maxim integrated™

应用笔记 156
DS1963S SHA 1-Wire API 用户指南

1.0 绪论

本文的目的是让 1-Wire®软件开发人员熟悉安全SHA应用中的API。在两个主要的开发包：Java™ 1-Wire API和 1-Wire公用程序包中均可得到这个API。本文既可以作为用各种API设计新的安全系统的应用指南，也有助于深入理解各种开发包配备的演示系统。

本文假定用户对SHA iButton®硬件和iButton采用的 1-Wire协议有了基本理解。DS1963S (SHA iButton)数据资料可从网站(www.maxim-ic.com.cn)下载，还可以从网上获得详述eCash (电子支付系统)签名证书结构的应用笔记[AN151, *Maxim Digital Monetary Certificates*]、用于安全系统的高层协议[AN157, *SHA iButton API概述*]、SHA-1 概述[AN1201, *1-Wire SHA-1 概述*]、以及适合于存储器件的文件系统的实现[AN114, *1-Wire File Structure*]。

在任何一个货币 SHA 应用中，协处理器和用户令牌(user token)是两个主要组成部分。协处理器是一个初始化后的 DS1963S，用于验证用户令牌是否为系统成员、确认用户的证书。用户令牌是装载货币证书、并识别系统用户的 DS1963S (或类似的 1-Wire 器件)。对于每一个 API，本文将简要介绍其初始化协处理器、初始化用户令牌和实现交易的方法。每个交易过程都可以进一步细化，包括用户鉴别、验证交易数据、根据动态信息更新交易数据。

在本文所有的代码示例中，底色是浅灰色的代码框内将高层任务拆分成更详细的操作条目。在 Java 1-Wire API 中，这些低层代码将使用 OneWireContainer18 类。在公用程序包中，这些代码将使用 SHA18.C 模块中的函数。

2.0 Java 1-Wire API 的 SHA 应用

Java 1-Wire API 中的 SHA 解决方案将一个新的软件包引入开发包系统：com.dalsemi.onewire.application.sha。图 1 给出了这个包的类，并且说明了它们之间的继承关系。在详述该包中所介绍的方法的必要特性时，容器级方法被认为是最基本的单元，不再进行细化(虽然这些资源可以在包中得到)。

