Cryptographic iButton™ Script Language

Version 1.01
9-Sep-97

Contents

CONTENTS 2

INTRODUCTION 4

CRYPTO IBUTTON SCRIPT LANGUAGE COMPONENTS 5

OBJECTS 5
COMPOSITE OBJECTS 7
SCRIPT OPERATORS 7
ASSIGNMENT STATEMENTS 10
FLOW CONTROL 11
FUNCTIONS (SOFTWARE ICS) 14

THE GROUP SOURCE FILE 16

OBJECT DECLARATION SECTION 16
SCRIPT IMPLEMENTATION SECTION 17

THE SYMBOL FILE 19

THE SYMBOL FILE PREPROCESSOR 19

TOOLS 23

THE OUTPUT LISTING FILE (.OUT) 23
THE IBUTTON MEMORY FILE (.IMF) 23
PROCEDURE 23

APPENDIX A: OBJECT DEFINITIONS 24

APPENDIX B: RESERVED WORDS 28

APPENDIX C: SCRIPT INTERPRETER ERROR CODES 30

APPENDIX D: PREPROCESSOR SPECIFICATION 33

APPENDIX E: SOFTWARE IC REFERENCE 35

APPENDIX F: SAMPLE SCRIPTS 37

Introduction

One of the major design goals of the Crypto iButton™ firmware is to give the service provider enough flexibility to support a broad range of security related applications. The two main components of the firmware that meet this goal are the command interpreter and the script interpreter. The command interpreter provides a set of firmware commands that allow the service provider to create a complete transaction group. A transaction group contains data objects (money registers, exponents, moduli, etc.) and script objects. Script objects contain a set of instructions for the manipulation of the data objects. The script interpreter carries out these instructions.

Scripts are actually created using a compiler that runs on a PC or a workstation. The output of the script compiler is a script object to be added to a transaction group. This document is mainly concerned with describing the input to the script compiler, known as the group file, and the language in which it is written. The language features described in this document exist in E-commerce Firmware, revision 1.00.

Crypto iButton Script Language Components

Objects

Crypto iButton objects are the most primitive data structure operated upon by the script interpreter. This section describes the internal representation of objects and the attributes which determine their accessibility.

All Crypto iButton objects are either user objects or automatic objects. The service provider creates user objects using the CreateCiBObject[1] command. This is typically when the object's attributes are set.

The internal representation of all objects is the same (see Figure 1). The first byte of the object structure is the object structure length. This byte is set at the time of object creation and determines the maximum length of the object data field. This implies that an object can shrink after creation but never grow past its initial length. The next byte in the object structure is the attribute byte. The attribute byte is simply the bitwise-or of any (or none) of the possible attribute bits. Currently 4 of the possible 8 attribute bits are being used. Table 1 describes the possible attributes of a Crypto iButton object.

Attribute	Definition	Value
Open	Anyone knowing the group PIN has full read/write access to an open object.	00H
Locked	Anyone knowing the group PIN has read-only access to locked objects. Only a script can alter the contents of a locked object.	01H
Private	Private objects may not be read or written from outside the Crypto iButton. Only a script has any access to private objects.	02H
Destructible	Destructible scripts become inaccessible to the script interpreter when the value of the Crypto iButton's real time clock exceeds the value of the group's destructor object.	04H
CiB Created	CiB created objects are elements of key sets generated by the Crypto iButton[2].	80H

Table 1

The next byte in the object structure specifies the object's type. Appendix A contains a complete list of Crypto iButton supported objects and their definitions. The length byte specifies the length in bytes of the current object data.

[1] The Crypto iButton Firmware Reference Manual contains a complete listing of all firmware commands.
[2] When an RSA key set is generated the Crypto iButton automatically makes one of the exponents private. A host system may use this attribute bit to determine that no one (even the service provider) ever knew the private exponent.

Object structure length (1 byte)	Attribute (1 byte)	Type (1 Byte)	Length (1 byte)	Object data (1-128 bytes)

Figure 1

The memory required for user objects is contained entirely with its transaction group. Table 2 contains a list of user objects, their allowable sizes and their type bytes.

User Object	**Size (bytes)**	**Type byte**
Modulus	1-128	20H
Exponent	1-128	21H
Money	1-128	22H
Counter	1-128	23H
Script	4-128	24H
ClockOffset	4	25H
SALT	1-128	26H
Configuration	1-128	27H
InputData	1-128	28H
Destructor	4	29H

Table 2

Automatic (auto) objects are created by the firmware when the Crypto iButton is initialized. The memory they occupy is reserved and their attributes are pre-set. Table 3 contains a list of automatic objects, their attributes, allowable sizes and corresponding type bytes.

Automatic Object	**Attributes**	**Size (bytes)**	**Type byte**
OutputData 1	Locked	1-128	A0H
OutputData 2	Locked	1-128	A1H
WorkingRegister	Private	1-128	A2H
ROMData	Locked	8	A3H
RandomFill	Private	1-128	A4H

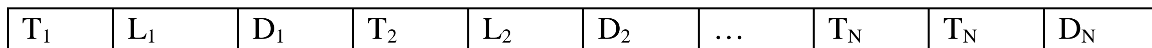
Table 3

The automatic objects are shared between groups. This leads to a different set of rules for auto objects since it is important that transaction groups be completely isolated from one

another. The Crypto iButton firmware manages these objects to make sure that one transaction group can not access (read or write) data belonging to another transaction group. Any time the Crypto iButton's command interpreter processes a group level command, it checks to see if a transition from one group to another has occurred. When a different group is accessed the firmware destroys the contents of the OutputData objects. This implies that the OutputData objects are not to be used for long-term storage. It is important that the group author keep this in mind when designing his/her group. The working register is used as a temporary working space for scripts and is unconditionally cleared every time the command interpreter is executed. Also, the integrity of the ROM data object is verified on every execution of the command interpreter.

