RSA BSAFE®

Crypto-C

Cryptographic Components for C

User’s Manual

Version 4.2

RSA Data Security.
A Security Dynamics Company

Contents

Figures and Tables

Figures

Tables

Introduction

Dear Crypto-C Developer:

Congratulations on your purchase of RSA BSAFE® Crypto-C 4.2, the state-of-the-art in cryptographic software toolkits! Crypto-C provides developers with the most important privacy, authentication, and data integrity routines. Crypto-C contains a full palette of popular cryptographic algorithms. This toolkit enables you to develop applications for a wide range of purposes, including electronic commerce, home banking, Webcasting, and enterprise security.

Crypto-C is written in C and is intended to be completely portable. It is available on a number of platforms and can be ported to most platforms with a minimum of effort. Crypto-C is a toolkit, not an application; it is intended to be integrated into operating systems, communications systems, and other applications. Therefore, you have a modest amount of work ahead of you. We have tried to make this task as clear as possible without limiting your alternatives. This *User's Manual,* with its code samples and tutorials, is the best place to start.

Thanks, and welcome to the RSA family.

Sincerely,

The Crypto-C Development Team
RSA Data Security, Inc.

The Crypto-C Toolkit

Crypto-C provides developers with a state-of-the-art implementation of the most important privacy, authentication, and data integrity routines. The following algorithms are implemented in Crypto-C 4.2:

Symmetric Ciphers

- DES
- Triple-DES
- DESX
- RC2
- RC4
- RC5

Message Digests

- MD
- MD2
- MD5
- SHA1

Message Authentication

- HMAC

Random Number Generation

- MD2
- MD5
- SHA1

Public-Key Algorithms

- RSA Public Key Cryptosystem
- Diffie-Hellman Key Agreement

Digital Signatures

- DSA

- RSA Digital Signatures

Elliptic Curve Public-Key Algorithms

- Elliptic Curve Digital Signature Algorithm (ECDSA)
- Elliptic Curve Diffie-Hellman Key Agreement
- Elliptic Curve Authenticated Encryption Scheme (ECAES)

Secret Sharing

- Bloom-Shamir Secret Sharing

Cryptographic Standards and Crypto-C

Crypto-C is a general-purpose programming tool that developers can use to write a wide variety of applications. Crypto-C was built to permit developers to make use of the Public-Key Cryptography Standards (PKCS) series of documents, which specify a standard way of performing basic cryptographic operations. Several higher-level standards, such as S/MIME, SET, IPSec, and SSL, require implementation of various PKCS standards. Since Crypto-C complies with PKCS, developers should find integrating Crypto-C into software implementing these standards to be a fairly easy task.

To obtain copies of the PKCS electronically on the Internet, see the PKCS section of RSA Data Security, Inc.'s web site, which is accessible via `http://www.rsa.com/rsalabs`. Alternatively, you may contact our sales department for a diskette.

Crypto-C and the Year 2000

Software applications that rely only on the last two digits of the current date field might behave erratically in the next century when the date changes to the year 2000. RSA Data Security, Inc. is frequently asked about how our products will handle the Year 2000 issue and what assurances we can provide our software development partners.

Crypto-C does not invoke time and date services, so it does not inherently have any Year 2000 issues to deal with.

However, Crypto-C accounts for only a portion of any particular application, and those applications might introduce Year 2000 bugs independent of the Crypto-C code. In turn, those applications might rely on the underlying platform and operating system for time and date services, which might introduce their own Year 2000 bugs into any application that uses our toolkits.

How to Reach RSA Data Security, Inc.

Developer Support

RSA Data Security, Inc. is committed to helping you effectively integrate our security into your applications. For details on our support plans, please contact a Telesales Representative at 650-295-7600, or view our support options online at `http://support.rsa.com`.

Web Site

In addition, you can reach the RSA Data Security, Inc. Web site at `http://www.rsa.com`. RSA Data Security, Inc. has pages for security bulletins, coming events, free software and publications, and an ftp site. RSA Data Security, Inc. also has a developer's corner at `http://www.rsa.com/rsa/developers/`. If you are interested in cryptography, RSA Data Security, Inc.'s Cryptography FAQ is available at `http://www.rsa.com/rsalabs/faq/`.

