Application Note 156
DS1963S SHA 1-Wire API Users Guide

1.0 Introduction

The purpose of this document is to familiarize 1-Wire® software developers with the APIs for producing secure SHA applications. There is an API available in both of the major development kits: the 1-Wire API for Java™ and the 1-Wire Public Domain Kit. This document should serve as both a tutorial for designing new secure applications with each API, as well as a walkthrough for a deeper understanding of the demo applications shipped with each development kit.

This document assumes a basic understanding of the SHA iButton® hardware as well as the 1-Wire protocol for using iButtons. The data sheet for the DS1963S (SHA iButton) is available on the website (www.maxim-ic.com). Also, there are application notes available that detail the structure of a signed certificate for eCash systems [AN151, *Maxim Digital Monetary Certificates*], a high-level protocol for secure applications [AN157, *SHA iButton API Overview*], a SHA-1 overview [AN1201, *1-Wire SHA-1 Overview*], and the implementation of a file system for memory devices [AN114, *1-Wire File Structure*].

In any monetary SHA application, the two major components are the coprocessor and the user token. The coprocessor is a DS1963S initialized for verifying a user token as a member of the system and validating the user's certificate. A user's token is a DS1963S (or comparable 1-Wire device) that carries a monetary certificate and identifies a user of the system. For each API, this document will give an overview of the methods used for initializing the coprocessor, initializing the user token, and performing transactions. Each transaction can be broken down to further steps, which could include authenticating the user, verifying the transaction data, and updating the transaction data with dynamic information.

For all the code samples in this document, the code boxes with the light gray shading indicate that the code is the breakdown of a higher-level task to the most detailed of operations. In the 1-Wire API for Java, this low-level code will use the OneWireContainer18 class. In the Public Domain Kit, it will use functions from the SHA18.C module.

2.0 SHA Applications with the 1-Wire API for Java

The solution for SHA Applications in the 1-Wire API for Java introduces a new package into the development kit: com.dalsemi.onewire.application.sha. A snapshot of the classes in that package, as well as their inheritance relationships, is shown in Figure 1. When detailing the nature of the necessary methods introduced in that package, the container-level methods will be considered atomic and no further breakdown will be provided (although the source is available in the kit).

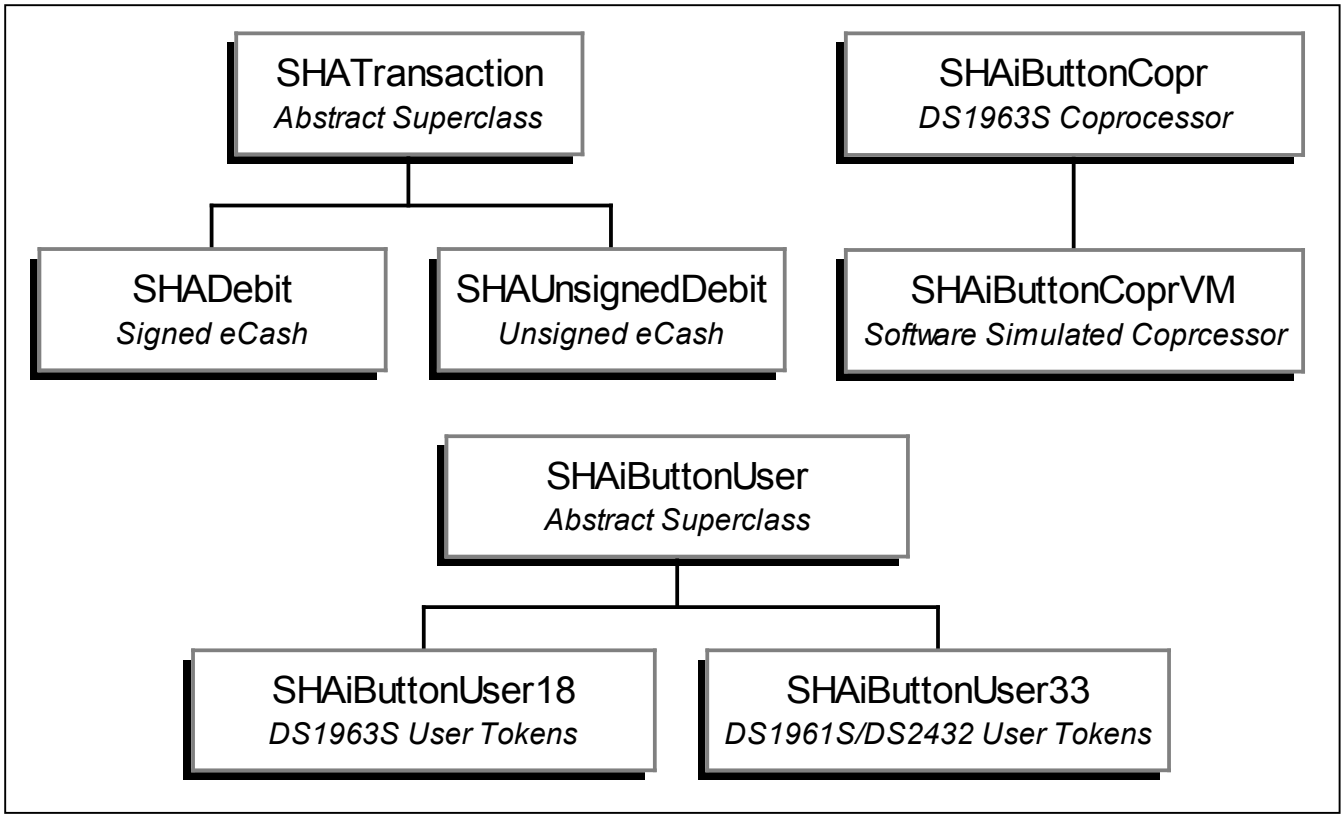
The SHATransaction abstract superclass is meant to represent all secure transactions. This class defines three key methods, which represent the fundamental steps in a typical transaction:

1) Verify that the user token is a valid member of the system (verifyUser)
2) Verify that the data is properly certified and has not been tampered with (verifyTransactionData)
3) Update the data as necessary and ensure the token receives the update (executeTransaction)

There are two sample transactions that extend SHATransaction: SHADebit and SHADebitUnsigned. The first implements an account debit system, where an initial account balance is stored in a signed monetary certificate (see AN151). During typical use, a user is validated as a member of the system and the signature on the certificate is verified. If the user and data are both valid, the balance stored in the monetary certificate is updated, a new signature is generated, and the certificate is written back to the device. The second transaction (SHADebitUnsigned) uses an unsigned monetary certificate and utilizes the unused space to implement a double-write scheme, protecting the data in EEPROM devices. This transaction type (for use with the DS1961S/DS2432 1-Wire devices) is covered in another application note.

SHA transactions have, as a constant member, a SHAiButtonCopr (the class that represents a DS1963S that has been initialized to perform as a coprocessor in SHA transactions). In that sense, a SHATransaction can be thought of as an extension to the SHAiButtonCopr. It serves the same purpose as the "decorator" classes in Java in that it separates the "thing to sign with" from "what it's signing." That way, the "thing to sign with" (i.e. the coprocessor, SHAiButtonCopr) can be easily moved to different systems ("what it's signing") with minimal coding for the application developer. In addition, a SHAiButtonCoprVM, or rather a simulated coprocessor, can be used in place of a hardware DS1963S. This class is provided as a convenience, but for optimal security a hardware coprocessor is recommended.