Composite Objects

Composite (or nested) objects are simply objects that contain embedded objects within their data area. They are used to bundle several pieces of information together in a single packet that can be interpreted later by another Crypto iButton. Using composite objects is a good way to reduce the number of objects contained within a transaction group. The data area of a composite object is shown in the Figure 2 below.



T_1	L_1	D_1	T_2	L_2	D_2	…	T_N	T_N	D_N

T_N = Type byte of embedded object N
L_N = Length byte of embedded object N
D_N = Data area of embedded object N

Figure 2

The overall length of a composite object can be described as follows:

$$L_c = 2 * N + \sum_{i=1}^{N} (L_E)_i$$

Where

L_C = Length of composite object
N = Number of embedded objects contained in composite
L_E = Length of embedded object

Script Operators

The Crypto iButton script language provides several operators that allow for the manipulation of object data. There are several types of operators including arithmetic, assignment, and comparison operators. This section provides a detailed description of all operators supported by the Crypto iButton script language.

Addition (+)

The addition operator provides for the binary addition of two identically sized objects. If the object data fields have different lengths or the addition results in an overflow the script interpreter will abort and return an error code.

Subtraction (-)

The subtraction operator provides for the binary subtraction of two identically sized objects. If the object data fields have different lengths or the subtraction results in an underflow the script interpreter will abort and return an error code. This prevents a money register from ever becoming negative.

Multiplication (*)

The multiplication operator may be used to multiply 2 objects of any size. If the result of the multiplication is too large to store in the target object, the script interpreter will generate an error code.

Exclusive-Or (Xor)

This operator provides for the bitwise exclusive-or of 2 objects. Like the addition and subtraction operators the Xor operator requires identically sized objects.

Exponentiation (^)

The exponentiation operator is used for modular exponentiation only. If the modulus (see below) operator does not follow the ^ operator the script interpreter will not attempt the exponentiation and will return an error code.

```
Temp := InputPacket ^ PublicExp;     { Illegal }
Temp := InputPacket ^ PublicExp Mod RSAModulus;          { OK }
```

Modulus (Mod)

The modulus operator produces the remainder of an integer division. The modulus operator is used in modular exponentiations such as those required by RSA and Diffie-Hellman.

```
Out := g ^ x Mod n;
```

It is also used in simple modular reductions such as those required by DSA (the digital signature algorithm).

```
r := g ^ k Mod p;
```

```
r := r mod q;           { Reduce result modulo q }
```

Assignment (:=)

This operator assigns the result of an expression to the object at the left side of the assignment.

Composite assignment (<-)

The <- operator is identical to the := operator except that the type and length of the object is copied to the data field of the object at the left side of <-. Both assignment operators are described in more detail in the assignment statement section below.

Comparison (=)

The comparison operator compares the type, length and data fields of the objects on either side of =. If the comparison fails the script interpreter aborts and returns an error code. The comparison operator is described in more detail in the script statement section below.

Member of (.)

The dot operator references objects embedded within a composite object.

```
SALT = temp.Salt[1];  { Check the 1st Salt object in temp }
```

In this statement the type and length bytes used in the comparison are the type and length of the SALT object contained within temp, not the type and length bytes of temp itself.

Concatenation (&)

The concatenation operator gives a script the ability to append objects. When the script interpreter appends one object to another it always includes the type and length as well as the data field in the copy.

```
Output <- Balance & Count;
```

After the statement above has been interpreted, the Output object will contain the type, length and data of both the Balance and Count objects.

Type-less concatenation (,)

The type-less concatenation operator is identical to the & operator except that the type and length are not included in the result.

```
Output := Balance , Count;
```

After interpretation of the above statement the output object will contain only the contents of Balance followed by the contents of Count.

Assignment Statements

There are 2 different assignment operators, regular assignment := and the composite assignment <-. The only difference between the 2 operators is that the <- operator includes the objects type and length before copying its data field.

```
Output <- Balance;
```

After this statement is interpreted the data area of Output contains the type byte of Balance, (a money register) the length of the Balance data field and finally the contents of the Balance data field. Everything described below that applies to the := assignment statements, also applies to the <- assignment statement.

The simplest assignment statement copies the data field of one object to the data field of another object.

```
Output := Balance;
```

In this statement the value of the money register (Balance) is copied into the output data object (Output). This is probably not a very useful statement since Balance is probably a locked object and is therefore readable.

When the script interpreter encounters certain objects (such as counters) on the right side of the assignment operator it interprets the behavior of that object and transfers the new object data field to the assignment target.

```
Output := Count;
```

If the value of Count before this statement is interpreted is 5, the new value of count becomes 6. The new value is placed in Output. The salt, clock offset and random fill objects also have special behavior associated with them (see Appendices A and B for details). If these objects are encountered on the left side of the assignment operator, or anywhere in a comparison statement, the script interpreter treats them like any other object.

A more useful assignment statement might combine some input data (perhaps a random challenge), a counter, ROM data and a time stamp.

```
Temp := (InputPacket & Count & ROM & Time);
```

The Temp object now contains the input data combined with the auto-incremented counter, the Crypto iButton's serial number and a time stamp formed by adding the value of Time, to the Crypto iButton's real time clock. The result might then be hashed and the padded result signed with the group's secret exponent.

```
Output := (SHA1(Temp) & Pad) ^ SecretExp Mod RSAMod;
```

Assignment statements must obey a few rules, which are intended to keep the implementation of the script interpreter simple. Embedded objects must not appear on the left side of either assignment operator. The following statements are both illegal.

```
Temp.Money[1] := Balance;
Temp.Salt[1] <- MySALT;
```

Also assignment statements are always evaluated from left to right. Parentheses may however be used when writing scripts to enhance readability.