SHATransaction 摘要父类用于描述所有的安全交易。该类定义了三个关键方法，描述一个典型交易的基本步骤：

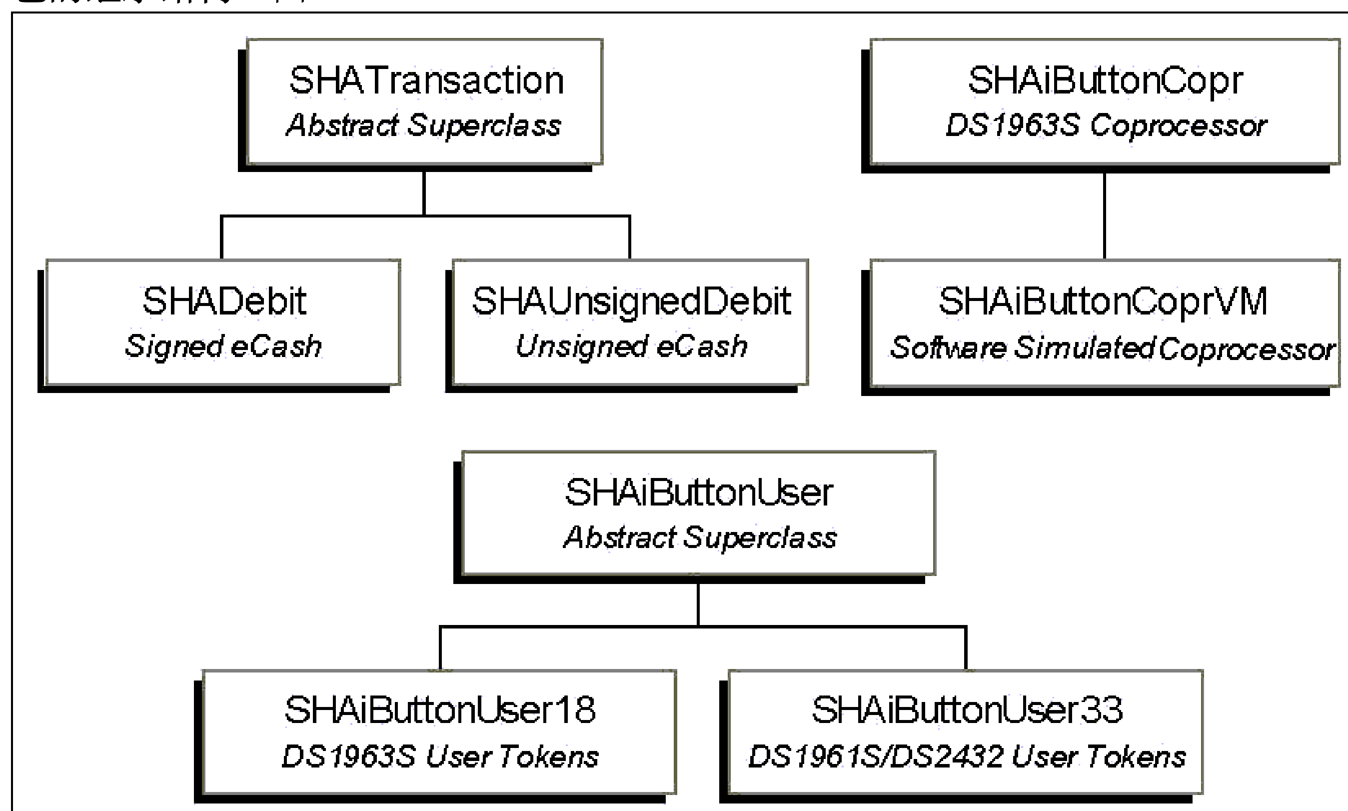
1) 验证用户令牌是否为系统的有效成员(verifyUser)
2) 验证数据是否经过适当鉴定、没有被窜改(verifyTransactionData)
3) 必要时更新数据并确保令牌接受更新(executeTransaction)

1-Wire 和iButton 是Dallas Semiconductor的注册商标。
Java 是 Sun Microsystems 的商标。

有两个交易样本扩展了 SHATransaction：SHADebit 和 SHADebitUnsigned。前者(SHADebit)建立了一个借方账户系统，将初始账目余额存储在签名的货币证书中(参见 AN151)。典型应用中，要验证用户是否是系统的有效成员，还要验证证书上的签名是否有效。如果用户和数据都是有效的，则更新存储在货币证书上的余额，生成新的签名，而且证书也被写回到器件中。后者(SHADebitUnsigned)用未签名的货币证书，利用空闲空间来实现一个备份方案，保护 EEPROM 器件中的数据。在另一个应用笔记中谈及了这种交易类型(用 DS1961S/DS2432 1-Wire 器件)。

SHA 交易将 SHAiButtonCopr 作为一个固定的成员(该类描述的是一个初始化了的 DS1963S，在 SHA 交易中用作协处理器)。从这个意义上说，SHATransaction 可以看成是 SHAiButtonCopr 的扩展。它的功能和 Java 中的“decorator”类相类似，因为它把“要签名的内容”和“正在签名的内容”区分开。这样，软件开发人员只需进行少量编码就可以很容易地把“要签名的内容”(即：协处理器，SHAiButtonCopr)应用到不同的系统中去(“正在签名的内容”)。另外，SHAiButtonCoprVM，或者说是仿真协处理器，可以用来代替硬件 DS1963S。提出该类是因为它具有一定的便利条件，但是为优化安全性推荐使用硬件协处理器。