Inheritance Hierarchy of Package Figure 1



Every method in SHATransaction has a SHAiButtonUser as a parameter. Provided with the API are two user objects: SHAiButtonUser33 and SHAiButtonUser18. Although the default implementation of SHAiButtonUser only supports the two kinds of SHA iButtons (DS1961S/DS2432 family 0x33 and DS1963S family 0x18), the concept of a user of a SHATransaction could be extended to support any 1-Wire memory device. The SHAiButtonUser33 and SHAiButtonUser18 are handy extensions, however, as they both provide authentication along with carrying the required account data. If iButton authentication

is something that isn't important to the specific system, the SHAiButtonUser class could easily be extended to support 1-Wire memory devices, which would just carry signed account information. Note that using other memory devices rules out the use of a challenge-response protocol for authenticating the iButton. There is still a minimal level of security in the user token's 64-bit ROM ID.

2.1 Initializing the Coprocessor

Initializing the coprocessor consists of two important steps: installing the system authentication secret and installing the system signing secret. An optional third step is to write all of the system configuration data to the iButton in the form of a file (see AN114). The configuration data doesn't have to be stored on the coprocessor iButton, but it's convenient to keep the system parameters as portable as the coprocessor. As an alternative, the file can be stored on a disk drive or the parameters can be hardcoded in your application.

The following blocks of code demonstrate the initialization of the coprocessor. Each block of code is well commented and the declarations of all variables used are shown. The intent is that the code can be copied into a project used after just a few edits.

Necessary System Parameters for a SHA Application Figure 2

```
/* The input data to be used for calculating the master authentication and signing
 * secrets.  The calculation involves splitting the secret into blocks of 47 bytes
 * (32 bytes to the data page, 15 bytes to the scratchpad) and then performing the
 * SHA command Compute First Secret, followed by Compute Next Secret for each 47
 * byte block after the first. */
byte[] inputAuthSecret, inputSignSecret;  // . . . initialize from user input

/* The page to use for the signing secret must be page 8, because secret 0 is the
 * only secret that can be used for the SHA command Sign Data Page.  For a list of
 * SHA commands, and what pages they can be used on, see the DS1963S datasheet. */
int signPageNumber = 8;

/* The authentication secret can be on any page as long as it doesn't collide with
 * the file holding coprocessor configuration information and not secret 0 */
int authPageNumber = 7;

/* The workspace page is the page the coprocessor will use for recreating a user's
 * unique authentication secret and it's authentication response. */
int wspcPageNumber = 9;

/* The binding information is used to create the "unique" secret for each button.
 * bindData is written to the data page of the user token, and bindCode is written
 * to the scratchpad.  The result of a Compute Next Secret using the system
 * authentication secret, the account page number, the rom ID of the user token, the
 * bind data, and the bind code becomes the user's unique authentication secret.
 * This is used to initialize a user token and the coprocessor must use the same
 * data to recreate the user token's unique secret. */
byte[] bindData = new byte[32], bindCode = new byte[7]; // . . . user input

/* Initial signature to use when signing the certificate */
byte[] initSignature = new byte[20]; // . . . user input

/* Three byte challenge used when signing the certificate */
byte[] signingChlg = new byte[3]; // . . . one time random calculation or user input

/* Name of the account file (and file extension) to create on user tokens */
byte[] acctFilename = ("DLSM").getBytes();
int acctFileExt = 102; // extension for "money" files
```

Although there are a few other parameters for a system that are stored in the service configuration file, these are the most essential parameters. The name of the provider and the version number of the system, for example, are not necessary for a minimal-functionality debit application. Initializing the coprocessor can be done by simply calling the constructor for SHAiButtonCopr, passing all the parameters as arguments. For a detailed list of all the parameters, refer to the JavaDocs for the 1-Wire API for Java.

Initializing the Coprocessor with SHAiButtonCopr Figure 3

```
// . . . Find the DS1963S iButton on the 1-Wire Network
OneWireContainer18 coprDevice = // see 'overview' in the JavaDocs for finding devices
/* Create the new instance of SHAiButtonCopr, ready to perform transactions! */
SHAiButtonCopr copr
   = new SHAiButtonCopr(coprDevice, "COPR.0", true, signPageNumber, authPageNumber,
      wspcPageNumber, 1, 1, acctFileExt, acctFilename, "Maxim", bindData, bindCode,
      "", initSignature, signingChlg, inputSignSecret, inputAuthSecret);
```

Using just the OneWireContainer18 class, it's possible to initialize the container without the com.dalsemi.onewire.application.sha package. The following code snippet demonstrates the steps for installing the secrets onto the device using just the OneWireContainer18. Note that this code segment is nearly identical to the actual implementation of the SHAiButtonCopr constructor used in Figure 3.

Container-Level Initialization of Coprocessor Figure 4

```
/* install the signing secret */
coprDevice.installMasterSecret(signPageNumber, inputSignSecret, signPageNumber&7);

/* install the authentication secret */
coprDevice.installMasterSecret(authPageNumber, inputAuthSecret, authPageNumber&7);

/* create the configuration file */
OWFileOutputSream fos = new OWFileOutputStream(shaDevice, "COPR.0");
/* Write all the service parameters to the output stream (see AN157 for format) */
fos.write(acctFilename,0,4);
fos.write(acctFile);
fos.write(signPageNumber);
fos.write(authPageNumber);
fos.write(wspcPageNumber);
fos.write(0); //version number
fos.write(0x01); //month
fos.write(0x01); //day
fos.write(0x07); //year MSB
fos.write(0xD1); //year LSB -> 1/1/2001
fos.write(bindData);
fos.write(bindCode);
fos.write(signingChlg);
fos.write(l_providerName.length);
fos.write(l_initSignature.length);
fos.write(l_auxData.length);
fos.write("Maxim".getBytes()); //Provider name
fos.write(initSignature);
fos.write("".getBytes()); //auxilliary info
fos.write(l_encCode); //encryption code
fos.write(0); //DS1961S Compatibility flag (see AN157)
fos.close();

/* Create the new instance of SHAiButtonCopr, ready to perform transactions! */
SHAiButtonCopr copr = new SHAiButtonCopr(coprDevice, "COPR.0");
```

If the SHA coprocessor will be used to authenticate DS1961S/DS2432 iButtons or to produce the copy-scratchpad authorization, the master authentication secret will have to be padded to accommodate the DS1961S's smaller scratchpad. A utility function is provided which will pad the secret appropriately. Then the "DS1961S compatibility flag" in the coprocessor file should be set to a non-zero value.

Padding the Authentication Secret Figure 5

```
inputAuthSecret = SHAiButtonCopr.reformatFor1961S(inputAuthSecret);
```

2.2 Initializing the User Token

Initializing the user token consists of two steps. First, install the master authentication secret and bind it to the iButton to produce the unique secret for the token. Second, write the certificate file to the iButton. It is actually a little more complicated than it may initially sound. The DS1963S has eight secrets that correspond to the 16 pages of memory (where each secret is shared by two pages). The certificate file must be written to a page which has a write cycle counter, which limits it to the last eight pages of the device's memory banks. Also, the certificate file must be written to one of the pages whose corresponding secret is the secret where the master authentication secret was installed. But, the 1-Wire file API doesn't allow the specification of a page number for the file to be stored on. The best that can be done is to give the file one of the reserved extensions that ensures special treatment. Extensions 101 and 102, for example, are reserved for files that must be written on pages with write cycle counters if the device has them (see AN114). One solution for this is to use the 1-Wire file API to create an empty stub file to write the certificate. This creates the proper directory entries for the certificate so it can be located dynamically. Then, the page number from the directory entry specifies the page where the master authentication secret should be installed and bound to the user token. When the certificate is actually written to the device, it is done with a direct page write, rather than using the file API. This will allow for much faster updates (very important in a responsive debit application) when actually debiting the token.

Using the SHAiButtonUser18 class, the installation of the secret as well as the creation of an empty account file can be done automatically with a constructor call. The actual contents of the file still need to be written after this step (i.e. the signed certificate must still be written to the device).