Flow Control

The Crypto iButton executes scripts in sequence unless told to do otherwise by a conditional or unconditional jump statement. There are two conditional statements supported in the Crypto iButton script language, the comparison statement and the **If-Then-Else** statement, and three unconditional statements, the **Goto**, **Continue,** and **Exit** statements.

Comparison statements

If the comparison of two objects fails, the script is aborted immediately and an error code[3] is returned. The following code fragment shows a common use of the comparison statement.

```
Temp := InputPacket ^PublicExp Mod RSAMod; { Decrypt signed packet }
MySALT = Temp.Salt[1];                     { Challenge properly met? }
MyMoney := MyMoney+Temp.Money[1];          { OK to increase balance. }
```

In this example a random challenge was generated by the Crypto iButton and contained in the object MySALT. Once the signed challenge was received from the recipient it was written into the InputPacket object. The script listed above decrypts the signed packet with the public key (it is assumed that the public key being used has been previously verified). The comparison statement is used to make sure that the SALT object in the resulting packet

[3] Appendix C contains a complete list of script interpreter error codes.

is the same as MySALT. If they are identical the group's money register balance is increased and the script interpreter returns a successful status code. If the SALT object in InputPacket is not identical to MySALT the script interpreter generates an error code and does not interpret any statements that follow. The comparison statement does not perform an actual jump. It simply continues if the comparison succeeds and aborts with error if the comparison fails.

If-Then-Else

The **If-Then-Else** statement, unlike the comparison statement, will perform a jump to an offset that depends on the result of a comparison. The following code segment shows a typical **If-Then-Else** statement.

```
Temp := InputPacket ^PublicExp Mod RSAMod; { Decrypt signed packet }

If MySALT = Temp.Salt[1] Then Begin          { Challenge properly met? }
        { OK to increase monetary balance }
        MyMoney := MyMoney+Temp.Money[1];
End Else Begin
        { Increment error counter }
        ErrorCount := ErrorCount;
End

{ Increment transaction counter }
TransactionCount := TransactionCount;
```

This example is similar to the previous one. The difference is that, after the comparison is made, more statements are executed, regardless of the outcome. If the random challenge was properly signed, the group's money register balance is increased. If the comparison fails, an error counter is incremented to keep track of the total number of failures. Regardless of the outcome, a transaction counter is incremented to keep track of the total number of times this script was executed. The **Else** can be omitted if there are no statements necessary to handle the failed comparison condition.

Goto

The **Goto** statement performs an unconditional jump to a label contained within the script that is currently being executed. This code segment shows two forward jumps to labels located within the same script. Backward jumps are also allowed.

```
Temp := InputPacket ^PublicExp Mod RSAMod; { Decrypt signed packet }
TransactionCount := TransactionCount;       {Increment Transaction Count}
If MySALT = Temp.Salt[1] Then               { Challenge properly met? }
        Goto DoTransaction;                 {Perform transaction}
```

```
Else
        ErrorCount := ErrorCount;               {Increment Error Counter}

Goto AfterTransaction;                          {Don't perform the transation}

DoTransaction:
MyMoney := MyMoney+Temp.Money[1];       { OK to increase balance. }

AfterTransaction:
```

This example performs the same operations as the previous one using **Goto** statements instead of using the more natural flow of the **If-Then-Else** statement.

Continue

The **Continue** statement performs a jump to another script within the same group. If a script exceeds the maximum object size (currently 128 bytes), it can be broken into multiple scripts and an unconditional jump can be made from one script to another. Script interpreter execution will continue with the first statement of the target script. These code fragments demonstrate a typical continue statement.

```
Script MajorTransaction1;
Begin
        …
        {Multiple statements using up almost the entire script object}
        …
Temp := InputPacket ^PublicExp Mod RSAMod; { Decrypt signed packet }
TransactionCount := TransactionCount;       {Increment Transaction Count}
Continue(MajorTransaction2);                {Continue with next script}
End
```

```
Script MajorTransaction2;
Begin
If MySALT = Temp.Salt[1] Then Begin         { Challenge properly met? }
        { OK to increase monetary balance }
        MyMoney := MyMoney+Temp.Money[1];
End Else Begin
        { Increment error counter }
        ErrorCount := ErrorCount;
End

{ Increment transaction counter }
TransactionCount := TransactionCount;
End
```

Note: The **Continue** statement performs a jump, not a call. There is no return to the original script.

Exit

The **Exit** statement aborts the execution of a script and returns a user-defined error code. This error code returns useful information to an application indicating neither success nor failure but a user-defined code indicating success or one of many error codes describing the reason for the failure. This code sample demonstrates one possible use of the **Exit** statement.

```
Temp := InputPacket ^PublicExp Mod RSAMod; {Decrypt signed packet }
TransactionCount := TransactionCount;      {Increment Transaction Count}
If TransactionCount = Temp.Counter[1] Then {Reached max transaction?}
        Goto MaxTransactionReached;
If MySALT = Temp.Salt[1] Then              {Challenge properly met? }
        Goto DoTransaction;                {Perform transaction}
Else
        ErrorCount := ErrorCount;          {Increment Error Counter}
Exit(1);                                   {Return bad challenge code}

DoTransaction:
MyMoney := MyMoney+Temp.Money[1];          {OK to increase balance. }
Exit(0);                                   {Exit with success code}
MaxTransactionReached:
Exit(2);                                   {Return max transaction code}
```

Here a value of zero is returned if the script executed flawlessly, a value of one is returned if the challenge was not met, and a value of two is returned if the maximum number of transactions allowed was reached. The return code is a single byte value.

Functions (Software ICs)

The Crypto iButton script language provides very little in the way of operators and even less for flow control. Because of these limitations is it is not convenient for implementing complex cryptographic algorithms such as hashing functions and block ciphers. In order to provide support for these and other complex operations the script interpreter[4] allows calls into a function library.

