# Guide to the Open Standard Feature Set of the Crypto iButton™



## I. Introduction

This document describes the open standard features that have been preprogrammed into the Dallas Semiconductor Primary Group (Dallas Primary) of the DS1954 Crypto iButton™ to facilitate its use for a variety of cryptographic applications. The features supported by the primary group depend on whether the part is prepared for export or intended for U.S./Canadian use only. At the present time the feature set of Dallas Primary is limited in the export model to accommodate current U.S. export restrictions on strong cryptography. The features supported by the export model include digital notary, digital signature, and strong authentication using the challenge-response protocol. (All of the features supported in the export model employ the strong cryptographic hash SHA-1 in such a way as to prevent the part from being used to enable the transmission of secret messages.) The U.S./Canada version of Dallas Primary provides all of the features of the export version and in addition supports such applications as secure session key exchange, public key encryption and decryption of e-mail, and encryption and decryption of personal data files stored on a computer disk. All encryption using RSA public key cryptography performed by the Crypto iButton™ is covered by a paid-up license from RSA Data Security, Inc.

U.S. and Canadian service providers with requirements for more complex functionality than that provided by Dallas Primary can easily design and program a proprietary Transaction Group of their own with the custom features that they require. The Dallas Primary is designed to support the core functionality common to many different applications, and to make it possible for customers to use the Crypto iButton™ for basic cryptographic applications without having to learn the script language for Transaction Group programming. The advanced features programmable with the script language include the secure collection and distribution of money, programmable timed expiration of selected features on a particular date, mutual remote authentication between Crypto iButtons™ to allow exchange of monetary value, software usage metering, and many others.

## II. Overview of the Firmware Design

The Crypto iButton™ contains a general-purpose, 8051-compatible microcontroller, a tamper-evident real-time clock, a high-speed modular exponentiation accelerator for large integers up to 1024 bits in length, input and output buffers with the standard iButton™ 1-Wire “front-end” for sending and receiving data, 32 KBytes of ROM memory with preprogrammed firmware, 6 KBytes of NVRAM for storage of critical data, and control circuitry that enables the microcontroller to be powered up to interpret and act on the data placed in an input data buffer, drawing its operating power from the 1-Wire line. The microcontroller, clock, memory, buffers, 1-Wire front-end, modular exponentiation accelerator, and control circuitry are integrated on a single silicon chip and packaged in a stainless steel microcan using packaging techniques which make it virtually impossible to probe the data in the NVRAM without destroying the data. Initially, most of the NVRAM is available for use to support cryptographic applications such as those mentioned above.

### A. Programmable Transaction Groups

The Crypto iButton™ is intended to be used by both a Service Provider who co-issues the device to his own

customers, and the registered person who is the end user of the device. The Service Provider loads the Crypto iButton™ with data to enable it to perform useful functions. The registered person issues commands to the Crypto iButton™ to perform operations programmed by the Service Provider for his benefit. For this reason the Crypto iButton™ offers functions to support the Service Provider in setting up the Crypto iButton™ for an intended application, and it also offers functions to allow the authorized person to invoke the services offered by the Service Provider.

Each Service Provider can reserve a block of NVRAM memory to support its services by creating a Transaction Group. A Transaction Group is simply a set of Objects that are defined by the Service Provider. These Objects include both data Objects (encryption keys, transaction counts, money amounts, date/time stamps, etc.) and Transaction Scripts which specify how to combine the data Objects in useful ways. Each Service Provider creates his own Transaction Group, which is independent of every other Transaction Group. Hence, multiple Service Providers can offer different services in the same Crypto iButton™. The number of independent Service Providers that can be supported depends on the number and length of the Objects defined in each Transaction Group. Examples of some of the Objects that can be defined within a Transaction Group are the following:

| | | |
|---|---|---|
| Modulus | Money Register | Input Data |
| Exponent | Clock Offset | Output Data |
| Transaction Script | Random Salt | Destructor |
| Transaction Counter | Configuration Data | |

Within each Transaction Group, the Crypto iButton™ will initially accept certain commands which have an irreversible effect. Once any of these irreversible commands is executed in a Transaction Group, it remains in effect for the life of the Crypto iButton™ or until the Transaction Group to which it applies is erased from the Crypto iButton™. In addition, there are certain commands which have an irreversible effect on the Crypto iButton™ as a whole, and remain in effect for the life of the Crypto iButton™ or until a master erase command is issued to erase the entire contents of the Crypto iButton™. These commands are essential to give the Service Provider the necessary control over the operations that can be performed by the user. Examples of some of the irreversible commands are:

| | |
|---|---|
| Privatize Object | Lock Object |
| Lock Transaction Group | Lock Crypto iButton™ |

Since much of the Crypto iButton™'s utility centers on its ability to keep a secret, the Privatize command is the most important irreversible command.

The fundamental concept implemented by the firmware is that the Service Provider can store Transaction Scripts in a Transaction Group to perform only those operations among Objects that he wishes the authorized person to be able to perform. The Service Provider can also store and privatize the private key or keys that allow the Crypto iButton™ to "sign" transactions on behalf of the Service Provider,

thereby guaranteeing their authenticity. By privatizing and/or locking one or more Objects in the Transaction Group and locking the Transaction Group itself, the Service Provider maintains control over what the Crypto iButton™ is allowed to do on his behalf. The user cannot add new Transaction Scripts and is therefore limited to the operations on Objects that can be performed with the Transaction Scripts programmed by the Service Provider.