包的继承结构　图 1



SHATransaction中的每一种方法都把SHAiButtonUser作为一个参数。API提供给两类用户对象：SHAiButtonUser33 和SHAiButtonUser18。虽然SHAiButtonUser默认设置只支持两类SHA iButton (家族码为 0x33 的DS1961S/DS2432 和家族码为 0x18 的DS1963S)，但是SHATransaction的用户概念可以扩展成支持所有的 1-Wire存储器件。 由于SHAiButtonUser33 和SHAiButtonUser18 只是提供承载的必要帐务数据证明，所以它们都易于扩展。如果iButton验证对特定的系统不很重要，那么SHAiButtonUser类可以很容易地扩展为支持那些承载有签名账户信息的 1-Wire存储器。需要注意即使其它存储器件不采用鉴别iButton的质询应答协议，用户令牌的 64 位ROM ID仍具有较低的安全性。

2.1 初始化协处理器

初始化协处理器有两个重要的步骤：安装系统鉴别密钥和安装系统签名密钥。第三步是可选的，就是把所有的系统配置数据以文件的形式写入iButton (参见AN114)。这些配置数据不是必须要保存在协处理器iButton上，但这样做可以方便地使系统参数像协处理器一样具有便携特性。还有一种可供选择的方法是把文件保存在磁盘驱动器上或者把参数硬编码到应用程序中。

下面的代码用于示范协处理器的初始化。每段代码都给出了详细的注释，并对所有用到的变量进行了声明。这样，只需对其进行很少的编辑，就可以把它们应用到其它系统中。

SHA 应用的必要的系统参数　图 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 its 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
```

一个系统还有一些其它最基本的参数存储在服务配置文件中。例如，提供者的姓名和系统的版本号，但它们对于最小功能的借方应用来说不是必要的。可以调用 SHAiButtonCopr 构造器对协处理器进行初始化，把这些参数作为变量赋值。至于更详细的全部参数列表，请参阅 Java 1-Wire API 的 JavaDocs。

使用 SHAiButtonCopr 初始化协处理器　图 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);
```

只用 OneWireContainer18 类就能够初始化容器、无需使用 com.dalsemi.onewire.application.sha 包。下面代码说明了只使用 OneWireContainer18 把密钥安装到器件的步骤。需要注意的是这段代码和图 3 中的 SHAiButtonCopr 构造器的实际执行过程几乎相同。

协处理器的容器级初始化 图 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 */
OWFileOutputStream 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”);
```

如果SHA协处理器是用于鉴别DS1961S/DS2432 iButton，或者用于产生复制暂存器的授权，需要截取主鉴别密钥以适应DS1961S较小的暂存器。这里提供了一个可适当截取密钥的函数。随后协处理器文件中的“DS1961S 兼容性标记”应当被置为非零值。

截取验证密钥　图 5

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

2.2 初始化用户令牌

初始化用户令牌分两步。第一步：安装主鉴别密钥并将其绑定到iButton上，以产生唯一的令牌密钥。第二步：向iButton中写入证书文件。实际操作要比最初听起来复杂一些。DS1963S的存储器有十六页，相应的有八个密钥(每两页共用一个密钥)。证书文件必须写到具有写次数计数器的页中，这样就将其限制在器件内存区的最后八页中。同时，写证书文件页的密钥应该就是安装的主鉴别密钥。但是，1-Wire文件API并不允许存储文件中按页码的说明。最好的办法就是给文件一个用于确保进行特殊处理的保留扩展名。例如，扩展名 101 和 102 就是为文件所保留的，如果器件有扩展名为 101 和 102 的文件，这些文件必须被写到具有写次数计数器的页中(参见AN114)。一种解决方案就是利用 1-Wire文件API创建一个空的根目录用来写证书，为证书建立一个适当的目录路径使其能够被动态定位。然后，来自目录路径的页码指明主鉴别密钥应当安装在哪一页并绑定到用户令牌。当证书被写入器件时，它实际上是直接写入页的，而不是利用文件API。这就使得在实际记录令牌时可以快速更新(这在应答式借方应用中是非常重要的)。

使用 SHAiButtonUser18 类时，密钥的安装和空账户文件的建立都可以通过调用构造器自动完成。文件的实际内容是在这一步之后写入的(也就是说仍需写入签名证书)。

使用 SHAiButtonUser 初始化用户令牌　图 6

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

图 7 说明了主鉴别密钥是如何安装到用户令牌上的，以及主鉴别密钥如何绑定以产生用户令牌的唯一密钥。

用户令牌中验证密钥的容器级安装　　图 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 its unique secret */
owc18.bindSecretToiButton(acctPage, copr.getBindData(), fullBindCode, acctPage&7);
```