Function calls may appear only in assignment statements. The following statement calls the SHA1 hash function and signs the result with the group's secret exponent.

```
Output := SHA1(InputPacket) ^ SecretExp Mod RSAMod;
```

[4] Only Crypto iButton Firmware versions 0.50 and above support function calling.

The parameter list is comma delimited and must be contained within parentheses. What is actually passed to the SHA1 function is the object ID of InputPacket, not the object data. This is similar to passing parameters by reference in other languages.

Currently (as of firmware revision 1.00) the only cryptographic functions[5] supported are the SHA1 and MD5 hashing functions. Other functions such as DES, triple DES and DSA are being considered for addition to the function library.

[5] Appendix E contains a complete description of all functions supported as of firmware revision 1.00

The Group Source File

The script compiler requires two input files to generate script objects, the group source file (with an extension of .ibg) and a symbol file (with an extension of .sym). The group source file is comprised of a declaration section and one or more implementation sections. The declaration section defines all group objects used by the scripts and specifies their attributes. The implementation section contains the script source code. These sections will now be described in detail.

Object Declaration Section

The object declaration section begins with the heading **TransactionGroup**, followed by the group name. A transaction group designed for exchanging a triple DES key might have the following header statement:

```
TransactionGroup('3DES Exchange');
```

The name of the transaction group must be contained within single quotes. The maximum allowable group name length is 16 bytes and may consist of any ASCII characters other than the single quote or parentheses. Note that the group source file is not case sensitive.

The object declarations follow immediately after the heading and are bracketed by the **begin** and **end** reserved words[6]. This section associates object names with their types and assigns their attributes. A sample declaration section for a group that uses RSA for exchanging a symmetric key follows.

```
TransactionGroup('3DES Exchange');
{
        Transaction group used to exchange a triple DES key.
}
Begin
        Open:           { Read/Write objects }
                DESKey: InputData;

        Locked:         { Read only objects }
                RSAMod: Modulus;
                PublicExp: Exponent;
                EncryptDESKey: Script;
                DecryptDESKey: Script;
                Result: OutputData;
```

[6] Appendix B contains a complete list of reserved words.

```
        Private:        { No read/write access to these objects }
                SecretExp: Exponent;
                Pad: RandomFill;
End
```

The declaration section is divided into **open**, **locked** and **private** subdivisions. The subdivision in which an object is declared determines its attributes. If objects are declared outside of any of these subdivisions the script compiler assumes they are **open** objects.

Note that any object can be made destructible by appending the word **Destructible** to the end of its declaration.

```
EncryptDESKey: Script; Destructible;
```

Since this particular sample group does not contain a destructor, the destructible attribute would have no affect on the group.

Script Implementation Section

A script implementation section consists of 2 parts. The first is the script declaration that begins with the reserved word **script** followed by the name assigned to the script in the object declaration section.

```
Script EncryptDESKey;
```

If the script itself is destructible, the word **destructible** should be appended to the end of the script declaration.

```
Script EncryptDESKey; Destructible;
```

The body of a script is contained within a **begin** and **end** block that follows the declaration. The entire EncryptDESKey script follows.

```
Script EncryptDESKey;
{
        Encrypt triple DES key in input data object using the public
exponent.
}
Begin
        Result := (DESKey & Pad) ^ PublicExp Mod RSAMod;
End
```

This sample script contains only a single statement. However multiple statements may be contained within the **begin/end** block[7].

[7] Appendix F contains several larger sample scripts.

The Symbol File

Symbol information is maintained in a file with a sym extension. The symbol file assigns object ids to objects declared in the group source file. The following symbol file would be used with the key exchange group file described above and is the simplest form of the symbol file. It gives no information about the objects, aside from the ids, and would produce objects with default sizes and initialized with all zeros.

```
RSAMod          =$01
PublicExp       =$02
SecretExp       =$03
DESKey                  =$04
EncryptDESKey           =$05
DecryptDESKey=$06
Result          =$A0
Pad             =$A4
```

The symbol file preprocessor

The script compiler preprocesses the symbol file before processing the group source file. Lists of preprocessor arguments can be added for each object in the symbol file to further describe its attributes such as size and initial data. The list for an object begins with the sequence **{+** and ends with the sequence **-}.** The preprocessor is not case sensitive.

Size

The size argument begins with the letter **S** and is followed by a number indicating the size of the object. This number can be decimal or hex (128 or $80). The argument **S**128 is equivalent to **S**$80. The following preprocessor line would tell the compiler to create the **InputData** object EncryptedDESKey with size 128 bytes and, by default, will initialize it with all zeros.

```
EncryptedDESKey = $04       {+ S128 -}
```

Initial Data

The initial data argument begins with the letter **I** and is followed with the actual data. There are three types of initial data. Random data, byte sequences and strings. A repeat value can follow this argument to indicate that the data provided is to be repeated multiple times. The repeat value is optional. The format for initial data is **I(R**L) or **I(**B)n or **I'**S**'**n where L is the

number of random bytes, B is a byte sequence delimited with spaces or commas, S is a text string and n is the number of times the data is to be repeated.

Random data

Random data is indicated with the letter **R** and is followed with a value indicating the number of random bytes to be generated. This random number is generated using a standard C library function, seeded with the time. The following preprocessor line will create a **Salt** object MySalt with size 64 bytes and initialize it with 64 bytes of random data.

```
MySalt = $07 {+ S$40 I(R$40) -}
```

Byte Sequence

Byte Sequence data is described by a list of space or comma delimited bytes enclosed between parenthesis. The following line will create an **Exponent** object PublicExponent with size 3 bytes and initialize it with the value 65537.

```
PublicExponent = $01 {+ S3 I($01,$00,$01) -}
```

String Data

String data is described by a text sequence enclosed between single quotes. The text between the quotes is copied with case preserved. The following line will create a **Configuration** object VersionString with size 64 and will initialize it with the string 'Debit transaction group, version 1.00'.

```
VersionString = 10 {+ S64 I' Debit transaction group, version 1.00' -}
```

Composite Data

The composite data argument begins with the letter **C** and is followed by composite data enclosed between parentheses. The format for composite data is

C(type1**,**len1[**,**initialdata1][**;** type2**,**len2[**,**initialdata2]]…**)**.