B. The Dallas Primary Feature Set

To facilitate use of the Crypto iButton™ for basic cryptographic applications, Dallas Semiconductor has designed and preprogrammed the Dallas Semiconductor Primary Group (Dallas Primary). While this group is necessarily limited in the variety of features that it provides, it nevertheless supports the basic features common to many different cryptographic applications. Applications based on Dallas Primary can be used with Crypto iButtons™ supplied by Dallas Semiconductor without any additional programming of the Crypto iButton™s. This simplifies distribution and reduces the amount of engineering required to develop services based on the Crypto iButton™. For example, a Service Provider wishing to provide secure e-mail services or digital notary services using Crypto iButtons™ can do so without having to add any additional information to the Crypto iButtons™.

One of the most important features provided by the Dallas Primary transaction group is the unique RSA key set generated by the Crypto iButton™. The Crypto iButton™ can be interrogated to determine the public key of this key set, but the private key will never be revealed. This feature offers an absolute assurance to the end user that the private key (on which the security of his transactions depends) will never be known to anyone. Even though it is possible for a Crypto iButton™ to be programmed with many different Transaction Groups, each having one or more RSA key sets for various purposes, the key set generated in the Dallas Primary group can be regarded for many purposes as “the” key set of the Crypto iButton™.

The feature set of Dallas Primary can be invoked with any modern programming language by making appropriate calls to the Crypto iButton™ API, which is documented in the *Cryptographic iButton™ Firmware Reference Manual* that can be downloaded from the Dallas Semiconductor website http://www.iButton.com/Crypto/index.html. In the sections which follow, the implementation of Dallas Primary is described in detail.

## III. Dallas Semiconductor Primary Group

A formal description of the U.S./Canada version of Dallas Primary is provided in Appendix A, and Appendix B contains the corresponding description of the more limited export version. The formal description consists of a symbol file which associates each symbolic representation of a data or script object with a sequential object number, and a group file which identifies the data types of the objects and defines the operations performed by the scripts. At Dallas Semiconductor these files were input to the Script Compiler, and the resulting object code was loaded directly into the Dallas Semiconductor Primary Group. This group was then locked so that it could not be altered. Note that once the data for the Dallas Primary group has been compiled with the Script Compiler, the symbolic names are no longer available and the scripts and data objects must be referenced by their numeric values as defined in the symbol file.