初始化用户令牌的最后一步是把账目证书写入存储文件的数据页。该证书有一个关于是否有签名的选项。使用 DS1961S/DS2432 时不必担心无签名的证书，因为这时需要知道写操作的密钥。

使用 SHADebit 安装签名证书　图 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
```

无论有签名、还是无签名，证书的格式是相同的。如果证书设置是无签名的，那么为 MAC 所保留的 20 个字节就能用来存储系统的任意特定数据。图 9 说明了人为地建立 AN151 中指定的鉴别文件格式。

建立新证书　图 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
```

如果证书的数据需要签名，通常使用协处理器对账目数据进行签名。对于 DS1963S 用户令牌来说，在将证书写入数据页之前，有必要先读取写次数计数器的值。因为待校验的数据就是要写的

数据，所以写计数器的当前值加 1。

协处理器利用 Sign Data 命令可以产生证书签名。该命令只有少数输入变量。首先是要签名的数据页。这种情况下，这个数据页实际上就是账务文件(如图 9 所构造的那样)，它被写到暂存器的签名页。其余输入都存储在协处理器的暂存器中(参照协处理器的“签名暂存器”)。暂存器中的第一个参数是器件的写次数计数器值。第二个是用户令牌存储页的页码，该页用来存储账务文件。随后是 56 位用户令牌地址(64 位 ROM ID 减去 CRC8 校验码)。最后一个参数是协处理器初始化时 3 个字节的激励质询码。

从 DS1963S 中检索写次数计数器　图 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
```

为签名数据设置协处理器暂存器　图 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);
```

签名数据 SHAiButtonCopr 的服务方法　图 12

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

图 12 中介绍的服务类方法可以细化为必要的容器方法来完成签名数据。图 13 说明了建立数据签名的过程。

在容器层使用协处理器的签名数据　图 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);
```

要使账目文件字节阵列中的数据是有效的账目文件，在文件的末尾必须是码求反后的、两个字节的 CRC16 校验码。关于在 1-Wire 文件结构上的有效文件的详细阐述可参考 AN114。

为 1-Wire 文件添加求反的 CRC16 校验码　图 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);
```

把账目文件写到 DS1963S 用户令牌的数据页是相当容易的。可利用容器类的一个函数写一页数据，上述代码就是把账目数据写到协处理器中进行签名的代码。

2.3 鉴别用户令牌

验证用户令牌涉及到一个简单的质询应答方案。首先，利用协处理器来产生 3 个字节的伪随机激励质询码。然后这个请求被写入用户令牌的暂存器内，发出 Read Authenticated Page (读鉴别页)命令(参见 DS1963S 数据资料)。这将返回存储页的全部内容，随后是页的写次数计数器的值和记录密钥位置的写次数计数器的值。DS1963S 用户令牌的暂存器将含有 20 个字节的 SHA 结果，其中包括：用户唯一的鉴别密钥、所读数据的页码、用户令牌的序列号和随机质询。随后协处理器一般将再生成这个 SHA 结果，以确保用户令牌是系统的有效成员。

利用 SHADebit 验证用户令牌　图 15

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

验证用户的过程对有签名和无签名的交易来说是一样的。从本质上，它可以利用 SHAiButtonCopr 类和 SHAiButtonUser 类细分成一些简单步骤。图 16 中的代码表示在交易类进行用户验证的基本实现方法(无验证差错)。

利用 SHAiButtonCopr 和 SHAiButtonUser 进行验证　图 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!");
```

图 17 描述如何利用 OneWireContainer18 类产生随机质询。同时，也对本章中其它代码中用到的变量进行了说明。

利用协处理器产生随机质询　图 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);
```