Conventions Used in This Manual

Italic is used for:

- new terms where they are introduced
- the names of manuals and books

`Lucida Typewriter Sans` is used for:

- anything that appears literally in a C program, such as the names of structures and functions supplied by Crypto-C: for example, `B_DecodeInit`

`Lucida Typewriter Sans Italic Bold` is used for:

- function parameters and placeholders that indicate that an item is replaced by some actual value in your own program: for example, ***`randomAlgorithm`***

`Lucida Typewriter Bold` is used for:

- text the user types in command line demos and text that is printed to the screen by the demos (only)

Structures and routines defined by Crypto-C are boxed:

```
/* Structures defined by Crypto-C */
```

Application code and samples are displayed in a box with a shaded outline:

```
/* Application code and samples */
```

Some Crypto-C functions are only available when used with a hardware application that has a BSAFE Hardware API interface (BHAPI). These functions are marked with the icon of a hammer.

Chapter 1

Quick Start

Organization

Chapter 1, the Quick Start, uses a code example to describe the basic encryption and decryption operations in Crypto-C.

Chapter 2 presents a brief outline of the basic cryptographic principles and terminology that are used in this manual.

Chapter 3 presents a brief description of the Crypto-C algorithm info types and key info types by functionality. It also covers system considerations when using Crypto-C.

Chapters 4-7 present sample code for the major Crypto-C operations.

Chapter 8 presents sample code for the BSAFE Hardware API (BHAPI).

describes the command line demos.

lists reference documents.

The Six-Step Sequence

The Crypto-C model generally follows a six-step sequence:

1. Create
2. Set
3. Init
4. Update
5. Final
6. Destroy

In addition, for every application, you must include the necessary header files; we will call this Step 0.

The six-step sequence makes it easier to maintain your code. For example, if you have implemented a message digest routine using MD2 and wish to use SHA1 instead, you simply need to make changes in Steps 2 and 3, Set and Init. The rest of your code can be reused. Similarly, if you originally programmed a routine under the assumption that it would get all the data from a single buffer, and you wish to modify it to take data from multiple buffers, you can simply change Step 4, Update.

Note: In some cases, an algorithm may not require an Update step.

The sections in this chapter show the following:

- a six-step encryption example
- a six-step decryption example
- using multiple Updates
- a summary of the six-step process

Introductory Example

The CD containing the Crypto-C library distribution also includes sample source code to accompany this *User's Manual*. One of the files on that CD, `introex.c`, is an example converting the Introductory Example into a program. Later in this manual are instructions on writing code for many Crypto-C operations. There are sample programs on the CD to accompany all the topics covered.

With the *Crypto-C Library Reference Manual* handy, we will encrypt the sentence, "Encrypt this sentence." To do this, we will use what is called a *stream cipher*, that is, an encryption method that encrypts data a character at a time, in a single stream. The cipher we will use is called RC4. This cipher can take a key size from 1 to 256 bytes. RC4 creates a "key stream" based on the key and XORs the stream of data with the key stream to create ciphertext.

The example in this section corresponds to the file `introex.c`.

Step 0: Include Files

You must include the necessary header files and the Crypto-C library in every application you write using Crypto-C:

```
#include "aglobal.h"
#include "bsafe.h"
#include "demochos.h"
```

When writing a Crypto-C application, include `aglobal.h` and `bsafe.h` in that order. If you wish to use the `DEMO_ALGORITHM_CHOOSER` (see "Selecting an Algorithm Chooser" on page 15), include `demochos.h` after `bsafe.h`. In addition, you must compile and link in `tstdlib.c`, which contains the memory management functions called by the Crypto-C library.

Note: For backward compatibility, the BSAFE 2.x `include` file names, `global.h` and `bsafe2.h`, are still valid. If your source code contains the older names, you should not have any problems.

Step 1: Creating an Algorithm Object

Whatever operation Crypto-C performs, it does so from an *algorithm object*. An algorithm object is used to hold information about an algorithm's parameters and to keep a context during a cryptographic operation such as encryption or decryption.

For our example, we will build an algorithm object that performs encryption.

You build an algorithm object in Steps 1 to 3. As you go through these steps, you specify the type of algorithm that is being used, supply any special information or parameters that the algorithm requires, and generate or supply a key for algorithms that need one.