# Sign a Document

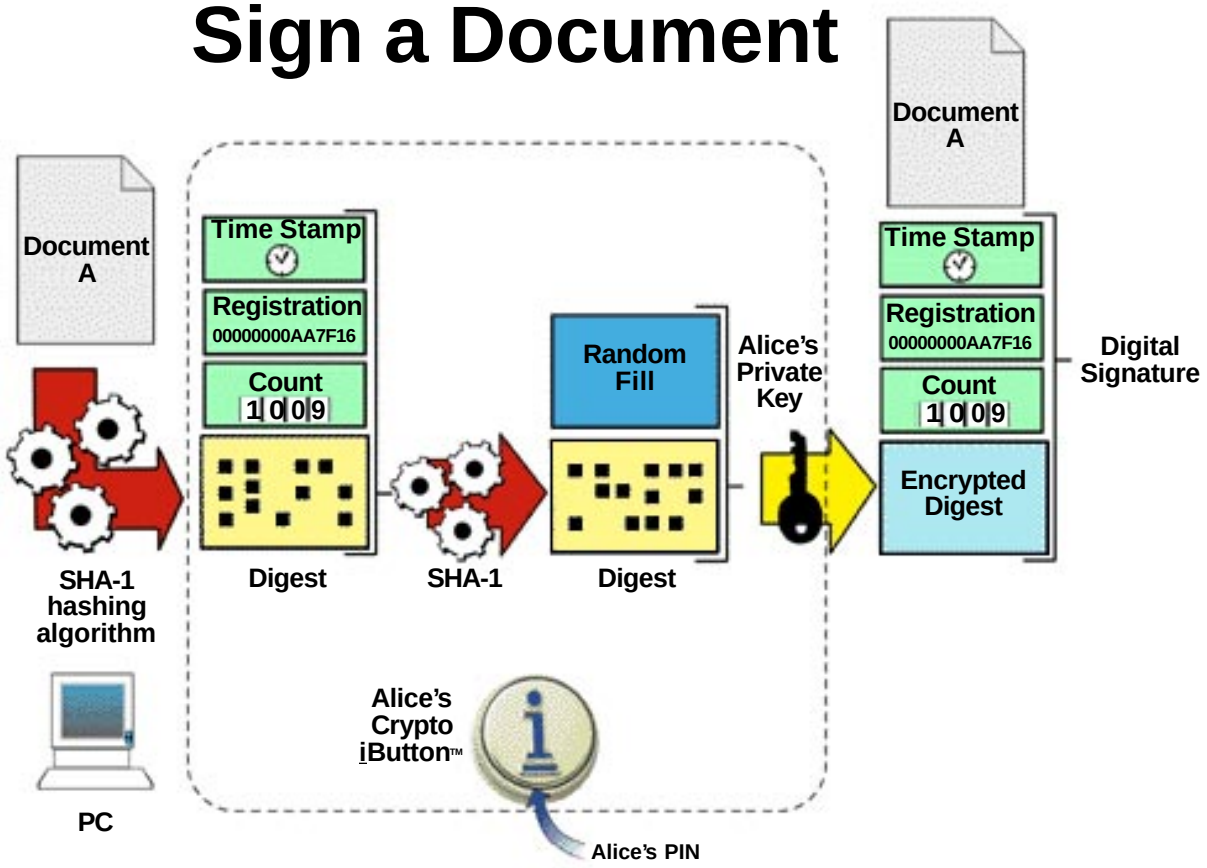


Figure 1

A. Digital Signatures and Challenge/Response Authentication

The export version of Dallas Primary supports only a single script, which is used for generating digital signatures and challenge/response authentications. The operation of this script for digitally signing a document is shown in Figure 1. This script, identified as SignCiBKey (object number 07) in the formal definitions of Appendices A and B, receives the input data to be signed in Input 1 (object number 04), appends the values of the transaction counter, unique registration number, and time stamp, and places this result in Output1 (output object number A0). It then applies the Secure Hash Algorithm SHA-1 (as defined in FIPS PUB 180-1) to this data, appends random fill to pad the data to a length one bit less than the length of the RSA modulus, and then signs the resulting data structure with the private RSA modulus and exponent. The resulting encrypted data is placed in Output2 (output object number A1). To use this function to obtain a digital signature for a document, a PC is first used to process the document with SHA-1, MD5, or other cryptographic hash function to obtain a message digest (20 bytes with SHA-1 or 16 bytes with MD5). The message digest is then written into Input 1, and the script SignCiBKey (object number 07) is invoked to perform the signature function. The data structures in Output1 and Output2 are then read from the Crypto iButton™ and appended to the original document, forming the digital signature. To use this script for challenge/response authentication, the remote party wishing to authenticate the presence of the Crypto iButton™ generates a random challenge. This challenge (typically 16 -20 bytes like the message digest of a document) is transmitted to the site where the Crypto iButton™ is located and written into Input 1 as described above. After execution of the SignCiBKey signature script, the data in Output1 and Output2 are read and transmitted back to the remote party for verification. The verification process is shown in Figure 2.

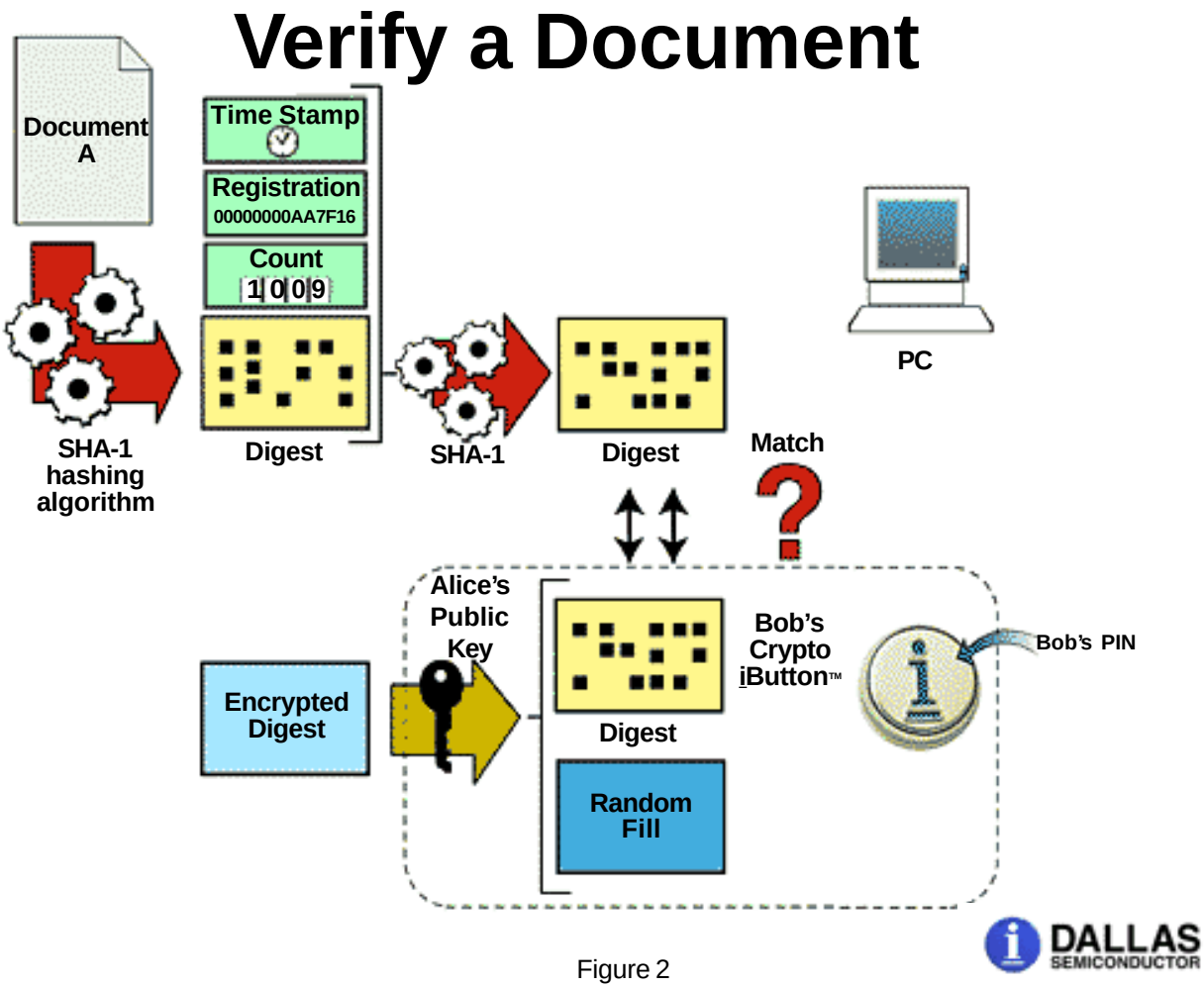


Figure 2

The remote party uses the known public key of the Crypto iButton™ it is authenticating to decrypt the encrypted data and compare the hash of the data structure containing the original challenge with the hash contained in the decrypted data structure. If these match, the remote party is assured that a specific Crypto iButton™ (i.e., the one having the private key corresponding to the known public key) was present when the challenge was issued. Note that because of the unique lasered registration number, the timestamp, and the incrementing transaction counter, this function is also suitable for implementation of a digital notary service.

The Crypto iButton™ API function calls that are required to communicate with the Crypto iButton™ to perform this digital signature operation are presented in Appendix C.