Initializing User Tokens with SHAiButtonUser Figure 6

```
/* For DS1963S user tokens */
OneWireContainer18 owc18 = // see overview in JavaDocs for finding devices.
SHAiButtonUser18 user18 = new SHAiButtonUser18(copr, owc18, true, inputAuthSecret);
```

Figure 7 illustrates how the master authentication secret is installed on the user token, as well as the actual binding of the master authentication secret to produce a secret that is unique to the user token.

Container-Level Installation of Authentication Secret in User Token Figure 7

```
/* Create the empty file on the user token */
OWFile owf = new OWFile(owc18, "DLSM.102");
owf.format();
owf.createNewFile();

/* Get the page number the account file will be stored on */
int acctPage = owf.getPageList()[0];
owf.close();

/* Install the master authentication secret, same as on the coprocessor */
owc18.installMasterSecret(acctPage, inputAuthSecret,  acctPage&7);

/* Create full binding code for DS1963S, for format see AN157 */
byte[] fullBindCode = new byte[15];
copr.getBindCode(fullBindCode, 0);
System.arraycopy(fullBindCode, 4, fullBindCode, 12, 3);
fullBindCode[4] = (byte)this.accountPageNumber;
System.arraycopy(owc18.getAddress(), 0, fullBindCode, 5, 7);

/* bind the master secret to this token to create it's unique secret */
owc18.bindSecretToiButton(acctPage, copr.getBindData(), fullBindCode, acctPage&7);
```

The final step when initializing a user token is to actually write the account certificate to the data page where the file was stored. The certificate has an option about whether or not it is signed. The main situation where there is no need for concern about an unsigned certificate is when using a DS1961S/DS2432, which requires knowledge of the secret to write to it.

Installing a Signed Certificate with SHADebit Figure 8

```
/* create the signed data file, sign it, and write it to the user token */
SHADebit debit = new SHADebit(copr, 1000/*initial amount*/, 50/*debit amount*/);
Debit.setupTransactionData(user18); // can be user18 or user33
```

Whether signed or unsigned, the format of the certificate remains the same. If left unsigned however, the 20 bytes reserved for the MAC can be used for any system specific data. Figure 9 illustrates how to manually format a certificate file as specified in AN151.

Creating a New Certificate Figure 9

```
/* eCertificate, see format in AN151 */
byte[] acctData = new byte[32];

acctData[0] = 29; // file length
acctData[1] = 0x01; // data type code or algorithm (0x01 dynamic eCash)

copr.getInitialSignature(acctData, 2); // Initial Signature

acctData[22] = 0x8B; acctData[23] = 0x48; // Conversion factor (ISO4217)
acctData[24] = 0xE8; acctData[25] = 0x03; acctData[26] = 0; // Account Balance ($10)
acctData[27] = 0; acctData[28] = 0; // TransactionID
acctData[29] = 0x00; // file continuation pointer
acctData[30] = 0x00; accountData[31] = 0x00; // ~CRC16
```

If the certificate data needs to be signed, the coprocessor device is used to sign the account data. For the DS1963S user token, it is necessary to get the value of the write cycle counter before writing the

certificate to the data page. The current value of the write cycle counter is then incremented by 1, since the data that it will be verified with is about to be written.

The certificate's signature is produced on the coprocessor using the Sign Data command. The command has a few variable inputs. The first of which is the data page that is to be signed. In this case, that page is the actual account file (as constructed in Figure 9), which is written to the signing page of the scratchpad. The rest of the inputs are all stored on the scratchpad of the coprocessor (referred to as the coprocessor's "signing scratchpad"). The first parameter stored on the scratchpad is the value of the write cycle counter for the part. Next is the page number of the user token's memory pages that hold the account file. This is followed by least significant 56 bits of the user token's address (64-bit Rom ID minus the CRC8). The last parameter is the 3-byte constant challenge set when the coprocessor was initialized.

Retrieving Write-Cycle Counter from DS1963S Figure 10

```
/* if using a DS1963S, need to get the value of the write-cycle counter.  Doing a
 * read authenticated page on the device will accomplish this. */
owc18.readAuthenticatedPage(acctPage, rawData, 0);

/* get the value of the write cycle counter for DS1963S user token only */
int wcc = (rawData[35]&0x0ff);
wcc = (wcc << 8) | (rawData[34]&0x0ff);
wcc = (wcc << 8) | (rawData[33]&0x0ff);
wcc = (wcc << 8) | (rawData[32]&0x0ff);
wcc += 1; // and increment it since we are going to write to the device
```

Setting up the Coprocessor's Scratchpad for Data Signing Figure 11

```
byte[] signScratchpad = new byte[32];

/* assign the wcc to the coprocessor's "signing" scratchpad */
signScratchpad[8]  = (wcc&0x0ff);
signScratchpad[9]  = ((wcc>>=8)&0x0ff);
signScratchpad[10] = ((wcc>>=8)&0x0ff);
signScratchpad[11] = ((wcc>>=8)&0x0ff);

/* get the page number of the account file */
signScratchpad[12] = (byte)acctPage;

/* get the Rom ID of the user token */
System.arraycopy(owc18.getAddress(),0,signScratchpad, 13, 7);

/* get the signing challenge */
System.arraycopy(signingChlg, 0, signScratchpad, 20, 3);
```

SHAiButtonCopr Helper Method for Signing Data Figure 12

```
/* sign the data with the coprocessor and return the mac right in the data */
copr.createDataSignature(acctData, signScratchpad, acctData, 2);
```

The helper method shown in Figure 12 can be broken down into the necessary container methods used to implement signing data. Figure 13 illustrates the process for creating a data signature.

Signing Data with Coprocessor at Container-Level Figure 13

```
/* write the account data to the signing page of the coprocessor */
coprDevice.writeDataPage(signPageNumber, acctData);

/* write the signScratchpad to the scratchpad of the coprocessor */
coprDevice.writeScratchpad(signPageNumber, 0, acctData, 0, 32);

/* sign the data and read the signature*/
coprDevice.SHAFunction(coprDevice.SIGN_DATA_PAGE, signPageNumber << 5);
coprDevice.readScratchpad(signScratchpad, 0);

/* place the resulting signature in the certificate */
System.arraycopy(signScratchpad, 8, acctData, 2, 20);
```

For the data in the account file byte array to be a valid account file, it must have an inverted, two-byte CRC16 at the end of the file. More detail on what makes a valid file on the 1-Wire file structure can be found in AN114.

Adding Inverted CRC16 to 1-Wire File Figure 14

```
/* calculate the inverted CRC16 */
int crc = ~CRC16.compute(acctData, 0, acctData[0]+1, acctPage);

/* now the file is ready to be written */
acctData[30] = (byte)crc;
acctData[31] = (byte)(crc>>8);
```

Writing the account file to the data page of the DS1963S user token is fairly easy. There is a utility function provided in the container class for writing a page of data, the exact function which is used to write the account data to the coprocessor for signing in the above code block.

2.3 Authenticating the User Token

Authenticating the user token involves a simple challenge-response scheme. First, the coprocessor is used to generate a 3-byte, pseudo-random challenge. This challenge is then written to the scratchpad of the user token and the Read Authenticated Page command is issued (see DS1963S data sheet). This returns the entire contents of the memory page followed by the value of the write cycle counter for the page and the write cycle counter for the secret location. The scratchpad of the DS1963S user token will contain the 20-byte SHA result of the user's unique authentication secret, the page number the data was read from, the serial number of the user token, and the random challenge. The coprocessor is then used to reproduce this SHA result, ensuring that the user token is a valid member of the system.

Authenticating User Token with SHADebit Figure 15

```
/* Verify user tokens authentication response, same for signed and unsigned */
SHADebit debit = new SHADebit(copr, 65536, 2);
debit.verifyUser(user18);
```

The process for verifying a user is the same for both signed and unsigned transactions. Naturally, this can be broken down into simple steps using the SHAiButtonCopr class and the SHAiButtonUser class. The code snippet presented in Figure 16 demonstrates the basic implementation of the user verification method in the transaction classes (minus the error checking).