In Step 1, we simply create the object. We do this by declaring a variable to be an algorithm object and calling `B_CreateAlgorithmObject`.

In this case, we name our algorithm object ***rc4Encrypter*** and declare it as follows:

```
B_ALGORITHM_OBJ rc4Encrypter = (B_ALGORITHM_OBJ)NULL_PTR;
```

The data type `B_ALGORITHM_OBJ` is defined in `bsafe.h`:

```
typedef POINTER B_ALGORITHM_OBJ;
```

where `POINTER` is defined in `aglobal.h`:

```
typedef unsigned char *POINTER;
```

and `NULL_PTR` is also defined in `aglobal.h`:

```
#define NULL_PTR ((POINTER)0)
```

So our variable, ***rc4Encrypter***, is a pointer. To prevent problems when the algorithm object is destroyed, it is a good idea to initialize it to `NULL_PTR`. See Step 6 for details.

To create an algorithm object, we call `B_CreateAlgorithmObject`. Chapter 4 of the *Library Reference Manual* gives the function prototypes and descriptions of all the Crypto-C calls. For `B_CreateAlgorithmObject`, we find:

```
int B_CreateAlgorithmObject (
  B_ALGORITHM_OBJ *algorithmObject                 /* new algorithm object */
);
```

Because `B_CreateAlgorithmObject` takes a pointer to a `B_ALGORITHM_OBJ` as its argument, we have to pass the address of ***rc4Encrypter***. The return value is an `int`. Most Crypto-C calls return either a 0 (zero), which indicates success, or a non-zero error code. After the call, look at the return value: if it is 0, continue; if not, stop. At

RSA Data Security, Inc., the tradition is to name the return value ***status***:

```
int status;
do {
  if ((status = B_CreateAlgorithmObject (&rc4Encrypter)) != 0)
    break;
        .
        .
        .
} while (0);
```

Standard RSA Data Security, Inc., coding practices use the above `do-while` construct to make it easy to break out of a sequence when encountering an error. If a Crypto-C function returns a non-zero value, `break` will exit the `do-while`, and further code dependent on the offending call will not be executed. However, any clean-up code, such as overwriting sensitive memory with zeroes (see Step 6), can follow the `do-while` and will always execute, whether or not there was an error.

Step 2: Setting the Algorithm Object

The variable ***rc4Encrypter*** is now an algorithm object, but we have not yet determined what type of operations it can perform. In Step 2, we associate the algorithm object with an algorithm and supply any special information or parameters the algorithm requires. We do this with `B_SetAlgorithmInfo`. Chapter 4 of the *Library Reference Manual* gives this function's prototype and description:

```
int B_SetAlgorithmInfo (
  B_ALGORITHM_OBJ algorithmObject,                   /* algorithm object */
  B_INFO_TYPE     infoType,             /* type of algorithm information */
  POINTER         info                       /* algorithm information */
);
```

The first argument is ***rc4Encrypter***. But what are the next two? The second argument is an *algorithm info type*, or AI. In Crypto-C, you specify the type of operation an algorithm object performs by setting the object to a particular AI. Chapter 2 of the *Library Reference Manual* describes the available AIs. Each AI description also lists the information that must accompany that AI when setting an algorithm object. That accompanying information is the third argument of `B_SetAlgorithmInfo`.

For our example, we want to choose a stream cipher AI. A stream cipher processes data in a stream of arbitrary length. This is in contrast to another common type of cipher, the block cipher, which processes data in blocks of a fixed size. In Crypto-C, there is a single stream cipher, RC4, and a number of AIs that can be used to implement it. For this example we will use `AI_RC4`; we pass this as the second argument to `B_SetAlgorithmInfo`.

The third argument is information that is specific to the AI we chose. For complex algorithms, this is input that is required by the algorithm, such as: parameters for algorithms that require them, "salt" and the desired number of iterations for password-based encryption, or an "initialization vector" for block ciphers. In our example, `AI_RC4` is a simple algorithm that does not require any parameters; its entry in Chapter 2 of the *Library Reference Manual* states that the format of the ***info*** supplied to `B_SetAlgorithmInfo` is `NULL_PTR`.