B. Encryption and Decryption of Messages and Session Keys

The U.S./Canada version of the Dallas Semiconductor Primary Group contains not only the digital signature and challenge/response support as described in section III.A. above but also data objects and scripts to enable encryption and decryption of data. The three scripts supported by this version are described below:

1. EncryptOutKey

The use of this script for encrypting an e-mail message is shown in Figure 3.

# Envelope Encryption

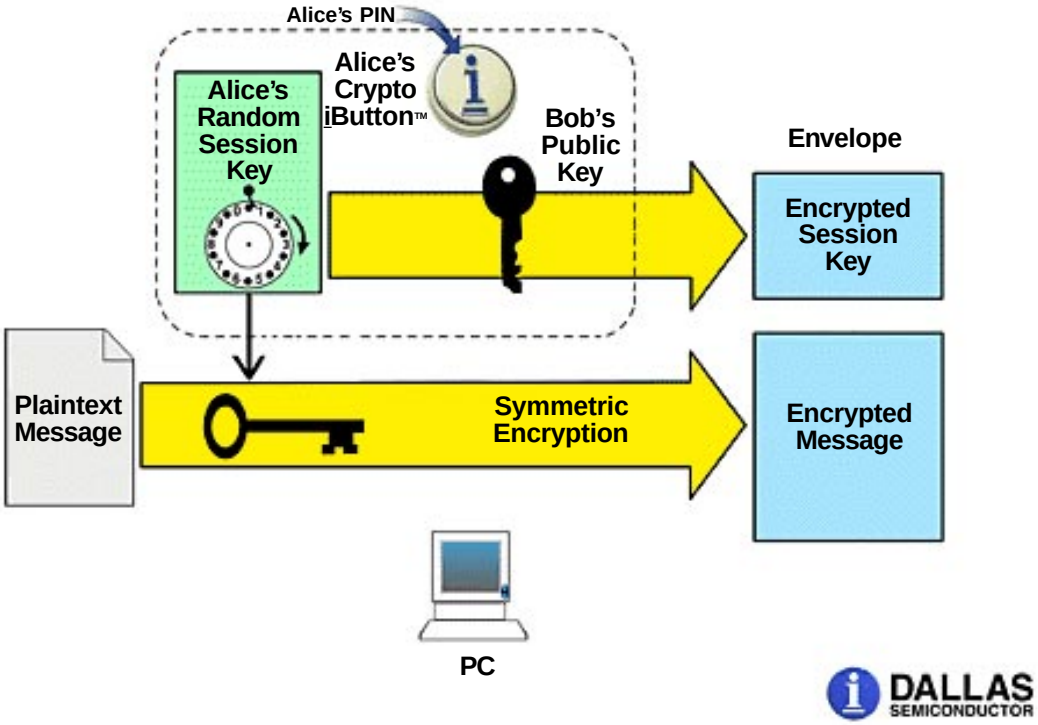


Figure 3

This script encrypts the data supplied in Input1 (object number 04) using the modulus and exponent that have been stored in the auxiliary key locations OutMod (object number 0B) and OutExp (object number 0A) of the Crypto iButton™. To send a secure e-mail by public key cryptography, the user obtains the verified public key of the intended recipient and writes its modulus and exponent into OutMod and OutExp respectively. He also generates a random one-time session key, encrypts the text of his message with this key using a fast symmetric encryption algorithm, and writes the session key into Input1. He then invokes the EncryptOutKey script (object number 0E) to perform the encryption. The result is then retrieved from Output1 (object number A0). The encrypted message is transmitted to the intended recipient along with the encrypted session key retrieved from Output1. This digital envelope encryption technique allows the message to be encrypted rapidly in the PC using any fast, secure, symmetric encryption algorithm. Note that Encrypt Out Key does not perform a secret operation, since the key used by the encryption is supplied to the Crypto iButton™ from the outside. It is provided as a convenience to the user as a fast, licensed, RSA encryption utility. (With an appropriate license from RSA Data Security, Inc., this operation could also be performed by the PC without any loss of security.)

2. DecryptCiBKey

The use of this script to decrypt an encrypted e-mail message is shown in Figure 4.