Using SHAiButtonCopr and SHAiButtonUser for Authentication Figure 16

```
/* Use coprocessor to generate a random challenge */
copr.generateChallenge(0, challenge, 0);

/* issue challenge to user getting back the account data, response MAC, and the
 * value of the write-cycle counter */
int wcc = user18.readAccountData(challenge, 0, rawData, 0, responseMAC, 0);
scratchpad[8]  = (wcc&0x0ff);
scratchpad[9]  = ((wcc>>=8)&0x0ff);
scratchpad[10] = ((wcc>>=8)&0x0ff);
scratchpad[11] = ((wcc>>=8)&0x0ff);

/* get user's full binding code */
fullBindCode = user18.getFullBindCode();

/* create the coprocessor's "signing" scratchpad */
System.arraycopy(fullBindCode, 4, scratchpad, 12, 8);
System.arraycopy(challenge, 0, scratchpad, 20, 3);

if(copr.verifyAuthentication(fullBindCode, rawData, scratchpad, responseMAC,
                             user18.getAuthorizationCommand()))
   System.out.println("User Token Authentication Successful!");
```

Figure 17 illustrates how to use the OneWireContainer18 class to produce a random challenge. Also included in that caption is the declaration of the variables used in the rest of the code snippets for this section.

Using the Coprocessor to Generate a Random Challenge Figure 17

```
byte[] rawData = new byte[42];
byte[] scratchpad = new byte[32];
byte[] responseMAC = new byte[20];
byte[] challenge = new byte[3];

/* Use the coprocessor to generate the challenge, page number is irrelevant but a
 * highly used page will generate a more random (less repeating) number */
coprDevice.eraseScratchPad(authPageNumber);
coprDevice.SHAFunction(ibcL.COMPUTE_CHALLENGE, authPageNumber << 5);
coprDevice.readScratchPad(scratchpad, 0);

/* copy the challenge bytes into challenge buffer */
System.arraycopy(scratchpad, 20, challenge, 0, 3);
```

After performing the steps in Figure 17, the scratchpad buffer of the coprocessor now contains the 20-byte result of a SHA calculation, starting at index 8. Since any three bytes are as good as any other three for a challenge, it is safe to leave the result as it is in the scratchpad. Indices 20 to 22 of the coprocessor's scratchpad hold the challenge bytes that will be used for the Read Authenticated Page. Using these three particular bytes makes it unnecessary to write the challenge back to the coprocessor later.

Answering a Random Challenge with a DS1963S User Token Figure 18

```
/* Write the challenge to the scratchpad of the user token */
owc18.writeScratchpad(acctPage, 0, scratchpad, 0, 32);

/* reads 42 bytes = 32 bytes of page data
 *                +  4 bytes page counter
 *                +  4 bytes secret counter
 *                +  2 bytes CRC  */
owc18.readAuthenticatedPage(acctPage, rawData, 0);

/* get the value of the write cycle counter*/
int wcc = (rawData[35]&0x0ff);
wcc = (wcc << 8) | (rawData[34]&0x0ff);
wcc = (wcc << 8) | (rawData[33]&0x0ff);
wcc = (wcc << 8) | (rawData[32]&0x0ff);

/* Read Scratchpad for resutling SHA computation and copy it into a buffer*/
owc18.readScratchpad(scratchpad, 0);
System.arraycopy(scratchpad, 8, responseMac, 0, 20);
```

After retrieving the MAC (Message Authentication Code) from the user token, it is now the job of the coprocessor to verify it. To verify the response, the user token's unique secret is recreated in the workspace secret of the coprocessor and the raw account data is signed with this workspace secret.

Validating the Authentication Response Figure 19

```
/* Bind DS1963S user token's unique secret to coprocessor */
byte[] fullBindCode = new byte[15];

/* Get the 7-byte binding code
copr.getBindCode(fullBindCode,0);
System.arrayCopy(fullBindCode, 4, fullBindCode, 12, 3);

/* get the page number of the account file and 7 bytes of the ROM ID */
fullBindCode[4] = (byte)acctPage;
System.arraycopy(owc18.getAddress(), 0, fullBindCode, 5, 7);

/* recreate user token's unique secret in wspc */
coprDevice.bindSecretToiButton(authPageNumber,
                               bindData, fullBindCode, wspcPageNumber);

/* the scratchpad of the coprocessor also needs the user's ROM ID and page number, */
 * In addition to the challenge bytes used and the write-cycle counter. */
System.arraycopy(fullBindCode, 4, scratchpad, 12, 8);
System.arraycopy(challenge, 0, scratchpad, 20, 3);
scratchpad[8]  = (wcc&0x0ff);
scratchpad[9]  = ((wcc>>=8)&0x0ff);
scratchpad[10] = ((wcc>>=8)&0x0ff);
scratchpad[11] = ((wcc>>=8)&0x0ff);

/* write to the coprocessor and validate */
coprDevice.writeDataPage(wspcPageNumber, rawData);
coprDevice.writeScratchpad(wspcPageNumber, 0, scratchpad, 0, 32);
coprDevice.SHAFunction(OneWireContainer18.VALIDATE_DATA_PAGE, wspcPageNumber);

/* match the signature generated by the coprocessor with the user token's MAC */
if( coprDevice.matchScratchpad(responseMAC, 0) )
   System.out.println("DS1963S Authentication Successful!");
```

2.4 Validating the User's Certificate

Validating a certificate from a user token looks a lot like the earlier section on initializing the data. The steps for signing the data are very similar, but the coprocessor-generated signature is merely matched against the existing signature, rather than being read into the certificate. Also, since the entire certificate is merely stored on a file in any user token, the process for validating the certificate is identical across all tokens. When using an unsigned certificate, this step may only consist of checking the user's account balance to make sure it makes sense in the system (i.e. it isn't negative or it isn't a trillion dollars).

Validating Transaction Data with SHADebit Figure 20

```
/* Verify user tokens data, including verifying the data signature */
SHADebit debit = new SHADebit(copr, 1000, 50);
debit.verifyTransactionData(user18);
```

For signed transaction data, the signature of the certificate must be verified with the coprocessor. As a nice initial check, the validity of the account balance information can be verified first. This can provide a quick check to avoid verifying the signature of the account information if it's unnecessary.

Validating the Account Balance Information Figure 21

```
byte[] acctData = // . . . already known from verifying the user
int wcc = // . . . already known from verifying the user

/* verify the user's account balance is 'legal' */
int balance = (acctData[26]&0x0ff);
balance = (balance << 8) | (acctData [25]&0x0ff);
balance = (balance << 8) | (acctData [24]&0x0ff);

/* MAX_BALANCE, MIN_BALANCE, and DEBIT_AMOUNT are constant integers */
if( balance>MAX_BALANCE )
   System.out.println("Too Much Money!");
else if( (balance-DEBIT_AMOUNT)<MIN_BALANCE )
   System.out.println("Not enough money to perform transaction!");
```

The verification of the signature looks very similar to how the signature was created in the first place. The CRC16 is cleared out (set to zero) and the data signature, after being saved in a buffer, is cleared out and replaced with the system's initial signature. Since the new signature will not be written back to the part and the current signature is being verified against the current value of the write cycle counter, the write cycle counter is not incremented this time for producing the data signature. The entire process is shown in Figure 22.

Note that, rather than reading the signature back and matching the signature bytes in software, the coprocessor's Match Scratchpad command is used. Since this command only transmits 20 bytes of MAC information to the coprocessor, rather than reading 32 bytes of scratchpad information from the coprocessor, this technique is slightly faster. When building a fast, responsive debit application, all non-essential reads/writes will have to be optimized in this manner. In addition, making sure that the correct MAC isn't transmitted from the coprocessor gives an invalid user token zero opportunity to grab the right MAC. These will leave the application less open to holes where the user can take advantage of system "retries." For example, if the system was designed to retry the user challenge operation if interrupted, but eventually repeated a 3-byte random challenge, the user could have stored the appropriate signature.