执行了图 17 所示步骤后，协处理器的暂存缓冲器中就包含了 20 个字节的SHA计算结果，从索引 8 开始。任意三个字节可作为激励质询码，如果保持与暂存器相同的结果会更安全。协处理器的暂存器索引号为 20 到 22 的单元用于保存质询字节，它们被用于Read Authenticated Page (读验证页)。利用这三个特殊的字节就无需在后面把质询写回到协处理器。

利用 DS1963S 用户令牌响应随机质询　图 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 resulting SHA computation and copy it into a buffer*/
owc18.readScratchpad(scratchpad, 0);
System.arraycopy(scratchpad, 8, responseMAC, 0, 20);
```

从用户令牌提取到 MAC (消息鉴别代码)后，协处理器就要对它进行验证。为了验证响应，在协处理器的工作区密钥中需重建用户令牌的唯一密钥，原始的账目数据则经过工作区密钥签名。

验证鉴定响应　图 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 验证用户证书

通过用户令牌验证证书看起来很像前面的数据初始化。这一步对数据签名而言非常相似，但协处理器产生的签名只是与现有的签名进行匹配，而不是被读入证书。同时，因为整个证书都保存在用户令牌的一个文件上，所以验证证书的过程对所有令牌来说都是一样的。使用无签名的证书时，这一步只是校验用户的账目余额，以确保它在系统中是有意义的(即不是负数或几万亿美元的数字)。

利用 **SHADebit** 验证交易数据　图 20

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

为了在交易数据上签名，证书上的签名必须经过协处理器的验证。妥善的初始验证应该首先验证账目余额信息的有效性。这可以避免在不必要的时候对账目信息的签名进行验证，以实现快速的校验。

验证账目余额信息　图 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!");
```

验证签名和怎样创建签名非常相似。CRC16 校验码被清除(置为 0 电平)，数据签名存入缓冲区以后也被清除，并代之以系统的初始签名。因为新的签名不再写回到这一部分，而当前的签名又要通过写次数计数器的值来进行验证，所以，在产生数据签名的这段时间里，写次数计数器并没有递增。全部过程见图 22。

注意，此处使用的是协处理器的 Match Scratchpad 命令，而不是在软件中读出签名并匹配签名字节。因为这个命令只向协处理器传送 20 个字节的 MAC 信息，而不是从协处理器读取 32 个字节的暂存器信息，所以这种方法稍微快一点。当建立一个快速、可应答的借方应用时，所有不必要的读/写操作都必须按照这种方式进行优化。另外，也要确保正确的 MAC 不要从协处理器传送出来，保证无效的用户令牌没有机会获得正确的 MAC。以上这些方法可以使应用的安全漏洞比较少，也就是用户可以利用系统“重试”的机会比较少。例如，如果系统在中断时允许用户重试质询操作，它实质上是重复 3 个字节的随机激励质询码，这时用户有可能得到存储的正确签名。

证书签名的容器级验证　图 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 更新用户证书

只有在使用动态数据的时候才有必要进行用户证书的更新。例如，如果存在证书中的只是简单的识别号，就没有必要进行更新。验证用户令牌能够保证该令牌不是另一个的副本，而验证证书则能够保证它没有被窜改。正如上述账目余额的例子，使用动态数据的时候，更新证书信息、恢复初始签名、清除 CRC16 校验码和重签数据将是必要的。这些步骤的细化(除了更新数据以外)和 2.1 节中的叙述相同。SHATransaction 类提供了一种更新(如果必要需重新签名)交易数据的方法。