Thus, we can make the call to `B_SetAlgorithmInfo`:

```
if ((status = B_SetAlgorithmInfo
      (rc4Encrypter, AI_RC4, NULL_PTR)) != 0)
  break;
```

Note: Once you have set an algorithm object, do not set it again. If you need an algorithm object to perform another type of operation, create a new one.

Step 3: Init

Now that we have created and set our algorithm object, ***rc4Encrypter***, it is ready to encrypt. Actually, since we haven't called `B_EncryptInit`, it is ready to decrypt as well. In Step 3, we choose the operations our algorithm object can perform by supplying the desired function pointers to the Crypto-C library; we also create and set a key object that will supply the key data the algorithm needs.

Note: An algorithm object can be used for either encryption or decryption, but not for both. You should create separate algorithm objects to handle each case.

Look at the entry for `AI_RC4` in Chapter 2 of the *Library Reference Manual*:

Crypto-C procedures to use with algorithm object:
`B_EncryptInit`, `B_EncryptUpdate`, `B_EncryptFinal`;
and `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`.
You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all ***randomAlgorithm*** arguments.

From this, you can see that `AI_RC4` can be used with encryption or decryption procedures; that is, it can be used to encrypt or to decrypt. We want to encrypt, so in Step 3, we will call `B_EncryptInit` to initialize our algorithm object to perform encryption. This call will also associate a key with the algorithm object.

See the description and prototype in Chapter 4 of the *Library Reference Manual* for `B_EncryptInit`:

```
int B_EncryptInit (
  B_ALGORITHM_OBJ     algorithmObject,                 /* algorithm object */
  B_KEY_OBJ           keyObject,                             /* key object */
  B_ALGORITHM_CHOOSER algorithmChooser,               /* algorithm chooser */
  A_SURRENDER_CTX    *surrenderContext                /* surrender context */
);
```

As in Step 2, the first argument is the algorithm object; once again, we use ***`rc4Encrypter`***. The next three arguments are new.

Step 3a: Creating a Key Object

The second argument is a key object, which is used to hold any key-related information, such as the RC4 key, and to supply this information to functions that require it. Before we can pass a key object as an argument, we must create and set it. Creating a key object is similar to creating an algorithm object. We name our key object ***`rc4Key`*** and declare it as follows:

```
B_KEY_OBJ rc4Key = (B_KEY_OBJ)NULL_PTR;
```

where `B_KEY_OBJ` is defined in `bsafe.h`:

```
        typedef POINTER B_KEY_OBJ;
```

Chapter 4 of the *Library Reference Manual* gives the description and prototype of `B_CreateKeyObject`:

```
int B_CreateKeyObject (
  B_KEY_OBJ *keyObject                                  /* new key object */
);
```

For our example, we use:

```
if ((status = B_CreateKeyObject (&rc4Key)) != 0)
  break;
```

Step 3b: Setting a Key Object

We have a key object, but it is not yet distinguished as an RC4 key. For that we need to use `B_SetKeyInfo`. See Chapter 4 of the *Library Reference Manual* for this function's description and prototype:

```
int B_SetKeyInfo (
  B_KEY_OBJ   keyObject,                                /* key object */
  B_INFO_TYPE infoType,                       /* type of key information */
  POINTER     info                                 /* key information */
);
```

This function is similar to `B_SetAlgorithmInfo`. The first argument is the key object just created, ***rc4Key***. The second argument is a key info type (KI), and the third argument is information that must accompany the given KI. We want to use a KI compatible with RC4 encryption, so we return to the entry for our AI, `AI_RC4`, in Chapter 2 of the *Library Reference Manual*:

Key info types for *keyObject* in `B_EncryptInit` **or** `B_DecryptInit`:
`KI_Item` that gives the address and length of the RC4 key.