Container-Level Validation of the Certificate Signature Figure 22

```
/* save the data signature */
byte[] dataSignature = new byte[20];
System.arraycopy(acctData, 2, dataSignature, 0, 20);

/* clear out the old signature and CRC 16 */
copr.getInitialSignature(acctData, 2);
acctData[30] = 0x00;
acctData[31] = 0x00;

/* assign the wcc to coprocessor's "signing" scratchpad (DS1961S wcc=0xFFFFFFFF) */
scratchpad[8]  = (wcc&0x0ff);
scratchpad[9]  = ((wcc>>=8)&0x0ff);
scratchpad[10] = ((wcc>>=8)&0x0ff);
scratchpad[11] = ((wcc>>=8)&0x0ff);

/* setup the rest of the "signing" scratchpad */
scratchpad[12] = (byte)user.getAccountPageNumber();
user.getAddress(scratchpad, 13, 7);
copr.getSigningChallenge(scratchpad, 20);

/* write the user's account page to the coprocessor */
coprDevice.writeDataPage(signPageNumber, rawData);

/* write the signing scratchpad */
coprDevice.writeScratchpad(signPageNumber, 0, scratchpad, 0, 32);

/* create the signature */
coprDevice.SHAFunction(OneWireContainer18.SIGN_DATA_PAGE, signPageNumber);

/* match the signature generated by the coprocessor with the user token's MAC */
if( coprDevice.matchScratchpad(dataSignature, 0) )
   System.out.println("Certificate verification Successful!");
```

2.5 Updating the User's Certificate

Updating the user's certificate is only necessary when using dynamic data. If, for example, the data stored in the certificate were simply an identification number, there wouldn't be any need for an update. Authenticating the user token would ensure that it wasn't copied from one token to another and validating the certificate would ensure that it wasn't tampered with. When using dynamic data, as in the example of the account balance, it will be necessary to update the certificate information, restore the initial signature, clear out the CRC16, and resign the data. The breakdown of these steps (with the exception of the updated data) is identical to those presented in section 2.1. The SHATransaction class provides a method for updating (and resigning, if necessary) the transaction data.

Debiting and Updating the User's Certificate Figure 23

```
/* Update user token, with a signed certificate */
SHADebit debit = new SHADebit(copr, 1000, 50);
debit.executeTransaction(user18);
```

3.0 SHA Applications with the 1-Wire Public Domain Kit

The solution for SHA Applications in the 1-Wire Public Domain Kit introduces a few new modules into the development kit. Figure 24 is a listing of the new modules.

SHA API Functions Figure 24

SHA18.C (*Device Layer*)
ReadScratchpadSHA18 – Reads the scratchpad of DS1963S with CRC16 verification.
WriteScratchpadSHA18 – Writes the scratchpad of DS1963S with CRC16 verification.
CopyScratchpadSHA18 – Copies the scratchpad of DS1963S with verification.
MatchScratchpadSHA18 – Matches the contents of the scratchpad with given MAC.
EraseScratchpadSHA18 – Erases the scratch of DS1963S.
ReadAuthPageSHA18 – Performs a Read Authenticated Page command on DS1963S.
ReadMemoryPageSHA18 – Reads a page of memory from DS1963S.
WriteDataPageSHA18 – Writes a page of memory to DS1963S.
SHAFunction18 – Executes a given SHA function on DS1963S (i.e. Sign Data Page, Auth. Host)
InstallSystemSecret18 – Installs a master secret onto DS1963S.
BindSecretToiButton18 – Creates unique DS1963S secret with master secret and binding data.
CopySecretSHA18 – Copies 8-bytes of a SHA result into secret memory.
SHAIB.C (*Protocol Layer*)
SelectSHA – Accesses SHA Device on 1-Wire Net and forces overdrive.
FindNewSHA – Loops to find all SHA Devices on 1-Wire Net while blocking for arrivals.
FindUserSHA – Finds all SHA devices with the specified user account file.
FindCoprSHA – Finds all SHA devices with the specified coprocessor file.
GetCoprFromRawData – Loads service setup data from raw bytes, probably from a file.
CreateChallenge – Creates a random challenge using the Generate Challenge SHA command.
AnswerChallenge – Writes challenge to scratchpad of a user token and reads the account info.
VerifyAuthResponse – Verifies the authentication response of a user token.
CreateDataSignature – Creates a data signature using Sign Data Page SHA command.
SHADEBIT.C (*Transaction Layer*)
InstallServiceData – Formats the user token and installs a new certificate.
UpdateServiceData – Signs certificate information and writes it to the user token.
VerifyUser – Challenges user token with random challenge and verifies the response.
VerifyData – Verifies the certificate signature.
ExecuteTransaction – Debits user's account balance and verifies that user received the update.

In the file SHAIB.H, there are 3 structures that define the coprocessor, the user, and the certificate. Figure 25 shows the format of the certificate.

struct DebitFile Figure 25

```
typedef struct { // See AN151 for certificate details
   uchar fileLength; // length of this file
   uchar dataTypeCode; // data type code - 0x01 for dynamic, 0x02 for static
   uchar signature[20]; // 20-byte data signature for this certificate
   uchar convFactor[2]; // country code and multiplier
   uchar balanceBytes[3]; // account balance
   uchar transID[2]; // transaction ID
   uchar contPtr; // file continuation pointer
   uchar crc16[2]; // crc16 of file
} DebitFile;
```

The SHAUser structure is used to maintain all pertinent information about a specific user token. Also, it maintains state between successive calls to API functions. For example, the accountFile field is populated when verifyUser is called, so the verifyData method uses this information for verifying the certificate signature without re-reading the device.

struct SHAUser Figure 26

```
typedef struct {
   // portnum and address of the device
   int   portnum;
   uchar devAN[8];

   uchar accountPageNumber; // page the user's account file is stored on
   long  writeCycleCounter; // Write cycle counter for account page
   uchar responseMAC[20]; // MAC from Read Authenticated Page command

   union {
      uchar raw[32]; // used for direct writes to button only
      DebitFile file; // use this for accessing individual fields
   } accountFile;
} SHAUser;
```

The SHACopr structure is used to maintain all system parameters. These parameters are, typically, stored in a file on the coprocessor device.

struct SHACopr Figure 27

```
typedef struct {
   // portnum and address of the device
   int   portnum;
   uchar devAN[8];

   uchar serviceFilename[5]; // name of the account file stored on the user token
   uchar signPageNumber; // memory page used for signing data (0 or 8)
   uchar authPageNumber; // memory page used for storing master authentication secret
   uchar wspcPageNumber; // memory page used for storing user's unique secret
   uchar versionNumber; // version number of the transaction system.
   uchar bindCode[7]; // Scratchpad binding data for producing unique secrets
   uchar bindData[32]; // Data page binding data for producing unique secrets
   uchar signChlg[3]; // signature used when signing account data
   uchar initSignature[20]; // challenge used when signing account data
   uchar* providerName; // name of the transaction system provider
   uchar* auxilliaryData; // any other pertinent information
   uchar encCode; // encryption code
   uchar ds1961Scompatible; // indicates that secret was padded for DS1961S
} SHACopr;
```

3.1 Initializing the Coprocessor

Initializing the coprocessor consists of two important steps: installing the system authentication secret and installing the system signing secret. An optional third step is to write all of the system configuration data to the iButton in the form of a file (see AN114). The configuration data doesn't have to be stored on the coprocessor iButton, but it's convenient to keep the system parameters as portable as the coprocessor. As an alternative, the file can be stored on a disk drive or the parameters can be hardcoded in your application.