借记和更新用户证书　图 23

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

3.0 具有 1-Wire 公用程序包的 SHA 应用

SHA 应用的解决方案在 1-Wire 公用程序包中引入了许多新的模块。图 24 列出了这些新的模块。

SHA API 函数　图 24

SHA18.C (*设备层*)
ReadScratchpadSHA18 一读 DS1963S 的暂存器，并进行 CRC16 校验。
WriteScratchpadSHA18 一写 DS1963S 的暂存器，并进行 CRC16 校验。
CopyScratchpadSHA18－复制 DS1963S 的暂存器，并进行校验。
MatchScratchpadSHA18－匹配暂存器和给定的 MAC 的内容。
EraseScratchpadSHA18－清除 DS1963S 的暂存器。
ReadAuthPageSHA18－执行 DS1963S 上的 Read Authenticated Page 命令。
ReadMemoryPageSHA18－从 DS1963S 读一页存储器
WriteDataPageSHA18－向 DS1963S 写一页存储器。
SHAFunction18－在 DS1963S 上执行一个给定的 SHA 函数(即 Sign Data Page、Auth. Host)。
InstallSystemSecret18－把主密钥安装到 DS1963S 上。
BindSecretToiButton18－创建有主密钥和绑定数据的唯一的 DS1963S 密钥。
CopySecretSHA18－将 8 个字节的 SHA 结果复制到密钥存储器。
SHAIB.C (*协议层*)
SelectSHA 一在 1-Wire 网上访问 SHA 器件，并强制其高速运行。
FindNewSHA－1-Wire 网上添加新器件时，循环执行直到找出所有 SHA 器件。
FindUserSHA－找出所有含指定用户账务文件的 SHA 器件。
FindCoprSHA－找出所有含指定协处理器文件的 SHA 器件。
*GetCoprFromRawData*一从原始字节、也可能是从文件装载服务安装数据。
CreatChallenge－利用 Generate Challenge SHA 命令产生一个随机质询。
AnswerChallenge－把激励质询码写入用户令牌的暂存器，并读取账务信息。
VerifyAuthResponse 一验证用户令牌的鉴定响应。
*CreateDataSignature*一利用 Sign Data Page SHA 命令产生一个数据签名。
SHADEBIT.C (*传输层*)
InstallServiceData－安排用户令牌的格式并安装一个新的证书。
UpdateServiceDate－在证书信息上签名并将其写入到用户令牌。
VerifyUser 一询问有随机质询的用户令牌并验证其响应。
VerifyData－验证证书的签名。
*ExecuteTransaction*一记录用户的账目余额，并确认用户得到了更新。

在文件 SHAIB.H 中有三个结构，分别定义了协处理器、用户和证书。图 25 说明了证书格式。

struct DebitFile　图 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;
```

SHAUser 结构用来保存特定用户令牌的所有相关信息。同时它也保存连续两次调用 API 函数之间的状态。例如，调用 verifyUser 函数时 accountFile 域被驻留，所以 verifyData 利用这一信息，可以不用重读器件就可以验证证书签名。

struct SHAUser 图 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;
```

SHACopr 结构用来保存所有的系统参数。这些参数一般保存在协处理器设备的一个文件中。

struct SHACopr 图 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 初始化协处理器

初始化协处理器有两个重要的步骤：安装系统验证密钥和安装系统签名密钥。第三步是可选的，就是把所有的系统配置数据以文件的形式写入iButton (参见AN114)。这些配置数据不一定必须保存在协处理器iButton上，但这样做可以方便地使系统参数像协处理器一样具有便携特性。还有一种可供选择的方法是把文件保存在磁盘驱动器上或者把参数硬编码到应用程序中。

下面这几段代码是初始化协处理器的必要步骤。每段代码都有详细的注释，并对所有用到的变量进行了声明，尽管这对于初始化来说并不必要。这样，只需对其进行很少的编辑，就可以把它们应用于其它系统。第一段代码是对最重要性质的声明，而不是初始化。