Key info types are described in Chapter 3 of the *Library Reference Manual*. Under the entry for `KI_ITEM` we find that the format of ***info*** supplied to `B_SetKeyInfo` is a pointer to an `ITEM` structure:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

len is the length of the key in bytes. RC4 takes key sizes of one to 256 bytes. A ten-byte key is generally sufficient for most applications. ***data*** is the key data. A real application would use a random number generator to produce ten bytes for the key (see "Generating Random Numbers" on page 147). For this example, we can simply use:

```
static unsigned char rc4KeyData[] = {
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10
};

ITEM rc4KeyItem;
rc4KeyItem.data = rc4KeyData;
rc4KeyItem.len = sizeof(rc4keyData);
```

Now we can complete the call to `B_SetKeyInfo`:

```
if ((status = B_SetKeyInfo
      (rc4Key, KI_Item, (POINTER)&rc4KeyItem)) != 0)
  break;
```

As with algorithm objects, once you have set a key object, you should not set it again. If you need another key object, you should create a new one.

Note: In a real application, for security reasons, you might want to zeroize and free your key data immediately after setting the key.

Now that we have created and set our key object, ***`rc4Key`***, we can pass it as the second argument to `B_EncryptInit`.

Selecting an Algorithm Chooser

The third argument to `B_EncryptInit` is an *algorithm chooser*; this is a structure that specifies which algorithm methods to link in. An algorithm method (AM) is the underlying code that actually performs the cryptographic operation. Because many AIs can perform more than one cryptographic function (for example, `AI_RC4` can perform encryption and decryption), an application often has a choice of which underlying algorithm method(s) need to be linked in.

An algorithm chooser lists all the AMs the application will use; only these AMs will be linked in. Crypto-C comes with a demonstration application containing the algorithm chooser `DEMO_ALGORITHM_CHOOSER`. You can use this algorithm chooser in any Crypto-C application as long as the module which defines it (`choosc.c`) is compiled and linked in. However, `DEMO_ALGORITHM_CHOOSER` will link in all the algorithm methods available, even though an application might use only two or three.

A developer can write an algorithm chooser for the specific application to make the executable image smaller. See “Algorithm Choosers” on page 118. in this manual for instructions on writing an algorithm chooser. For this example, we will use DEMO_ALGORITHM_CHOOSER as the third argument of B_EncryptInit.

Surrender Context

The fourth argument of B_EncryptInit is a *surrender context*, which controls when and how the application surrenders control during time-consuming operations. The application developer can put together an A_SURRENDER_CTX structure containing a surrender function and other information. Crypto-C applications call this surrender function at regular intervals.

The surrender function can simply print out information to the user that indicates that the Crypto-C operation is currently executing, or it can provide the user with a means of halting the operation if it is taking too much time. A surrender context is not required; if none is desired, simply pass a properly cast NULL_PTR. See "The Surrender Context" on page 120. for a more detailed description of the A_SURRENDER_CTX structure. For this example, we will use (A_SURRENDER_CTX *)NULL_PTR.

We can now complete our call to B_EncryptInit:

```
if ((status = B_EncryptInit
      (rc4Encrypter, rc4Key, DEMO_ALGORITHM_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

In Steps 1 through 3, we created our algorithm object and initialized it with the information that it needs to perform RC4 encryption. In Step 4, we can enter the data to encrypt with the B_EncryptUpdate function. Chapter 4 of the *Library Reference Manual* provides the following description and prototype:

```
int B_EncryptUpdate (
  B_ALGORITHM_OBJ  algorithmObject,                  /* algorithm object */
  unsigned char   *partOut,                         /* output data buffer */
  unsigned int    *partOutLen,                   /* length of output data */
  unsigned int     maxPartOutLen,           /* size of output data buffer */
  unsigned char   *partIn,                                /* input data */
  unsigned int     partInLen,                    /* length of input data */
  B_ALGORITHM_OBJ  randomAlgorithm,                 /* random byte source */
  A_SURRENDER_CTX *surrenderContext                  /* surrender context */
);
```

The first argument is our algorithm object, ***rc4Encrypter***.

The other arguments call for the plaintext input and encrypted output. Because the output depends on the input, we start with the fifth and sixth arguments, which describe the input.

We name our input ***dataToEncrypt*** and declare it as follows:

```
static char dataToEncrypt[] = “Encrypt this sentence.”;
```

Crypto-C needs to know how many bytes our input is, so we use `strlen`:

```
unsigned int dataToEncryptLen;
dataToEncryptLen = (unsigned int)strlen (dataToEncrypt) + 1;
```

If your data is not a string — that is, if it does not end with a `NULL` terminating character — do not use `strlen` to determine its length.

The output is described by the second, third, and fourth arguments.

The second argument is described in the prototype as `unsigned char *partOut`. This does *not* mean you simply declare a variable to be `unsigned char *` and pass it as the argument. The output argument that you pass is a pointer to a buffer of allocated memory. This is an important point; see “Algorithm Choosers” on page 118 for a detailed discussion of this topic.

For now, we declare:

```
unsigned char *encryptedData = NULL_PTR;
```

For a stream cipher, the length of the encrypted (output) data is equal to the length of the input data. So we allocate ***dataToEncryptLen*** bytes with `T_malloc`:

```
encryptedData = T_malloc (dataToEncryptLen);
if ((status = (encryptedData == NULL_PTR)) != 0)
  break;
```

The code above uses the Crypto-C routine `T_malloc`. Crypto-C supplies its own memory management routines to increase code portability and to meet the special requirements of handling encrypted data. The Crypto-C memory management routines reside in the file `tstdlib.c`; make sure this file is compiled and linked in.

These routines are described in Chapter 4 of the *Library Reference Manual* and in "Memory-Management Routines" on page 123 of this manual.

In our example, the `T_malloc` routine from `tstdlib.c` returns a pointer to the allocated memory. If, for some reason, it cannot allocate memory (for example, when there is not enough memory available), `T_malloc` will return `NULL_PTR`. It is imperative to always check the return value of `T_malloc`, even if you are allocating only a small number of bytes. `T_malloc` also sets an `unsigned char *` variable; it is a good idea to initialize this variable to `NULL_PTR`. See "Step 6: Destroy" on page 20. for more information.

The third argument to `B_EncryptUpdate` is a pointer to an `unsigned int`. `B_EncryptUpdate` returns a value indicating how many bytes it placed into the output buffer. It will place this value at the address specified by the pointer to the `unsigned int`. Make the proper declaration:

```
unsigned int outputLenUpdate;
```

Crypto-C might not encrypt all the input data during a call to `B_EncryptUpdate`. Any unprocessed data will be saved in a buffer inside the algorithm object created by Crypto-C and encrypted during a subsequent call to Update (see "Multiple Updates" on page 28) or during the call to `B_EncryptFinal` (see "Step 5: Final" on page 19). This is why it is important to keep track of how many bytes Crypto-C wrote to the output buffer.

The fourth argument to `B_EncryptUpdate` is the size of the output buffer. The Update function must know the size of the buffer. The Update function will not attempt to place data into unallocated memory; instead, it returns an error if it needs to place more bytes into the buffer than are allocated. In our example, we will use ***`dataToEncryptLen`*** as our output data size.

The seventh argument is a random algorithm. Recall that in Chapter 2 of the *Library Reference Manual*, the description of `AI_RC4` states:

You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all ***`randomAlgorithm`*** arguments.

That is exactly what we will supply in our example.

For the eighth argument, once again, we pass a properly cast `NULL_PTR` as the surrender context. When we put this all together, our Update call is:

```
if ((status = B_EncryptUpdate
      (rc4Encrypter, encryptedData, &outputLenUpdate,
      dataToEncryptLen, dataToEncrypt, dataToEncryptLen,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

encrypted DataLen = outputLenUpdate + outputLenFinal
```

Note the warning in the *Library Reference Manual* Chapter 2 entry for `AI_RC4`:

> Due to the nature of the RC4 algorithm, security is compromised if multiple data blocks are encrypted with the same RC4 key. Therefore, `B_EncryptUpdate` cannot be called after `B_EncryptFinal`. To begin an encryption operation for a new data block, you must call `B_EncryptInit` and supply a new key.

This simply means that you should not use the same key for two different encryption sessions.

Step 5: Final

`B_EncryptFinal` finalizes the encryption process by encrypting any data that `B_EncryptUpdate` could not. See Chapter 4 of the *Library Reference Manual* for the function's description and prototype:

```
int B_EncryptFinal (
  B_ALGORITHM_OBJ  algorithmObject,                    /* algorithm object */
  unsigned char   *partOut,                          /* output data buffer */
  unsigned int    *partOutLen,                     /* length of output data */
  unsigned int     maxPartOutLen,              /* size of output data buffer */
  B_ALGORITHM_OBJ  randomAlgorithm,                  /* random byte source */
  A_SURRENDER_CTX *surrenderContext                   /* surrender context */
);
```

For our example, the first argument is ***rc4Encrypter***.

The second argument is a pointer to the output buffer that we created for `B_EncryptUpdate`. However, `B_EncryptUpdate` has already placed some data into that buffer, so we must pass the address of the next byte that is available *after* the already filled bytes to `B_EncryptFinal`. That is the address of the beginning of the buffer plus the number of bytes that `B_EncryptUpdate` filled, or ***encryptedData*** + ***outputLenUpdate***.

The third argument is a pointer to an `unsigned int`; `B_EncryptFinal` will set that `unsigned int` to the number of bytes it encrypted.

The fourth argument is the size of the buffer available to `B_EncryptFinal`. Because `B_EncryptUpdate` has already written to part of the buffer, this value will be the total size of the buffer minus the number of bytes `B_EncryptUpdate` has used, or ***dataToEncryptLen*** - ***outputLenUpdate***.

Once again, we can pass properly-cast null pointers for the fifth and sixth arguments, which are the random algorithm and surrender context.

Then, for our example, we have:

```
if ((status = B_EncryptFinal
      (rc4Encrypter, encryptedData + outputLenUpdate,
      &outputLenFinal, dataToEncryptLen - outputLenUpdate,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

When you are done with an algorithm or key object, you must destroy it. The Destroy function frees up any memory that was allocated by Crypto-C and zeroizes any sensitive memory. Because you will always want to destroy the objects, place these function calls after the `do-while` construct. That way, even if there is an error somewhere and the program breaks out of the `do-while` before executing all the calls within the `do-while`, the Destroy functions will execute. In case the error occurs before an object has been created, it is a good idea to initialize objects to `NULL_PTR`. If an object is `NULL_PTR`, the Destroy function does nothing.

Chapter 4 of the *Library Reference Manual* gives the description and prototype of the Destroy functions:

```
void B_DestroyKeyObject (
  B_KEY_OBJ       *keyObject                     /* pointer to key object */
);
void B_DestroyAlgorithmObject (
  B_ALGORITHM_OBJ *algorithmObject         /* pointer to algorithm object */
);
```

For our example, we use the following:

```
B_DestroyKeyObject (&rc4Key);
B_DestroyAlgorithmObject (&rc4Encrypter);
```

In addition to destroying any objects that you created, any memory you allocated must be freed when you are done with it. This means that each `T_malloc` must have a corresponding `T_free`. Placing the `T_free` after the `do-while` guarantees that it will be called even if there is an error somewhere. However, there is a concern that if there is an error before the `T_malloc` and the program breaks out of the `do-while` before memory is allocated, then `T_free` will be called without a corresponding `T_malloc`. That is why it is important to initialize the pointer to `NULL_PTR`. If the argument to `T_free` is `NULL_PTR`, the extra call to `T_free` does nothing.

See Chapter 4 of the *Library Reference Manual* for the `T_free` prototype:

```
void T_free (
  POINTER block                                          /* block address */
);
```

For this example, call `T_free` as follows:

```
T_free (encryptedData);
```

Note: Using `T_free` means you can no longer access the data at that address. Do not free a buffer until you no longer need the data it contains. If you will need the data later, you might want to save it to a file first.

You may want to zeroize any sensitive data before you free it. To do this, duplicate the following sequence after the `do-while`. If there is an error inside the `do-while` before you zeroize and free, you are still guaranteed to perform this important task:

```
if (rc4KeyItem.data != NULL_PTR) {
  T_memset (rc4KeyItem.data, 0, rc4KeyItem.len);
  T_free (rc4KeyItem.data);
  rc4KeyItem.data = NULL_PTR;
  rc4KeyItem.len = 0;
}
```