# Envelope Decryption

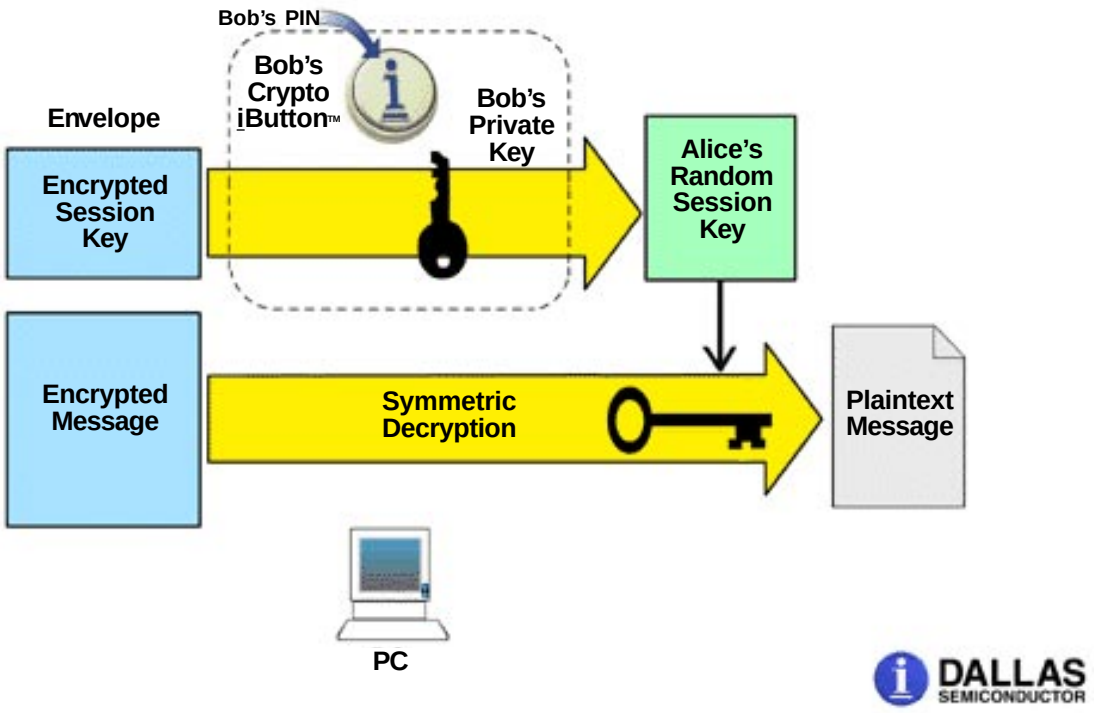


Figure 4

This script decrypts the data supplied in Input1 with the Crypto iButton™'s private key and places the result in Output1. To decrypt a secret message prepared using the EncryptOutKey function described above, the user writes the encrypted session key into Input1 and invokes the DecryptCiB Key script (object number 0D) to decrypt the session key. The decrypted session key is then retrieved from Output1 and used to decrypt the secret message in the PC using the high-speed symmetric decryption algorithm. Note that because the session key decryption operation is performed entirely within the Crypto iButton™ using the private key known only to the Crypto iButton™, this operation can be performed only by the correct Crypto iButton™.

3. EncryptCiBKey

To encrypt files containing confidential data, a user can call on the EncryptCiBKey script (object number 0C) to encrypt the session key using the Crypto iButton™'s public key. The function DecryptCiBKey described above can then be used to decrypt these files. This file encryption is secure because only the correct Crypto iButton™ can decrypt the files once they have been encrypted. The user generates a random one-time session key and then uses it as described in section III.B.1. above to encrypt the data. The data can be recovered by writing the encrypted session key into Input1, calling on DecryptCiBKey to decrypt the session key, and then decrypt the encrypted data file with the session key using the fast symmetric algorithm.

The Crypto iButton™ API function calls that are required to communicate with the Crypto iButton™ to perform these encryption and decryption operations are presented in Appendix D.

## IV. Certificates

Many applications supported by Dallas Primary such as secure e-mail and digital signature generation require some mechanism for public key verification. In the case of secure e-mail it is necessary for the sender to know the public key of the intended recipient in order to send him or her an encrypted message. For digital signature applications using the SignCiBKey script, the verifier must know the public key of the signer for use during the signature verification process.

Limited support for public key verification is provided in the form of a certificate signed by Dallas Semiconductor that is stored in the GpCertificate configuration object. The 1st step in creating this certificate is combining the public key contained within Dallas Primary, the contents of the GpInfo configuration object and the Crypto iButton™'s unique serial number. This entire packet is then hashed using the SHA-1 hashing algorithm and the resulting hash is padded with random data and signed using Dallas Semiconductor's private key.

This certificate does not bind an end user to a Crypto iButton™. It does however provide assurance that the public key in question was in fact generated by a specific Crypto iButton™ for the Dallas Primary group. Anyone wishing to validate the Crypto iButton™'s public key may do so using Dallas Semiconductor's public key. The U.S./Canada version of the Client Activation Group, "Dallas, CA2", contains a script called DalPubCrypt to perform the decryption of data signed using Dallas Semiconductor's private key. After the certificate is decrypted the hash value is extracted and compared with a hash computed as described above during the generation of GpCertificate.

A formal description of the Client Activation Group is contained in Appendix E.

# Appendix A

## U.S./Canada Version of Dallas Primary Symbol and Group Files

```
{
Symbol file for the 'Dallas Primary' group (U.S./Canada version).

    Ver 1.00 : Initial symbol file creation
    3/11/97
}
CiBPublicExp      =   01  { User Objects }
CiBModulus        =   02
CiBPrivateExp     =   03
Input1            =   04
SignCount         =   05
TimeStamp         =   06
SignCiBKey        =   07
GpInfo            =   08
GpCertificate     =   09
OutExp            =   0A
OutMod            =   0B
EncryptCiBKey     =   0C
DecryptCiBKey     =   0D
EncryptOutKey     =   0E

Output1           =   A0  { Auto objects used by Dallas Primary }
Output2           =   A1
RegNumber         =   A3
Padding           =   A4

Functions:                { Declare software ICs referenced in scripts }
    SHA1          =   01
```

---

```
{
    Group file for the 'Dallas Primary' Transaction Group (U.S./Canada version).

        Ver 1.00 : Initial group file creation
        3/11/97

    The Dallas Semiconductor Primary Group contains 4 primary scripts and their supporting data objects.  Three
    of the scripts (documented below) provide RSA encryption/decryption capability.  The fourth script is used for
    signature generation.
}
```

```
Transaction Group('Dallas Primary');

Begin
    Open:
        Input1:              InputData;
        OutExp:              Exponent;
        OutMod:              Modulus;

    Locked:
        CiBPublicExp:        Exponent;
        CiBModulus:          Modulus;
        SignCount:           Counter;
        TimeStamp:           ClockOffset;
        RegNumber:           ROMData;
        GpInfo,
        GpCertificate:       Configuration;
        Output1, Output2:    OutputData;

        EncryptCiBKey,
        DecryptCiBKey,
        EncryptOutKey,
        SignCiBKey:          Script;

    Private:
        CiBPrivateExp:       Exponent;
        Padding:             RandomFill;
End

Script EncryptCiBKey;
{
    Encrypt data supplied in the input object using the Crypto iButton™'s public key.
}
Begin
    Output1 := Input1 ^ CiBPublicExp Mod CiBModulus;
End

Script DecryptCiBKey;
{
    Decrypt data supplied in the input object using the Crypto iButton™'s private key.
}
Begin
    Output1 := Input1 ^ CiBPrivateExp Mod CiBModulus;
End

Script EncryptOutKey;
{
```

```
    This script simply provides easy access to the Crypto iButton™'s co-processor, allowing the user to perform
    an arbitrary modular exponentiation.
}
Begin
    Output1 := Input1 ^ OutExp Mod OutMod;
End

Script SignCiBKey;
{
    Sign data supplied in Input1.  This script appends the value of SignCount, RegNumber, and TimeStamp to
    Input1.  The result of these concatenations is placed in Output1.  Output1 is then hashed using SHA-1.
    Random data is appended to the hash value and the result is signed with the Crypto iButton™'s private key.
    The resulting signature is placed in Output2.
}
Begin
    Output1 := Input1 & SignCount & RegNumber & TimeStamp;
    Output2 := (SHA1(Output1) & Padding) ^ CiBPrivateExp Mod CiBModulus;
End
```