SHA 应用的必要系统参数　图 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
```

一个系统还有一些其它最基本的参数存储在服务配置文件中。例如，提供者的姓名和系统的版本号，但它们对于最小功能的借方应用来说不是必要的。

初始化协处理器　图 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 初始化用户令牌

初始化用户令牌分两步。第一步：安装主鉴别密钥并将其绑定到iButton上，以产生唯一的令牌密钥。第二步：向iButton中写入证书文件。实际操作要比最初听起来复杂一些，DS1963S的存储器有十六页，相应的有八个密钥(每两页共用一个密钥)。证书文件必须写到具有写次数计数器的页中，这样就将其限制在器件内存区的最后八页中。同时，写证书文件页的密钥应该就是安装的主鉴别密钥。但是，1-Wire文件API并不允许存储文件中按页码的说明。最好的办法就是给文件一个用于确保进行特殊处理的保留的扩展名。例如，扩展名 101 和 102 就是为文件所保留的，如果器件有扩展名为 101 和 102 的文件，那么这些文件必须被写到具有写次数计数器的页中(参见AN114)。一种解决方案就是利用 1-Wire文件API来创建一个空的根目录用来写证书，为证书建立一个适当的目录路径使其能够被动态定位。然后，来自目录路径的页码指明主鉴别密钥应当安装

在哪一页并绑定到用户令牌。当证书被写入器件中时，它实际上是直接写入页的，而不是利用文件API。能够在实际记录令牌时快速更新(这在应答式借方应用中是非常重要的)。

通过使用 SHADEBIT.C 模块，密钥的安装和账目文件的创建可以通过调用一个单独函数完成。实际上新的证书文件的内容是在这步之后写入器件的。

利用 SHADEBIT.C 初始化用户令牌　图 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);
```

图 31 说明主验证密钥是如何安装到用户令牌上，以及主验证密钥是如何绑定以产生用户令牌唯一的鉴定密钥。

在 DS1963S 用户令牌上安装鉴定密钥　图 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);
```

初始化用户令牌的最后一步是把账目证书写入存储文件的数据页。该证书有一个关于是否有签名的选项。如果没有签名，那么用于存储签名的 20 个字节可以用来存储任何其它需要存储的关于用户令牌的有用数据。使用 DS1961S/DS2432 时需要知道写操作的密钥，不必担心无签名的证书。不管是否有签名，证书的格式保持相同。图 32 说明了如何规划 AN151 中所指定的证书文件格式。

建立一个新的证书 图 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
```

协处理器利用 Sign Data (签名数据)命令可以产生证书签名。该命令只有少数输入变量。首先是要签名的数据页。这种情况下，这个数据页实际上就是账务文件(如图 32 所构造的那样)，它被写到暂存器的签名页。其余输入都存储在协处理器的暂存器中(参照协处理器的"签名暂存器")。暂存器中的第一个参数是器件的写次数计数值(加 1，因为进行验证的数据将被写入)。第二个是用户令牌存储页的页码，该页用来存储账务文件。随后是 56 位用户令牌地址(64 位 ROM ID 减去 CRC8 校验码)。最后一个参数是协处理器初始化时产生的 3 个激励质询字节。

为数字签名设置协处理器的暂存器 图 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);
```

使用 **SHAIB.C** 模块的签名数据　图 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);
```

图 34 所示的服务方法可以细分成几个必要的用于实现数据签名的低层方法。图 35 说明了产生数据签名的过程。

使用 **SHA18** 模块的协处理器的签名数据　图 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);
```

要使账目文件字节阵列中的数据是有效的账目文件，在文件的末尾必须是码求反后的、两个字节的 CRC16 校验码。关于在 1-Wire 文件结构上的有效文件的详细阐述可参考 AN114。

为 **1-Wire** 文件添加求反后的 **CRC16**　图 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);
```

把账目文件写到 DS1963S 用户令牌的数据页是相当容易的。可利用容器类的一个函数写一页数据，上述代码实际上是把账目数据写到协处理器中进行签名。

3.3 鉴别用户令牌

验证用户令牌涉及到一个简单的质询应答方案。首先，利用协处理器来产生 3 个字节的伪随机激励质询码。然后这个请求被写入用户令牌的暂存器内，发出 Read Authenticated Page (读鉴别页)命令(参见 DS1963S 数据资料)。这将返回存储页的全部内容，随后是页的写次数计数器的值和记录密钥位置的写次数计数器的值。DS1963S 用户令牌的暂存器将含有 20 个字节的 SHA 结果，其中包括：用户唯一的鉴别密钥、所读数据的页码、用户令牌的序列号和随机质询。然后协处理器一般将再次生成这个 SHA 结果，以确保用户令牌是系统的有效成员。