The following blocks of code demonstrate the steps necessary for initializing the coprocessor. Each block of code is well commented and the declarations, though not necessarily the initialization, of all variables used are shown. The intent is that the code can be copied into a project and compiled after just a few edits. This first block shows the declaration, but not the initialization, of the most important properties.

Necessary System Parameters for a SHA Application Figure 28

```
/* The input data to be used for calculating the master authentication and signing
 * secrets.  The calculation involves splitting the secret into blocks of 47 bytes
 * (32 bytes to the data page, 15 bytes to the scratchpad) and then performing the
 * SHA command Compute First Secret, followed by Compute Next Secret for each 47
 * byte block after the first. */
uchar inputAuthSecret[47], inputSignSecret[47];  // . . . initialize from user input

/* The coprocessor */
SHACopr copr;
copr.portnum = 0;

/* Name of the account file (and file extension) to create on user tokens */
memcpy(copr.serviceFilename, "DLSM", 4);
copr.serviceFilename[4] = 102; // extension for money files

/* The page to use for the signing secret must be page 8, because secret 0 is the
 * only secret that can be used for the SHA command Sign Data Page.  For a list of
 * SHA commands, and what pages they can be used on, see the DS1963S datasheet. */
copr.signPageNumber = 8;

/* The authentication secret can be on any page as long as it doesn't collide with
 * the file holding coprocessor configuration information and not secret 0 */
copr.authPageNumber = 7;

/* The workspace page is the page the coprocessor will use for recreating a user's
 * unique authentication secret and it's authentication response. */
copr.wspcPageNumber = 9;

/* The binding information is used to create the "unique" secret for each button.
 * bindData is written to the data page of the user token, and bindCode is written
 * to the scratchpad.  The result of a Compute Next Secret using the system
 * authentication secret, the account page number, the rom ID of the user token, the
 * bind data, and the bind code becomes the user's unique authentication secret.
 * This is used to initialize a user token and the coprocessor must use the same
 * data to recreate the user token's unique secret. */
copr.bindData = //32 bytes
copr.bindCode = //7 bytes

/* Initial signature to use when signing the certificate */
copr.initSignature = // 20 bytes of user input

/* Three byte challenge used when signing the certificate */
copr.signingChlg = // 3 bytes
```

Although there are a few other parameters for a system that are stored in the service configuration file, these are the most essential parameters. The name of the provider and the version number of the system, for example, are not necessary for a minimal-functionality debit application.

Initializing the Coprocessor Figure 29

```
/* Find the first SHA device on the 1-Wire Net */
FindNewSHA(copr.portnum, copr.devAN, FALSE);

/* Install the master authentication secret and the master signing secret */
InstallSystemSecret18(copr.portnum, copr.signPageNumber, copr.signPageNumber&7,
                      inputSignSecret, 47, FALSE))
InstallSystemSecret18(copr.portnum, copr.authPageNumber, copr.authPageNumber&7,
                      inputAuthSecret, 47, TRUE))


/* prepare the service file to write to the coprocessor */
int namelen = strlen(copr.providerName);
int auxlen = strlen(copr.auxilliaryData);
uchar* coprFile = malloc(80 + namelen + auxlen);
memcpy(coprFile, copr.serviceFilename, 5);
coprFile[5] = copr.signPageNumber;
coprFile[6] = copr.authPageNumber;
coprFile[7] = copr.wspcPageNumber;
coprFile[8] = copr.versionNumber;
memcpy(&coprFile[13], copr.bindData, 32);
memcpy(&coprFile[45], copr.bindCode, 7);
memcpy(&coprFile[52], copr.signChlg, 3);
coprFile[55] = namelen;
coprFile[56] = 20; // length of the initial signature
coprFile[57] = auxlen;
memcpy(&coprFile[58], copr.providerName, namelen );
memcpy(&coprFile[58+namelen], copr.initSignature, 20 );
memcpy(&coprFile[78+namelen], copr.auxilliaryData, auxlen );
coprFile[78+namelen+auxlen] = copr.encCode;
coprFile[79+namelen+auxlen] = copr.ds1961Scompatible;

/* Create FileEntry for "COPR.000" file */
FileEntry feCopr;
memcpy(feCopr.Name, "COPR", 4);
feCopr.Ext = 0;

/* using the 1-Wire file commands in public domain kit, format the device */
int handle, maxwrite;
owFormat(copr.portnum, copr.devAN);
owCreateFile(copr.portnum, copr.devAN, &maxwrite, &handle, &feCopr);
owWriteFile(copr.portnum, copr.devAN, handle, coprFile, 80+namelen+auxlen);
```

3.2 Initializing the User Token

Initializing the user token consists of two steps. First, install the master authentication secret and bind it to the iButton to produce the unique secret for the token. Second, write the certificate file to the iButton. It is actually a little more complicated than it may initially sound. The DS1963S has eight secrets that correspond to the 16 pages of memory (where each secret is shared by two pages). The certificate file must be written to a page which has a write cycle counter, which limits it to the last eight pages of the device's memory banks. Also, the certificate file must be written to one of the pages whose corresponding secret is the secret where the master authentication secret was installed. But, the 1-Wire file API doesn't allow the specification of a page number for the file to be stored on. The best that can be done is to give the file one of the reserved extensions that ensures special treatment. Extensions 101 and 102, for example, are reserved for files that must be written on pages with write cycle counters if the device has them (see AN114). One solution for this is to use the 1-Wire file API to create an empty stub file to write the certificate. This creates the proper directory entries for the certificate so it can be located dynamically.

Then, the page number from the directory entry specifies the page where the master authentication secret should be installed and bound to the user token. When the certificate is actually written to the device, it is done with a direct page write, rather than using the file API. This will allow for much faster updates (very important in a responsive debit application) when actually debiting the token.

Using the SHADEBIT.C module, the installation of the secret as well as the creation of an account file can be done automatically with a single function call. The actual contents of a new certificate file are written to the device after this step.

Initializing User Tokens with SHADEBIT.C Figure 30

```
/* For DS1963S user tokens */
SHAUser user;
user.portnum = 0;
FindNewSHA(user.portnum, user.devAN, FALSE);

/* Install the authentication secret and write a signed certificate to the device */
InstallServiceData(copr, user, inputAuthSecret, 47);
```

Figure 31 illustrates how the master authentication secret is installed on the user token, as well as the actual binding of the master authentication secret to produce an authentication secret that is unique to the user token.

Installing Authentication Secret on DS1963S User Token Figure 31

```
/* Create the empty file on the user token */
FileEntry fe;
memcpy(fe.Name, copr.serviceFilename, 4);
fe.Ext = copr.serviceFilename[4];
owFormat(user.portnum, user.devAN));
owCreateFile(user.portnum, user.devAN, &maxwrite, &handle, &fe));
owCloseFile(user.portnum, user.devAN, handle);

/* File must be created first, so we can get the page number */
user.accountPageNumber = fe.Spage;

/* Install the master authentication secret, same as on the coprocessor */
installSystemSecret18(user.accountPageNumber, inputAuthSecret,
                      user.accountPageNumber&7);

/* format the bind code properly, for format see AN157 */
uchar fullBindCode[15];
memcpy(fullBindCode, copr.bindCode, 4);
fullBindCode[4] = (uchar)user.accountPageNumber;
memcpy(&fullBindCode[5], user.devAN, 7);
memcpy(&fullBindCode[12], &(copr.bindCode[4]), 3);

/* create the unique secret for iButton */
BindSecretToiButton18(user.portnum, user.accountPageNumber,
                      user.accountPageNumber&7,
                      copr.bindData, fullBindCode, TRUE);
```

The final step when initializing a user token is to actually write the account certificate to the data page where the file was stored. The certificate has an option about whether or not it is signed. If left unsigned, the 20 bytes of data where the signature would be stored can be used to store any other useful data that needs to store about the user token. The main situation where there is no need for concern about an

unsigned certificate is when using a DS1961S/DS2432, which requires knowledge of the secret to write to it. Whether signed or unsigned, the format of the certificate remains the same. Figure 32 illustrates how to manually format a certificate file as specified in AN151.