The type can be any of the types that can be allowed in the declaration section of the group source file (**Salt**, **Money**, etc.). The length is the number of bytes that the type uses within the configuration data. The initial data is optional and is described in the Initial Data section above. The following line will create a **Configuration** object Config1 with size 40 and with composite data containing a **ROMData** object, with size 8 initialized with zeros, and a **Salt** object, with size 20 initialized with random data.

```
Config1 = $04 {+ s40 C(ROMData,8,I(0)8;Salt,20,I(R20)) -}
```

Button Created

The Crypto iButton can generate its own objects, such as an entire RSA key set consisting of a modulus, a public exponent and a private exponent or it can generate the modulus and private exponent when given the public exponent. Note: The object ids for Button created objects must be ordered correctly and contiguously or the objects will not be created properly. When the Crypto iButton generates an entire key set, it creates the modulus first followed by the public exponent and then the private exponent. When the public exponent is supplied, the Crypto iButton creates the modulus in the next object and the private exponent in the following object. This ordering must be observed in the symbol file.

The following two lines show examples of key sets generated by the Crypto iButton.

```
{complete button created key set}
CiBModulus0   = 1 {+ S128 B -}
CiBPublicExp0 = 2 {+ S128 B -}
CiBPrivateExp0= 3 {+ S128 B -}
```

```
{button created modulus and private key}
CiBPublicExp1 = 4 {+ S3 I($01,$00,$01) -}
CiBModulus1   = 5 {+ S128 B -}
CiBPrivateExp1= 6 {+ S128 B -}
```

Group Password

An initial password for the group can be specified at the top of the symbol file. Please note that anyone that has access to the symbol file will know the initial group password. The password argument begins with the letter **P** and is followed by a byte sequence or a string, described in the Initial Data section, of size 8 or less bytes. The following lines will set the group password to 8 ‘U’s using first a byte sequence and then a string.

```
P($55, $55, $55, $55, $55, $55, $55, $55)
```

```
P’UUUUUUUU’
```

Replace With

The Replace With statement provides a mechanism to replace a numeric constant with a more readable name.

```
Replace True With 1
Replace False With 0
```

Tools

There are two software development tools currently available for programming Crypto iButtons. The first tool, **scompile**, takes two input files, a group source file, <filename>, and a symbol file, <filename>.sym. If these files compile successfully, two output files are generated, an output listing file, <filename>.out, and an iButton memory file, <filename>.imf. If the compile does not succeed, only the output listing file, containing errors, is generated. The second tool, **imftocib** uses the contents of the iButton memory file (iMF) to create the group (described in the input files) in a Crypto iButton.

The Output Listing File (.out)

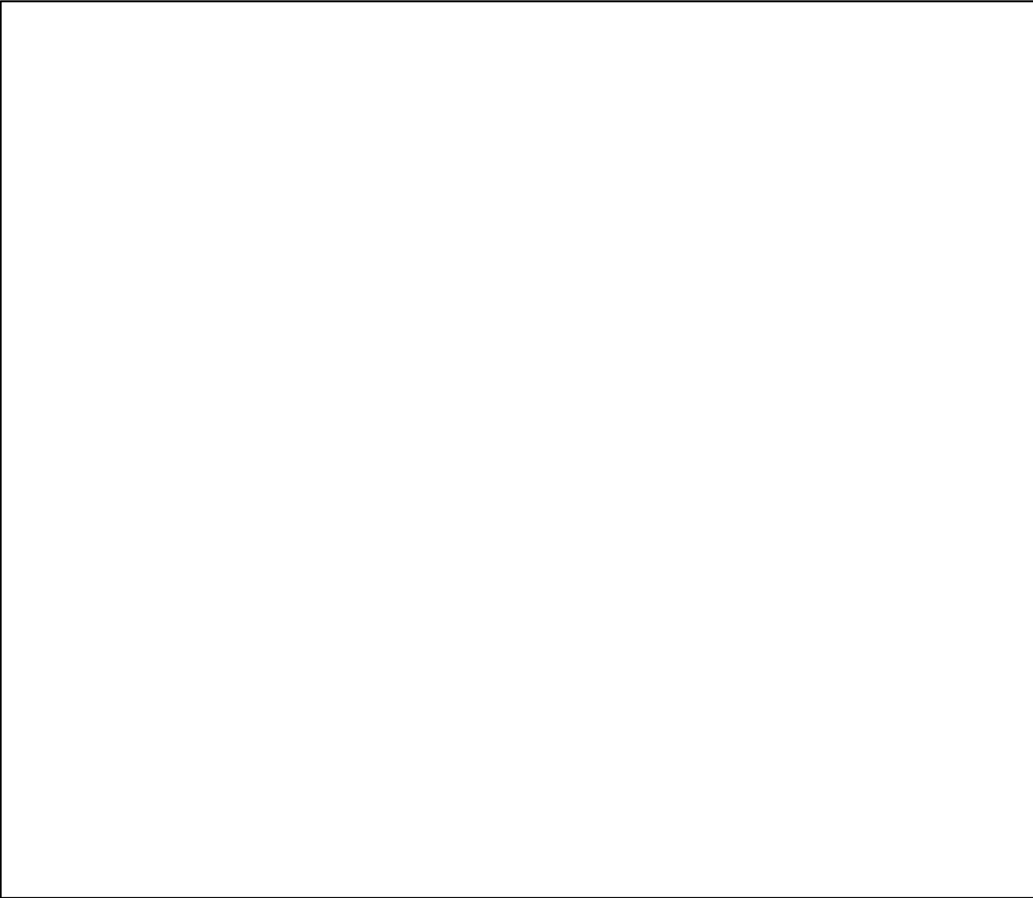
The output listing file, generated by **scompile**, contains information that is dependent on the success or failure of the compiler. If the compile succeeds, this file contains information about the transaction group. This information includes the name of the group, a list of objects contained within the group and a hex dump of the scripts generated. If the compile fails, the file contains error information including the reason for failure and the line number where the failure occurred.