使用 SHADEBIT.C 模块鉴定用户令牌　图 37

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

验证用户的过程对有签名和无签名的交易来说是一样的。从本质上，它可以利用 SHAIB.C 模块细分成一些简单步骤。图 38 中的代码表示在交易类进行用户验证的基本实现方法(无验证差错)。

使用 SHAIB.C 模块鉴定用户令牌　图 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);
```

图 39 演示了如何使用最低层模块(SHA18.C)产生随机质询。

使用协处理器产生随机激励质询码　图 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);
```

执行了图 39 所示步骤后，协处理器的暂存缓冲器中就包含了 20 个字节的SHA计算结果，从索引 8 开始。任意三个字节可作为激励质询码，如果保持与暂存器相同的结果会更安全。协处理器的暂存器索引号为 20 到 22 的单元存保存质询字节，它们被用于Read Authenticated Page (读验证页)。利用这三个特殊的字节就无需在以后把质询写回到协处理器。

使用 **DS1963S** 用户令牌响应随机质询　图 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);
```

从用户令牌提取到 MAC (消息鉴别代码)后，协处理器就要对它进行验证。为了验证响应，在协处理器的工作区密钥中重建用户令牌的唯一密钥，而原始的账目数据则经过工作区密钥签名。

验证认证响应　图 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 验证用户证书

通过用户令牌验证证书看起来很像前面的数据初始化。这一步对数据签名而言非常相似，但协处理器产生的签名只是与现有签名进行匹配，而不是被读入证书。同时，因为整个证书都保存在用户令牌的一个文件上，所以验证证书的过程对所有令牌来说都是一样的。使用无签名的证书时，这一步只是校验用户的账目余额，以保证它在系统中是有意义的(即不是负数或几万亿美元的数字)。

使用 SHADEBIT.C 模块验证交易数据　图 42

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

为了在交易数据上签名，证书上的签名必须经过协处理器的验证。妥善的初始验证应该首先验证账目余额信息的有效性。这可以避免在不必要的时候对账目信息的签名进行验证，以实现快速校验。

验证账目余额信息　图 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!");
```

验证签名和怎样创建签名非常相似。CRC16 被清除(置为 0 电平)，数据签名存入缓冲区以后也被清除，并代之以系统的初始签名。因为新的签名不再写回这一部分，而当前的签名又要通过写次数计数器的值来进行验证，所以，在产生数据签名的这段时间里，写次数计数器并没有递增。全部过程见图 44。

注意，此处使用的是协处理器的 Match Scratchpad (匹配暂存器)命令，而不是在软件中读出签名并匹配签名字节。因为这个命令只向协处理器传送 20 个字节的 MAC 信息，而不是从协处理器读取 32 个字节的暂存器信息，所以这种方法稍微快一点。当建立一个快速、可应答的借方应用时，所有不必要的读/写操作都必须按照这种方式进行优化。另外，也要确保正确的 MAC 不要从协处理器传送出来，保证无效的用户令牌没有机会获得正确的 MAC。以上这些方法可以使应用的安全漏洞比较少，也就是用户可以利用系统“重试”的机会比较少。例如，如果系统在中断时允许用户重试质询操作，它实质上是重复 3 个字节的随机激励质询码，这时用户有可能得到存储的正确签名。

验证证书签名 图 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 更新用户证书

只有在使用动态数据的时候才有必要进行用户证书的更新。例如，如果存在证书中的只是简单的识别号，就没有必要进行更新。验证用户令牌能够保证该令牌不是另一个的副本，而验证证书则能够保证它没有被窜改。正如上述账目余额的例子，使用动态数据的时候，更新证书信息、恢复初始签名、清除 CRC16 和重签数据将是必要的。这些步骤的细化(除了更新数据以外)和 2.1 节中的叙述相同。图 45 说明的是更新(如果必要需重新签名)交易数据的函数。

记录和更新用户证书 图 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);
```