select(…)	devAN	selects the device for communication
Erase Scratchpad	[W] {C3h, TA1, TA2}	clears the HIDE flag so that scratchpad data could be read
reset()		
resume()		
Compute Challenge	[W]{33h,TA1,TA2,CCh}	launches the Compute Challenge command
readBytes(2)	[R] {inverted CRC-16 bytes}	reads two bytes of the inverted CRC-16 of command, TA1, TA2 and control byte.
readBytes(len)		check to see if SHA computation has completed. A valle of AAh indicates success.
reset()		
resume()		
Read Scratchpad	[R] {b0,b1,…b31}	reads 32 bytes from the scratchpad. The SHA computation result is stored between offsets 8 and 27. Pick any three bytes as the desired challenge (=*challenge*)
reset()		resets the 1-wire network and returns

ANSWERING THE AUTHENTICATION CHALLENGE (ANSWERCHALLENGE)

When a user iButton is presented with a challenge from the local host, it responds with a MAC it computes form the selected data page and its device authentication secret.

byte[] resp=answerChallenge(byte[] devAN, byte pageNum, byte[] challenge)

The call returns the following data items packed in a single dimension byte array if successful, null if error occurred.

uData[32] = resp[0:31] data from page *pageNum*
pageWCC [4] = resp[32:35] write cycle counter of page *pageNum*
uMAC[20] = resp[36:55] the authentication response MAC

TA1=lowAddress(pageNum)
TA2=highAddress(pageNum)

variable name	type	description
devAN	byte[8]	target device address number
pageNum	byte	target data page number. Note that this page number must pair with the device authentication secret number.
challenge	byte[3]	challenge bytes
resp	byte[56]	a one-dimensional data array packed per the table above.

For I/O efficiency the challenge bytes are often padded to a full 32-byte block for writing to scratchpad (*pad_challenge*):

PADDING CHALLENGE BYTES (PAD_CHALLENGE)

20 bytes padding	3 bytes	9 bytes padding
20 00h bytes	*challenge*	9 00h bytes

Command	**Data stream**	**Notes**
		write padded challenge bytes (pad_challenge) to scratchpad
select(…)	devAN	selects the user button for communication
Erase Scratchpad	[W]{C3h,TA1,TA2}	Erases the scratchpad, sets the device ready for receiving data
reset()		
resume()		
Write Scratchpad	[W] {0Fh, TA1, TA2, *pad_challenge*}	writes the padded challenge data to scratchpad.
reset()		
		compute the response MAC
resume()		
Read Auth. Page	[W] {A5h, TA1, TA2} [R] {32 bytes data from data page (*uData*), counter of data page (*pageWCC*), counter of secret page, inverted CRC-16 of command, address, data, and counter bytes}	(1) this command causes the user button to compute a MAC based on its authentication secret, data of selected page, and the challenge from scratchpad. (2) TCU would read contents of data page, counter of data page, counter of secret page, and inverted CRC-16 of command, address, data and counter bytes.
readBytes(len)		read status bytes to see if SHA computation has completed
reset()		
		read response MAC *(uMAC)*

resume()		
Read Scratchpad	R: {TA1, TA2, ES, 32 bytes scratchpad data} uMAC starts at offset 8 and ends at 27	reads the computation result from the scratchpad. Note that the desired MAC (20 bytes) is stored starting at offset 8.
reset()		resets the 1-wire network and return

VERIFYING THE AUTHENTICATION RESPONSE (VERIFYAUTHRESPONSE)

This API is called to verify if the user iButton's authentication response is correct. The function must be called after the user iButton's device authentication secret has been recreated in a workspace secret in the coprocessor.

int status = verifyAuthResponse(byte[] devAN, byte wkDataPage, byte[] uData, byte[] uAN, byte uPageNum, byte[] uPageWCC, byte[] challenge, byte[] uMAC)

status =	0	user response MAC (uMAC) is valid
	1	user response is invalid
	-1	error occurred

variable name	type	description
devAN	byte[8]	target device address number
wkDataPage	byte	workspace data page number
uData	byte[32]	user device data page (of page uPageNum) contents, obtained when challenging user device for the authentication response.
uAN	byte[8]	user device address number
uPageNum	byte	user device authentication data page number
uPageWCC	byte[4]	user device authentication data page write cycle counter
challenge	byte[3]	the challenge bytes, used to challenge user iButton for authentication MAC
uMAC	byte[20]	the user iButton's authentication MAC

TA1W=lowAddress(wkDataPage)
TA2W=highAddress(wkDataPage)

For I/O efficiency, the challenge and other service and device parameters are padded to a full 32 byte page for writing to scratchpad.

PADDING CHALLENGE AND SERVICE PARAMETERS (PAD_AUTH)

8 padding bytes	4 bytes	1 byte	7 bytes	3 bytes	9 padding bytes
8 00h bytes	*uPageWCC*	*uPageNum*	*uAN[0:6]*	*challenge*	*9 00h bytes*

Command	**Data stream**	**Notes**
writeToDataPage(...)	devAN,wkDataPage,uData	**write uData to workspace data page**
		write service parameters and challenge bytes to scratchpad
resume()		
Write Scratchpad	[W] {0FH, TA1W, TA2W, *pad_auth*} [R] {inverted CRC-16 bytes}	writes the padded challenge bytes and service parameters to scratchpad
reset()		
		compute the response MAC
resume()		
Validate Data Page	[W] {33h,TA1W, TA2W,3Ch} [R] {inverted CRC-16}	The computation results are placed in the scratchpad between offsets 8 and 27.
readBytes(len)		read enough (for example len=5) status bytes to see if operation has completed
reset()		
		check the user iButton response MAC
resume()		
Match Scratchpad	[W] {3Ch, *uMAC*} [R] {inverted CRC-16 bytes, status bytes}	Check the last status byte: AAh = data matches FFh = data does not match
reset()		resets the network and returns