The iButton Memory File (.imf)

The iMF contains an image of one (or more) transaction groups as it might appear in a Crypto iButton's memory space. This intermediate file allows a transaction group to be programmed into one or more Crypto iButtons, using **imftocib**, in a relatively short period of time (button created objects may stretch this period significantly depending on object size).

Procedure

1. Verify that the Blue Dot Receptor is connected to the hardware port and any necessary software libraries are installed.
2. Write a group source file.
3. Write a symbol file to accompany the source file.
4. Run **scompile** using the group source file name as the only command-line parameter.
5. If the compile failed, view the output listing file to find the source of the errors, modify the source file or symbol file and return to step 3. Otherwise go to step 6.
6. Connect a Crypto iButton to the Blue Dot Receptor and run **imftocib** using the iButton memory file name as the only command-line parameter.
7. Test the transaction group.



Appendix A: Object Definitions

OutputData

The OutputData object is used by transaction scripts as an output buffer. Two of these objects are automatically created during Crypto iButton initialization. They are shared by all transaction groups and are cleared automatically whenever a new transaction group is accessed. Each output data object can be as large as 128 bytes in length and inherits PIN protection from its group. There may not be any additional Output Data objects in a transaction group.

WorkingRegister

This object is used by the script interpreter as working space and may be used in a transaction script. This object is automatically created when the transaction group is created. It is a private object and cannot be read using the ReadObject command. There may only be one Working Register object in a transaction group.

ROMData

This object is automatically created when the transaction group is created. It is a locked object and cannot be altered using the write object command. This object is eight bytes in length and its contents are identical to the eight byte ROM Data of the Crypto iButton. There may only be one ROM Data object in a transaction group.

RandomFill

When the script interpreter encounters this type of object it automatically pads the current message so that its length is 1 bit smaller than the length of the following modulus. This object is automatically created when the transaction group is created. It is a private object and may not be read using the read object command. There may only be one Random Fill object in a transaction group.

Modulus

A modulus object is a large integer of at most 128 bytes in length. It must be used by scripts, which perform modular exponentiations.

Exponent

An exponent object is (typically) a large integer of at most 128 bytes in length. It is used as the exponent value in modular exponentiations.

Money

The money object may be used to represent money or some other form of credit. Once this object has been created it must be locked to prevent a user from tampering with its value. Once locked only invoking a transaction script can alter the value of this object. A typical

transaction group, which performs monetary transactions, might have one script for withdrawals from the money register and one for deposits to the money register.

Counter

The counter object is usually initialized to zero when it is created. Every time a transaction script, which references this object, is invoked, the transaction counter increments by 1. Once a transaction counter has been locked it is read only and provides an irreversible counter.

Script

A script is a series of instructions to be carried out by the Crypto iButton's script interpreter. When invoked the Crypto iButton firmware interprets the instructions in the script and typically places the results in the output data object (see above). The actual script is simply a list of objects and valid script operators. Scripts may be as long as 128 bytes and may be continued as long as needed.

ClockOffset

This object is a 4-byte number, which contains the difference between the reading of the Crypto iButton's real-time clock and some convenient time (e.g. 12:00AM, January 1, 1970). The true time can then be obtained from the Crypto iButton by adding the value of the clock offset to the real-time clock.

SALT

A SALT (random challenge) object is simply a large random number. When the script interpreter encounters a SALT object on the right-hand side of an assignment operator a new random number replaces its value.

Configuration

This is a user-defined structure with a maximum length of 128 bytes. This object is typically used to store configuration information specific to its transaction group. For example, the configuration data object may be used to specify the format of the money register object (i.e. the type of currency it represents). This object has no pre-defined structure.

InputData

An input data object is simply an input buffer with a maximum length of 128 bytes. The host uses input data objects to store data to be processed by transaction scripts.

Destructor

A destructor object is 4 bytes in length and is initialized to some value greater than the Crypto iButton's real time clock. When the script interpreter is called it checks the group to see if it contains a destructor. If it does, it checks the script itself, and all objects referenced in the script to see if they are destructible. If any of the objects are destructible, the script

interpreter compares the value of the destructor with the value of the real time clock. If the value in the clock is greater than the destructor's value, the script interpreter terminates the script with the ERR_DESTRUCTED_OBJECT error code. There may only be one destructor object in a transaction group.

Appendix B: Reserved Words

Begin

Used to specify the start of the script implementation or the object declaration section

End

Used to specify the end of the script implementation or the object declaration section

Open

Used to specify the start of the list of **open** objects.

Locked

Used to specify the start of the list of **locked** objects.

Private

Used to specify the start of the list of **private** objects.

TransactionGroup

Usually the 1st line of a group source file. The **TransactionGroup** line specifies the group name.

Script

As well as being an object type, the reserved word **script** is used in the implementation section to specify the beginning of the script code.

Note that all object types are reserved words. A list of object types and their definitions appears in Appendix A.

If, **Then** and **Else**

Used to test a condition and jump to an offset determined by the result of the test.

Continue

Used to jump from one script to another script contained within the same group.

Goto

Used to jump to a label contained within the same script.