# Appendix B

## Export Version of Dallas Primary Symbol and Group Files

```
{
    Symbol file for the 'Dallas Primary' group (export version).

    Ver 1.00 : Initial symbol file creation
    3/11/97
}
CiBPublicExp      =   01  { User Objects }
CiBModulus        =   02
CiBPrivateExp     =   03
Input1            =   04
SignCount         =   05
TimeStamp         =   06
SignCiBKey        =   07
GpInfo            =   08
GpCertificate     =   09

Output1           =   A0  { Auto objects used by Dallas Primary }
Output2           =   A1
RegNumber         =   A3
Padding           =   A4

Functions:                { Declare software ICs referenced in scripts }
    SHA1          =   01
```

---

```
{
    Group file for the 'Dallas Primary' Transaction Group (export version).

        Ver 1.00 : Initial group file creation
        3/11/97

    The Dallas Semiconductor Primary Group contains one primary script and its supporting data objects.
    This script is used for signature generation.
}

Transaction Group('Dallas Primary');
```

```
Begin
    Open:
        Input1:                 InputData;

    Locked:
        CiBPublicExp:           Exponent;
        CiBModulus:             Modulus;
        SignCount:              Counter;
        TimeStamp:              ClockOffset;
        RegNumber:              ROMData;
        GpInfo,
        GpCertificate:          Configuration;
        Output1, Output2:       OutputData;

        SignCiBKey:             Script;

    Private:
        CiBPrivateExp:          Exponent;
        Padding:                RandomFill;
End

Script SignCiBKey;
{
    Sign data supplied in Input1.  This script appends the value of SignCount, RegNumber, and TimeStamp to
    Input1.  The result of these concatenations is placed in Output1.  Output1 is then hashed using SHA-1.
    Random data is appended to the hash value and the result is signed with the Crypto iButton™'s private key.
    The resulting signature is placed in Output2.
}
Begin
    Output1 := Input1 & SignCount & RegNumber & TimeStamp;
    Output2 := (SHA1(Output1) & Padding) ^ CiBPrivateExp Mod CiBModulus;
End
```

# Appendix C

## Crypto iButton™ API Calls to Generate a Digital Signature

The Crypto iButton™ API function calls that are required to communicate with the Crypto iButton™ to perform the digital signature operation are outlined below:

1. Import Constant Values from the Symbol File

   The Symbol File should be imported into the source code so that the symbolic names can be used instead of the more obscure object numbers:

```
Const
    CiBPublicExp      =  1;     { User Objects }
    CiBModulus        =  2;
    CiBPrivateExp     =  3;
    Input1            =  4;
    SignCount         =  5;
    TimeStamp         =  6;
    SignCiBKey        =  7;
    GpInfo            =  8;
    GpCertificate     =  9;

    OutExp            =  10;    { U.S./Canada Version Objects }
    OutMod            =  11;
    EncryptCiBKey     =  12;
    DecryptCiBKey     =  13;
    EncryptOutKey     =  14;

    Output1           =  160;   { Auto objects used by Dallas Primary }
    Output2           =  161;
    RegNumber         =  163;
    Padding           =  164;
```

2. Provide Function Prototypes for the Required API Calls

```
Function FindCiBs(P: Pointer): Pointer;
StdCall; External 'CIBAPI.DLL';

Function SelectCiB(P: Pointer): Boolean;
StdCall; External 'CIBAPI.DLL';

Function GetGroupID(lpGroupName, lpGroupID, lpRP: Pointer): Boolean;
StdCall; External 'CIBAPI.DLL';
```

```
Function ReadCiBObject(GroupID: Byte; lpGroupPIN: Pointer; ObjectID: Byte;
lpObject, lpRP: Pointer): Boolean; StdCall; External 'CIBAPI.DLL';

Function WriteCiBObject(GroupID: Byte; lpGroupPIN: Pointer; ObjectID: Byte;
lpObject, lpRP: Pointer): Boolean; StdCall; External 'CIBAPI.DLL';

Function InvokeScript(GroupID: Byte; lpGroupPIN: Pointer; ObjectID: Byte;
RunTime: Word; lpRP: Pointer): Boolean; StdCall; External 'CIBAPI.DLL';

Function GetCiBError: Word; StdCall; External 'CIBAPI.DLL';
```