Creating a New Certificate Figure 32

```
/* eCertificate, see format in AN151 */
uchar acctData[32];

acctData[0] = 29; // file length
acctData[1] = 0x01; // data type code or algorithm (0x01 dynamic eCash)

memcpy(acctData, copr.initSignature, 20); // Initial Signature

acctData[22] = 0x8B; acctData[23] = 0x48; // Conversion factor (ISO4217)
acctData[24] = 0xE8; acctData[25] = 0x03; acctData[26] = 0; // Account Balance ($10)
acctData[27] = 0; acctData[28] = 0; // TransactionID
acctData[29] = 0x00; // file continuation pointer
acctData[30] = 0x00; accountData[31] = 0x00; // ~CRC16
```

The certificate's signature is produced on the coprocessor using the Sign Data command. The command has a few variable inputs. The first of which is the data page that is to be signed. In this case, that page is the actual account file (as constructed in Figure 32), which is written to the signing page of the scratchpad. The rest of the inputs are all stored on the scratchpad of the coprocessor (referred to as the coprocessor's "signing scratchpad"). The first parameter stored on the scratchpad is the value of the write cycle counter for the part (incremented by 1, since the data that it will be verified with is about to be written). Next is the page number of the user token's memory pages that will hold the account file. This is followed by least significant 56 bits of the user token's address (64-bit Rom ID minus the CRC8). The last parameter is the 3-byte constant challenge set when the coprocessor was initialized.

Setting up the Coprocessor's Scratchpad for Data Signing Figure 33

```
uchar signScratchpad[32];
int wcc;

/* need to get the value of the write-cycle counter. */
user.writeCycleCounter = ReadAuthPageSHA18(user.portnum, user.accountPageNumber,
                                           user.accountFile.raw, NULL,TRUE);

/* and increment it since we are about to write to the device */
int wcc = user->writeCycleCounter + 1;

/* assign the wcc to the coprocessor's "signing" scratchpad */
signScratchpad[8]  = (wcc&0x0ff);
signScratchpad[9]  = ((wcc>>=8)&0x0ff);
signScratchpad[10] = ((wcc>>=8)&0x0ff);
signScratchpad[11] = ((wcc>>=8)&0x0ff);

/* get the page number of the account file and Rom ID of the user token */
signScratchpad[12] = (byte)user.accountPageNumber;
System.arraycopy(owc18.getAddress(),0,signScratchpad, 13, 7);

/* get the signing challenge */
System.arraycopy(signingChlg, 0, signScratchpad, 20, 3);
```

Signing Data with SHAIB.C Module Figure 34

```
/* sign the data with the coprocessor and set the value of certificate signature */
CreateDataSignature(copr, user->accountFile.raw, signScratchpad,
                    user->accountFile.file.signature, TRUE);
```

The helper method shown in Figure 34 can be broken down into the necessary low-level methods used to implement signing data. Figure 35 illustrates this process for creating a data signature

Signing Data with Coprocessor with SHA18 Module Figure 35

```
int addr = copr.signPageNumber<<5; // physical address of the page
uchar buffer[32];

/* write the account data to the signing page of the coprocessor */
WriteDataPageSHA18(copr.portnum, copr.signPageNumber, user.accountFile.raw, FALSE);

/* write the signScratchpad to the scratchpad of the coprocessor */
WriteScratchpadSHA18(copr.portnum, addr, signScratchpad, 32, TRUE);

/* sign the data and read the signature*/
SHAFunction18(copr.portnum, SHA_SIGN_DATA_PAGE, addr, TRUE);
ReadScratchpadSHA18(copr.portnum, 0, 0, buffer, TRUE);

/* place the resulting signature in the certificate */
System.arraycopy(user.accountFile.signature, &buffer[8], 20);
```

For the data in the account file byte array to be a valid account file, it must have an inverted, two-byte CRC16 at the end of the file. More detail on what makes a valid file on the 1-Wire File Structure can be found in AN114.

Adding Inverted CRC16 to 1-Wire File Figure 36

```
/* calculate the inverted CRC16 */
setcrc16(user.portnum, user.accountPageNumber);
for (i = 0; i < 30; i++)
   crc16 = docrc16(user.portnum,user.accountFile.raw[i]);
crc16 = ~crc16;

/* now the file is ready to be written */
user.accountFile.file.crc16[0] = (uchar)crc16;
user.accountFile.file.crc16[1] = (uchar)(crc16>>8);
```

Writing the account file to the data page of the DS1963S user token is fairly easy. There is a utility function provided for writing a page of data, the exact function which is used to write the account data to the coprocessor for signing in the above code block.

3.3 Authenticating the User Token

Authenticating the user token involves a simple challenge-response scheme. First, the coprocessor is used to generate a 3-byte, pseudo-random challenge. This challenge is then written to the scratchpad of the user token and the Read Authenticated Page command is issued (see DS1963S data sheet). This returns the entire contents of the memory page followed by the value of the write cycle counter for the page and the write cycle counter for the secret location. The scratchpad of the DS1963S user token will contain the 20-byte SHA result of the user's unique authentication secret, the page number the data was read from,

the serial number of the user token, and the random challenge. The coprocessor is then used to reproduce this SHA result, ensuring that the user token is a valid member of the system.

Authenticating User Token with SHADEBIT.C Module Figure 37

```
/* Verify user tokens authentication response, same for signed and unsigned */
VerifyUser(copr, user, TRUE);
```

The process for verifying a user is the same for both signed and unsigned transactions. Naturally, this can be broken down into simple steps using the SHAIB.C module. The code snippet presented in Figure 38 demonstrates the basic implementation of the user verification method in the protocol-layer module (minus the error checking).

Authenticating User Token with SHAIB.C Module Figure 38

```
uchar chlg[3]; // random challenge bytes
/* Use coprocessor to generate a random challenge */
CreateChallenge(copr, copr.signPageNumber, chlg, 0);

/* issue challenge to user getting back the account data, response MAC, and the
 * value of the write-cycle counter */
AnswerChallenge(user, chlg);

/* use coprocessor to verify the authentication response */
VerifyAuthResponse(copr, user, chlg, TRUE);
```

Figure 39 illustrates how to use the lowest-level module (SHA18.C) to produce a random challenge.

Using the Coprocessor to Generate a Random Challenge Figure 39

```
uchar scratchpad[32]; // temporary buffer
uchar chlg[3];

/* Use the coprocessor to generate the challenge, page number is irrelevant but a
 * highly used page will generate a more random (less repeating) number */
EraseScratchpadSHA18(copr.portnum, 0, FALSE);
SHAFunction18(copr.portnum, SHA_COMPUTE_CHALLENGE, copr.signPageNumber<<5, TRUE);
ReadScratchpadSHA18(copr.portnum, 0, 0, scratchpad, TRUE);

/* copy the challenge bytes into challenge buffer */
memcpy(chlg, &scratchpad[20], 3);
```

After performing the steps in Figure 39, the scratchpad buffer of the coprocessor now contains the 20-byte result of a SHA calculation, starting at index 8. Since any three bytes are as good as any other three for a challenge, it is safe to leave the result as it is in the scratchpad. Indices 20 to 22 of the coprocessor's scratchpad hold the challenge bytes that will be used for the Read Authenticated Page. Using these three particular bytes makes it unnecessary to write the challenge back to the coprocessor later.