Exit

Aborts execution of a script and returns a user-defined error code.

Replace, With

The Replace With statement is used to associate a numerical constant with a name.

Appendix C: Script Interpreter Error Codes

Table 4 (Error codes)

Error Code	Value	Description
ERR_OPERATOR_NOT_EXPECTED	C0H	Returned when an operator was misused.
ERR_EOS_EXPECTED	C1H	An end of statement (;) was expected but was not found.
ERR_BAD_ID	C2H	An object that did not exist was referenced within the script.
ERR_NOT_COMPOSITE	C3H	Returned when the member of (.) operator references an embedded object which may not contain embedded objects.
ERR_UNEXPECTED_END	C4H	Returned when a statement ends unexpectedly.
ERR_NOT_AN_OPERATOR	C5H	Returned when an operator was expected, but no operator was found.
ERR_BAD_TYPE	C6H	The first byte after the member of (.) operator must be an object type specification byte. If it is not a valid type byte, the script interpreter will return this error code.
ERR_MEMBER_NOT_FOUND	C7H	The script interpreter could not find the embedded object referenced by the member of (.) operator.
ERR_BAD_COMPARE	C8H	The left and right values of a comparison statement were not identical.
ERR_BAD_ADDITION	C9H	An overflow occurred while adding 2 objects.
ERR_BAD_SUBTRACTION	CAH	An underflow occurred while subtracting 2 objects.

Error Code	**Value**	**Description**
ERR_SIZE	CBH	A size mismatch error has occurred. This error code is usually returned when the interpreter is instructed to move an object into another object with a smaller data field.
ERR_NOT_EXPONENT	CDH	The object id of an exponent did not follow the exponentiation (^) operator
ERR_NOT_MODULUS	CEH	The object ID of a modulus did not follow the modulus (Mod) operator.
ERR_MOD_OP_EXPECTED	CFH	The exponentiation (^) operator and the object ID of an exponent were found, but not followed by the modulus (Mod) operator and the ID of a modulus.
ERR_OBJECT_DESTRUCTED	D0H	Before the script interpreter begins executing the instructions within the script, it checks all of the objects referenced by the script to make sure they have not become inactive. For an object to be inactive it must be a destructible object and the group must contain a destructor whose value is less than that of the real time clock.
ERR_CIB_INACTIVE	D1H	The Crypto iButton's activation period has expired. This implies the real time clock's value has exceeded that of the group 1 destructor.
ERR_FUNCTION_CALL	D2H	An error occurred within the function library. This is typically caused by invalid parameters being passed to a function.
ERR_PARAMETER_COUNT	D3H	An incorrect number of parameters were supplied to a function.
ERR_GOTO	D4H	The offset following a goto was invalid.

ERR_NOT_SCRIPT	D5H	The object ID supplied in a continue statement was not the ID of a script object.
ERR_BAD_ELSE	D6H	The offset specified in an else statement was out of range.

Appendix D: Preprocessor Specification

Preprocessor arguments will be placed in the symbol file that accompanies every group source file. All preprocessor argument sequences will begin with the string "**{+**" and will end with the string "**-}**". Multiple arguments will be separated by a space. Arguments are processed from left to right.

All **bold** characters are the actual characters used in an argument, all *italicized* characters/strings indicate variables and all parameters contained [within square brackets] are optional.

Pre-processor group arguments

P(b_1 b_2 … b_{m-1} b_m) - initialize with the byte sequence $b_1 \ldots b_m$,
bytes are separated by commas or white space,

P'*password*' - Set group password to *password*.

Pre-processor object arguments

B - Button created. {default: no}

S*n* - Object size = *n*, $0 < n < 129$. {default: *n*=1}

I(R*l_1*) or **I(**$\vec{B}_1$)[n_1] or **I**$\vec{S}_1$[n_1]- Initial data.

{**R***l* - initialize with random data, [l random bytes]

$\vec{B}$ (=$b_1 \ldots b_m$) - initialize with the byte sequence $b_1 \ldots b_m$, [repeated *n* times]
bytes are separated by commas or white space,

$\vec{S}$ (=**'***string***'**) - initialize with the string *string*, [*n* times]

- default: all bytes initialized with zero}

C(*$CompositeData_1$* or *$InitialData_1$*[[**;***$CompositeData_2$* or *$InitialData_2$*]…[*$CompositeData_m$* or *$InitialData_m$*]]**)**

{*$CompositeData_i$* = *$type_i$***,** *len_i* [**,** *$InitialData_i$*]}

Composite entries can be either *CompositeData*, which specifies the type *type*, length *len* and optional initialization data *InitialData* or simply *InitialData, which* does not contain the type or length information.

Composite Object Example

```
ConfigInfo      = $04 {+ S45 C(ROMData, 8, I(0)8;
                           InputData, 17, I'Special Project 1';
                           SALT, $14, I(R20)) -}
```

{Configuration object (with ObjectId 4), 45 bytes in length with 8 bytes of RomData, initialized to 0, 17 bytes of input data initialized to 'Special Project 1' and a 20 byte Salt initialized with random data}

Symbol File Example

```
{+ P'password' -}
{User objects}
PrivateExponent     =$01   {+ S128 B -}
PublicExponent      =$02   {+ S128 B -}
MyModulus           =$03   {+ S128 B-}
ConfigInfo          =$04   {+ S45 C(ROMData,8,I(0)8;
                                InputData,17,I'Special Project 1';
                                SALT,$14,I(R$14)) -}
Input1              =$05   {+ S128 I(0)128 -}
EncryptScript       =$06   {Script information generated by compiler}

{Auto Objects}
Output              =$A0

Functions:
SHA1                =$01
```

Appendix E: Software IC reference

SHA1 Function code (01)

This function performs the Secure Hash Algorithm as described in FIPS 180-1. SHA-1 requires the object ID of the object to be hashed.

```
Hash := SHA1(Message);
```

The parameter passed to SHA1 must be an object ID and not a reference to a composite object. The following statement is not supported.

```
Hash := SHA1(Input.Configuration[1]);
```

ToBCD Function code (02)

This function converts an object stored in binary using little-endian byte ordering to BCD using big-endian byte ordering. ToBCD requires the ID of the binary object.

```
BCDOut := ToBCD(BinBalance);
```

The parameter passed to ToBCD must be an object ID and not a reference to a composite object.