3. Declare Constants, Types, and Variables

```
Const
    MaxPINLen      = 8;
    MaxNameLen     = 16;
    MaxPacketLen   = 128;
    MaxObjLen      = 128;
    Name : String  = 'Dallas Primary';   { Dallas Semiconductor Primary Group }
    LargeKeyTime   = 1500; { Estimated time parameter for large key encryption }
    SmallKeyTime   = 250;  { Estimated time parameter for small key encryption }

Type
    TGroupPIN = Record
        Len     : Byte;
        PINData : Array[1..MaxPINLen] of Byte;
    End;
    PGroupPin = ^TGroupPin;

    TGroupName = Record
        Len      : Byte;
        NameData : Array[1..MaxNameLen] of Byte;
    End;
    PGroupName = ^TGroupName;

    TCiBObj = Record
        Attr, OType, Len : Byte;
        ObjData : Array[1..MaxObjLen] of Byte;
    End;
    PCiBObj = ^TCiBObj;

    TRetPacket = Record
        CSB, DataLen : Byte;
        CmdData : Array[1..MaxPacketLen] of Byte;
    End;
```

```
    PRetPacket = ^TRetPacket;

Var
    lpGroupPin      : PGroupPin;
    lpGroupName     : PGroupName;
    lpObject        : PCiBObj;
    lpRP            : PRetPacket;

    GroupPin        : TGroupPin;
    GroupName       : TGroupName;
    CiBObject       : TCiBObj;
    CiBRP           : TRetPacket;
    GroupID         : Byte;
```

4. Initialize Data Structures

```
Procedure Initialize;

Begin
    lpGroupPin      := @GroupPin;
    lpGroupName     := @GroupName;
    lpObject        := @CiBObject;
    lpRP            := @CiBRP;
    With GroupName do Begin
        Len := Length(Name);
        For I := 1 to Len do NameData[I] := Byte(Name[I]);
    End;
    GroupPin.Len := 0;
End;
```

5. Establish a Communication Link to the Crypto iButton™

For example, a function such as the one below can be used to identify and link to the first Crypto iButton™ found on the 1-Wire bus:

```
Function AttachCiB: Boolean;
{
    This function uses FindCiBs and SelectCiB to establish a connection to the first Crypto iButton™
    on the 1-Wire bus.  It returns True if the attachment was successful and the device contains a
    group named 'Dallas Primary', otherwise it returns False.
}

Var
    RomPoint : Pointer;
    CiBCount : Word;

Begin
```

```
    CiBCount := 0;
    RomPoint := FindCiBs(@CiBCount);
    AttachCiB := (RomPoint <> Nil) and (CiBCount > 0) and SelectCiB(RomPoint)
    and GetGroupID(lpGroupName, @GroupID, lpRP);
End;
```

6. Transmit the Data to be Signed to the Crypto iButton™

   First put the data to be signed in the structure pointed to by lpObject, then execute a command of the form:

   ```
   F := WriteCiBObject(GroupID, lpGroupPin, Input1, lpObject, lpRP);
   ```

   (Note that if there is a Group PIN, the structure pointed to by lpGroupPin must contain the PIN. If there is no group PIN, the Len parameter of the GroupPin structure must be zero.)

   If F returns True, the operation was successful, otherwise call the GetCiBError API function to determine the type of error that has occurred. Values returned byGetCiBError and their meanings are given in the *Cryptographic iButton™ Firmware Reference Manual*. Two particular errors that should be considered are ERR_BAD_GROUP_PIN = $82 and ERR_INACTIVE = $D1. The first error occurs when the part expects a PIN but the value of the PIN specified in the GroupPIN variable is missing or incorrect. The second error occurs if the activation of the Crypto iButton™ has expired and the device requires reactivation from the Dallas Semiconductor activation website.

7. Execute the Signature Script

   Now that the data to be signed has been written into the input object designated by Input1, the signature script can be executed by calling the InvokeScript API function:

   ```
   F := InvokeScript(GroupID, lpGroupPin, SignCiBKey, LargeKeyTime, lpRP);
   ```

   Note that the value of the estimated time parameter LargeKeyTime given in Step 3 above is based on an RSA key length of 1024 bits. If the key length is less than this value, a smaller value of LargeKeyTime may be used. (The SmallKeyTime value is used when encrypting or verifying a digital signature with a small public exponent, such as 65537.)

   If the function returns True, the operation was successful, otherwise call the GetCiBError API function to determine the type of error that has occurred. On successful completion, the script leaves the results of the digital signature calculation in the output objects Output1 and Output2.

8. Read Back the Two Components of the Completed Digital Signature

   The digital signature has two components that must be read back from the Crypto iButton™. Object1 contains the exact data to be “signed” to create the digital signature or response to a challenge. This data must be collected from the Crypto iButton™ and supplied as a part of the digital signature. It contains the timestamp, the unique lasered registration number of the Crypto iButton™, the unique transaction counter number, and the challenge or message digest. The function to read this data structure is called as follows:

F := ReadCiBObject(GroupID, lpGroupPin, Output1, lpObject, lpRP);

Again, the function returns True if it was successful. The data is returned through the pointer lpObject. To create the digital signature, the Crypto iButton™ appended a packet of random fill to this data to make it one bit shorter than the modulus. Then it was signed by exponentiating it with the private key. The following function call returns the result of this operation:

F := ReadCiBObject(GroupID, lpGroupPin, Output2, lpObject, lpRP);

If it returns True, the data is returned through the pointer lpObject. The response to a challenge requires that both the data from Output1 and the data from Output2 be returned to the remote party who provided the challenge. For a digital signature, both the contents of Object1 and the contents of Object2 must be appended to the original document to form the digital signature.

Note that because of the random fill, every digital signature and response to a challenge is different, even if the message digest or challenge is the same. This feature helps to defeat certain kinds of attacks on the security of the Crypto iButton™.

# Appendix D

## Crypto iButton™ API Calls to Perform Encryption and Decryption

The Crypto iButton™ API function calls that are required to communicate with the Crypto iButton™ to perform the encryption and decryption operations are outlined below:

1. Perform Steps 1 through 5 described in Appendix C above to set up the environment and prepare the Crypto iButton™ for use.