Answering a Random Challenge with a DS1963S User Token Figure 40

```
int acctAddr = user.accountPageNumber<<5; //physical address of account file
uchar scratchpad[32];
memcpy(&scratchpad[20], chlg, 3);

/* Write the challenge to the scratchpad of the user token */
EraseScratchpadSHA18(user.portnum, acctAddr, FALSE);
WriteScratchpadSHA18(user.portnum, acctAddr, scratchpad, 32, TRUE);

/* perform authenticated read to get the page data and the resulting MAC */
user.writeCycleCounter =
         ReadAuthPageSHA18(user.portnum,
                           user.accountPageNumber,
                           user.accountFile.raw,
                           user.responseMAC, TRUE);
```

After retrieving the MAC (Message Authentication Code) from the user token, it is now the job of the coprocessor to verify it. To verify the response, the user token's unique secret is recreated in the workspace secret of the coprocessor and the raw account data is signed with this workspace secret.

Validating the Authentication Response Figure 41

```
int wcc = user.writeCycleCounter;

/* Bind DS1963S user token's unique secret to coprocessor */
uchar fullBindCode[15];

/* Get the 7-byte binding code
memcpy(fullBindCode, copr.bindCode, 4);
memcpy(&fullBindCode[12], &copr.bindCode[4], 3);

/* get the page number of the account file and 7 bytes of the ROM ID */
fullBindCode[4] = user.accountPageNumber;
memcpy(&fullBindCode[5], user.devAN, 7);

/* recreate user token's unique secret in workspace secret*/
BindSecretToiButton18(copr.authPageNumber, copr.bindData, fullBindCode,
                      copr.wspcPageNumber);

/* the scratchpad of the coprocessor now needs the user's ROM ID and page number,
 * In addition to the challenge bytes used and the write-cycle counter. */
memcpy(&scratchpad[12], fullBindCode[4], 8);
memcpy(&scratchpad[20], chlg, 3);
scratchpad[8]  = (wcc&0x0ff);
scratchpad[9]  = ((wcc>>=8)&0x0ff);
scratchpad[10] = ((wcc>>=8)&0x0ff);
scratchpad[11] = ((wcc>>=8)&0x0ff);

/* write to the coprocessor and validate */
int wspcAddr = copr.wspcPageNumber<<5; //physical address of wspc page
WriteDataPageSHA18(copr.portnum, copr.wspcPageNumber, user.accountFile.raw, FALSE);
WriteScratchpadSHA18(copr.portnum, wspcAddr, scratchpad, 32, TRUE);
SHAFunction18(copr.portnum, SHA_VALIDATE_DATA_PAGE, wspcAddr, TRUE);

if( MatchScratchpadSHA18(copr.portnum, user.responseMAC, TRUE) )
   printf("DS1963S Authentication Successful!");
```

3.4 Validating the User's Certificate

Validating a certificate from a user token looks a lot like the earlier section on initializing the data. The steps for signing the data are very similar, but the coprocessor-generated signature is merely matched against the existing signature, rather than being read into the certificate. Also, since the entire certificate is merely stored on a file in any user token, the process for validating the certificate is identical across all tokens. When using an unsigned certificate, this step may only consist of checking the user's account balance to make sure it makes sense in the system (i.e. it isn't negative or it isn't a trillion dollars).

Validating Transaction Data with SHADEBIT.C Module Figure 42

```
/* Verify user tokens data, including verifying the data signature */
VerifyData(&copr, &user);
```

For signed transaction data, the signature of the certificate must be verified with the coprocessor. As a nice initial check, the validity of the account balance information can be verified first. This can provide a quick check to avoid verifying the signature of the account information if it's unnecessary.

Validating the Account Balance Information Figure 43

```
/* verify the user's account balance is 'legal' */
int balance = (user.accountFile.file.balance[26]&0x0ff);
balance = (balance << 8) | (user.accountFile.file.balance[25]&0x0ff);
balance = (balance << 8) | (user.accountFile.file.balance[24]&0x0ff);

/* MAX_BALANCE, MIN_BALANCE are constant integers */
if( balance>MAX_BALANCE )
   printf("Too Much Money!");
else if( balance<MIN_BALANCE )
   printf("Not enough money to perform transaction!");
```

The verification of the signature looks very similar to how the signature was created in the first place. The CRC16 is cleared out (set to zero) and the data signature, after being saved in a buffer, is cleared out and replaced with the system's initial signature. Since the new signature will not be written back to the part and the current signature is being verified against the current value of the write cycle counter, the write cycle counter is not incremented this time for producing the data signature. The entire process is shown in Figure 44.

Note that, rather than reading the signature back and matching the signature bytes in software, the coprocessor's Match Scratchpad command is used. Since this command only transmits 20 bytes of MAC information to the coprocessor, rather than reading 32 bytes of scratchpad information from the coprocessor, this technique is slightly faster. When building a fast, responsive debit application, all non-essential reads/writes will have to be optimized in this manner. In addition, making sure that the correct MAC isn't transmitted from the coprocessor gives an invalid user token no opportunity to grab the right MAC. This will leave the application less open to holes where the user can take advantage of system "retries." For example, if the system was designed to retry the user challenge operation if interrupted, but eventually repeated a 3-byte random challenge, the user could have stored the appropriate signature.

Validating the Certificate Signature Figure 44

```
int signAddr = copr.signPageNumber<<5; // physical address of signing page
int wcc = user.writeCycleCounter;
uchar scratchpad[32];

/* save the data signature */
uchar dataSignature[20];
memcpy(dataSignature, user.accountFile.file.signature, 20);

/* clear out the old signature and CRC 16 */
memcpy(user.accountFile.file.signature, copr.initSignature, 20);
user.accountFile.file.crc16[0] = 0x00;
user.accountFile.file.crc16[1] = 0x00;

/* assign the wcc to coprocessor's "signing" scratchpad (DS1961S wcc=0xFFFFFFFF) */
scratchpad[8]  = (wcc&0x0ff);
scratchpad[9]  = ((wcc>>=8)&0x0ff);
scratchpad[10] = ((wcc>>=8)&0x0ff);
scratchpad[11] = ((wcc>>=8)&0x0ff);

/* setup the rest of the "signing" scratchpad */
scratchpad[12] = user.accountPageNum;
memcpy(&scratchpad[13], user.devAN, 7);
memcpy(&scratchpad[20], copr.signChlg, 3);

owSerialNum(copr.portnum, copr.devAN, FALSE);
/* write the user's account page to the coprocessor */
WriteDataPageSHA18(copr.portnum, copr.signPageNumber, user.accountFile.raw, FALSE);
/* write the signing scratchpad */
WriteScratchpadSHA18(copr.portnum, signAddr, scratchpad, 0, 0, TRUE);
/* create the signature */
SHAFunction18(copr.portnum, SHA_SIGN_DATA_PAGE, signAddr, TRUE);

/* match the signature generated by the coprocessor with the user token's MAC */
if(MatchScratchpadSHA18(copr.portnum, dataSignature, TRUE) )
   printf("Certificate verification Successful!");
```

3.5 Updating the User's Certificate

Updating the user's certificate is only necessary when using dynamic data. If, for example, the data stored in the certificate were simply an identification number, there wouldn't be any need for an update. Authenticating the user token would ensure that it wasn't copied from one token to another and validating the certificate would ensure that it wasn't tampered with. When using dynamic data, as in the example of the account balance, it will be necessary to update the certificate information, restore the initial signature, clear out the CRC16, and resign the data. The breakdown of these steps (with the exception of the updated data) is identical to those presented in section 2.1. Figure 45 shows the function for updating (and re-signing if necessary) the transaction data.

Debiting and Updating the User's Certificate Figure 45

```
/* Update user token after subtracting 50 cents, with a signed certificate.
 * verify that the user token was updated with another read authenticated page. */
ExecuteTransaction(copr, user, 50, TRUE);
```