ToBin Function code (03)

This function converts an object stored in BCD using big-endian byte ordering to binary using little-endian byte ordering. ToBin requires the ID of a BCD encoded object.

```
BinOut := ToBin(BCDBalance);
```

The parameter passed to ToBin must be an object ID and not a reference to a composite object.

MD5 Function code (04)

This function performs the MD5 message digest algorithm. MD5 requires 4 parameters described below.

```
Hash := MD5(MessageData, HashVal, BlockCount, LastMessage);
```

MessageData is the object ID of the data to be hashed. If this is not the last block(s) of the entire message, the length of MessageData must be either 64 or 128 bytes. When MD5 is called with the final block of the message the length of MessageData can be any valid object length.

HashVal is the object ID of the object containing the initial hash value. When MD5 is called with the 1st block(s) of the message the ID passed for HashVal must be 0.

BlockCount is the object ID of the block counter. This is typically a locked configuration object. When MD5 is called with the 1st block(s) of message data the contents of the BlockCount object are automatically initialized.

LastMessage is either a 0 or a 1. If the current call to MD5 does not contain the last block(s) of the message, LastMessage must be 0. Otherwise LastMessage must equal 1.

The following code sample hashes the contents of 2 objects and places the resutt in the output object named Hash.

```
Temp := MD5(Input1, 0, BlockCount, 0);
Hash := MD5(Input2, Temp, BlockCount, 1);
```

There is no practical limit on the total length of the message hashed using the MD5 function.

Appendix F: Sample Scripts

Example 1: Digital Notary

The digital notary example listed below takes a small message as input data and hashes it using the SHA1 function. A script execution count, time stamp and the Crypto iButton serial number are all appended to the input and the result is hashed. The hash is then padded with random data and the result is signed using the group's secret exponent.

```
TransactionGroup('Digital Notary');
{
        Transaction group that digitally signs information provided in the
input data object.
}
Begin
        Open:
                Msg: InputData;
        Locked:
                SignIt: Script;
                Certificate,
                Plain:    OutputData;
                RSAMod: Modulus;
                ExecCnt: Counter;
                TimeStamp: ClockOffset;
                SerId: ROMData;
        Private:
                SecretExp: Exponent;
                Temp: WorkingRegister;
                Pad : RandomFill;
End

Script DigitalNotary;

Begin
        Temp <- Msg & ExecCnt & TimeStamp & SerId;
        Plain := Temp;
        Temp <- SHA1(Temp);
        Certificate := (Temp & Pad)^ SecretExp Mod RSAMod;
End
```

The first assignment statement begins by copying Msg into Temp as an embedded object. ExecCnt is then incremented by 1 and the result is appended to Msg. TimeStamp is a clock-offset object. When a clock-offset object is referenced on the right side of an assignment operator, the interpreter adds the clock offset value to current value of the real time clock. The result is then appended to temp. Note that when ExecCnt is referenced its value is changed. However the value of TimeStamp is not altered. The first assignment statement

concludes by appending the Crypto iButton's serial number to the other 3 objects embedded in Temp. Next the contents of temp are copied to a readable output data object to be used in the signature verification process.

The next statement hashes the intermediate value and reuses the temp object to hold the result of the hash function. Finally the resulting hash value is padded with random data and signed using the group's secret RSA exponent.

Example 2: Electronic Purse

This sample group maintains a monetary balance in a money register using two scripts. One script is used for deposits and the other for withdrawals. The deposit script is made destructible to prevent refill of the money register after a pre-specified period of time.

```
TransactionGroup('Checkbook');
{
        Maintain balance in a money register using separate scripts for
deposits and withdrawals.
}

Begin
        Open:
                Request:        InputData;

        Locked:
                Deposit:        Script;
                Withdrawal:     Script;
                Challenge:      SALT;
                Time:           ClockOffset;
                RSAMod,
                VendorMod:      Modulus;
                VendorPub:      Exponent;
                SerId:          ROMData;
                LifeSpan:       Destructor;
                Plain,
                Signed:                 OutputData;

        Private:
                SecretExp:      Exponent;
                Temp:           WorkingRegister;
                Pad:            RandomFill;
End

Script Deposit; Destructible;
{
        Make deposit to money register if input certificate was signed by
the vendor.
}
Begin
        { Decrypt packet using vendor's public key }
```

```
        Temp := Request ^ VendorPub Mod VendorMod;
        { Make sure the challenge was successfully met }
        Temp.Salt[1] = Challenge;
        { Generate a new random challenge }
        Challenge := Challenge;
        Balance := Balance + Temp.Money[1];
End

Script Withdrawal;
{
        Make withdrawal from money register and sign the input packet
with the group's private key.
}
Begin
        Balance := Balance – Request.Money[1];
        Plain := Request & SerId & Time;
        Signed := (SHA1(Plain) & Pad) ^ SecretExp Mod RSAMod;
End
```

The deposit script decrypts the request certificate using the vendor's public key and checks the SALT object embedded in the resulting plaintext. If it matches the value of the Challenge object, a new value of Challenge is generated and the money register is increased by the amount indicated in the embedded money object. Note that line 4 (Challenge := Challenge;) of this script is required to avoid packet replay. Each time a random challenge is successfully signed a new random challenge must be generated.