2. EncryptOutKey

   To use this script, the intended recipient's public key must first be loaded into OutExp and OutMod, and the random session key must be loaded into Input1. Then the EncryptOutKey script must be executed, and the result retrieved from Output1:

   a. Load the Exponent

      First put the RSA exponent of the intended recipient's key into the variable pointed to by lpObject, then execute the API function:

      F := WriteCiBObject(GroupID, lpGroupPIN, OutExp, lpObject, lpRP);

      If the function returns True, the exponent has been written successfully into OutExp.

   b. Load the Modulus

      First put the RSA modulus of the intended recipient's key into the variable pointed to by lpObject, then execute the API function:

      F := WriteCiBObject(GroupID, lpGroupPIN, OutMod, lpObject, lpRP);

      If the function returns True, the modulus has been written successfully into OutMod.

   c. Load the Session Key

      First put the session key used for encrypting the message into the variable pointed to by lpObject, then execute the API function:

      F := WriteCiBObject(GroupID, lpGroupPIN, Input1, lpObject, lpRP);

      If the function returns True, the session key has been written successfully into Input1.

   d. Invoke the Script

      F := InvokeScript(GroupID, lpGroupPin, EncryptOutKey, SmallKeyTime, lpRP);

If the function returns True, the encryption has been performed successfully and the result is waiting for retrieval in Output1. Note that the use of the SmallKeyTime parameter is justified if the public exponent supplied in step 2.a. above is small. A full-length exponent requires use of LargeKeyTime.

e. Retrieve the Result

F := ReadCiBObject(GroupID, lpGroupPin, Output1, lpObject, lpRP);

If the function returns True, the session key encrypted with the intended recipient's public key is available in the variable pointed to by lpObject.

3. DecryptCiBKey

To use this script, the encrypted session key is written into Input1, the script is invoked, and the decrypted result is read from Output1:

a. Load the Encrypted Session Key

First put the session key used for encrypting the message into the variable pointed to by lpObject, then execute the API function:

F := WriteCiBObject(GroupID, lpGroupPIN, Input1, lpObject, lpRP);

If the function returns True, the encrypted session key has been written successfully into Input1.

b. Invoke the Script

F := InvokeScript(GroupID, lpGroupPin, DecryptCiBKey, LargeKeyTime, lpRP);

If the function returns True, the decryption has been performed successfully and the result is waiting for retrieval in Output1.

c. Retrieve the Result

F := ReadCiBObject(GroupID, lpGroupPin, Output1, lpObject, lpRP);

If the function returns True, the decrypted session key is available in the variable pointed to by lpObject.

4. EncryptCiBKey

To use this script, the session key is written into Input1, the script is invoked, and the encrypted result is read from Output1:

a. Load the Session Key

First put the session key used for encrypting the message into the variable pointed to by lpObject, then execute the API function:

F := WriteCiBObject(GroupID, lpGroupPIN, Input1, lpObject, lpRP);

If the function returns True, the session key has been written successfully into Input1.

b. Invoke the Script

F := InvokeScript(GroupID, lpGroupPin, DecryptCiBKey, LargeKeyTime, lpRP);
If the function returns True, the encryption has been performed successfully and the result is waiting for retrieval in Output1.

c. Retrieve the Result

F := ReadCiBObject(GroupID, lpGroupPin, Output1, lpObject, lpRP);

If the function returns True, the encrypted session key is available in the variable pointed to by lpObject.

# Appendix E

## U.S./Canada Version of the Dallas Semiconductor Client Activation Group

```
{
    Symbol file for the U.S./Canada version of the client activation group.

    Ver 1.00          : Initial symbol file creation
    11/10/96

    Ver 1.00 -> 2.00 : Added DalPubCrypt script symbol info.
}

DalPubExp          = 01     { User objects }
DalModulus         = 02
Input1             = 03
ActivateCiB        = 04
RandomChlg         = 05
GlobDest           = 06
ActiveSeconds      = 07
DalPubCrypt        = 08

Output1            = A0     { Auto objects used by Dallas, CA2 }
Temp               = A2
```

---

```
{
    Group file for the U.S./Canada version of the client activation group. This group is created as the 1st transaction
    group in every Crypto iButton issued in the U.S. and Canada.

    Ver 1.00          : Initial group file creation
    11/10/96

    Ver 1.00 -> 2.00  : Added DalPubCrypt script to assist with
    3/11/97           decryption of certificates signed with the
                      Dallas Semiconductor private key.
}

TransactionGroup('Dallas, CA2');

Begin
    Open:
        Input1:       InputData;
```

```
    Locked:
        DalPubExp:      Exponent;
        DalModulus:     Modulus;
        RandomChlg:     SALT;
        GlobDest:       Destructor;
        ActiveSeconds:  ClockOffset;
        Output1:        OutputData;
        ActivateCiB,
        DalPubCrypt:    Script;

    Private:
        Temp:           WorkingRegister;
End

Script ActivateCiB;
{
    Input1 is supposedly a packet (containing the SALT RandomChlg and a number of seconds for the Crypto
    iButton to remain active) signed using Dallas Semiconductor's private key. ActivateCiB decrypts the packet
    using Dallas Seiconductor's public key and places the results in Temp (the working register).
    If the signed challenge matches the last challenge generated by the CiB, the seconds count passed in the
    signed packet is added to the current value of the CiB's RTC (real time clock) and the result is placed in the
    destructor.
}
Begin
    Temp := Input1 ^ DalPubExp Mod DalModulus;
    Temp.SALT[1] = RandomChlg;   { Abort if SALTs don't match }
    RandomChlg := RandomChlg;   { Generate new challenge }
    ActiveSeconds := Temp.Configuration[1];
    GlobDest := ActiveSeconds;   { Destructor = ActiveSeconds + RTC }
End

Script DalPubCrypt;
{
    This script is typically used to decrypt a certificate signed using Dallas Semiconductor's private key.
}
Begin
    Output1 := Input1 ^ DalPubExp Mod DalModulus;
End
```