Putting It All Together

Now we can put Steps 0 through 6 into a program. This program can be found in the file `introex.c`:

```
#include "aglobal.h"
#include "bsafe.h"
#include "demochos.h"

void PrintBuf PROTO_LIST ((unsigned char *, unsigned int));

void main()
{
  B_KEY_OBJ rc4Key = (B_KEY_OBJ)NULL_PTR;
  B_ALGORITHM_OBJ rc4Encrypter = (B_ALGORITHM_OBJ)NULL_PTR;

  /*  The RC4 key is hard-coded in this example. In a real application,
      use a random number generator to produce the key.  */
  unsigned char rc4KeyData[10] = {
      0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10
  };

  static char dataToEncryp[] = "Encrypt this sentence.";
  unsigned char *encryptedData = NULL_PTR;
  unsigned int dataToEncryptLen, encryptedDataLen;
  unsigned int outputLenUpdate, outputLenFinal;
  unsigned int status;

  do {
    dataToEncryptLen = strlen (dataToEncrypt) + 1;

    /*  Step 1:  Create an algorithm object.  */
    if ((status = B_CreateAlgorithmObject (&rc4Encrypter)) != 0)
      break;

    /*  Step 2:  Set the algorithm to a type that does rc4 encryption.
                 AI_RC4 will do.  */
    if ((status = B_SetAlgorithmInfo
         (rc4Encrypter, AI_RC4, NULL_PTR)) != 0)
      break;

    /*  Step 3a:  Create a key object.  */
    if ((status = B_CreateKeyObject (&rc4Key)) != 0)
      break;
```

```
/*  Step 3b:  Set the key object with the 10-byte key.  */
rc4KeyItem.data = rc4KeyData;
rc4KeyItem.len = rc4KeyDataLen;

if ((status = B_SetKeyInfo
      (rc4Key, KI_Item, (POINTER)&rc4KeyItem)) != 0)
  break;

if (rc4KeyItem.data != NULL_PTR) {
  T_memset (rc4KeyItem.data, 0, rc4KeyItem.len);
  T_free (rc4KeyItem.data);
  rc4KeyItem.data = NULL_PTR;
  rc4KeyItem.len = 0;
}

/*  Step 3:  Init  */
if ((status = B_EncryptInit
      (rc4Encrypter, rc4Key, DEMO_ALGORITHM_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

/*  Step 4:  Update  */
encryptedData = T_malloc (dataToEncryptLen);
if ((status = (encryptedData == NULL_PTR)) != 0)
  break;

if ((status = B_EncryptUpdate
      (rc4Encrypter, encryptedData, &outputLenUpdate,
      dataToEncryptLen, (unsigned char *)dataToEncrypt,
      dataToEncryptLen, (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

/*  Step 5:  Final  */
if ((status = B_EncryptFinal
      (rc4Encrypter, encryptedData + outputLenUpdate,
      &outputLenFinal, dataToEncryptLen - outputLenUpdate,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

encryptedDataLen = outputLenUpdate + outputLenFinal;
printf ("Encrypted data (%u bytes):\n", encryptedDataLen);
PrintBuf (encryptedData, encryptedDataLen);
```

```
  } while (0);

  /*  Done with the key and algorithm objects, so destroy them.  */
  B_DestroyKeyObject (&rc4Key);
  B_DestroyAlgorithmObject (&rc4Encrypter);

  /*  Free up any memory allocated, save it to a file or print it out first
      if you need to save it.  */
  if (rc4KeyItem.data != NULL_PTR) {
    T_memset (rc4KeyItem.data, 0, rc4KeyItem.len);
    T_free (rc4KeyItem.data);
    rc4KeyItem.data = NULL_PTR;
    rc4KeyItem.len = 0;
  }

  if (encryptedData != NULL_PTR){
    T_memset (encryptedData, 0, dataToEncryptLen);
    T_free (encryptedData);
    encryptedData = NULL_PTR;
  }

} /*  end main  */
```

You may find it a useful exercise to compile and link this program. Also, it could also be instructive to add some print statements. For instance, what are the values of ***outputLenUpdate*** and ***outputLenFinal***?

While it is possible to print the ***encryptedData***, it will not be an ASCII string — it is not any kind of string, because there is no `NULL` terminating character. The encrypted data is binary data, so it may be more useful to print out the result byte-by-byte in hex-ASCII strings. For an example of a function that does this, see the `PrintBuf()` routine in the sample program. In addition, note that when writing Crypto-C output to (and reading it from) files, it is usually more useful (in some cases, even necessary) to open the files in binary mode.

To run this exercise, first compile `introex.c`, `tstdlib.c`, and `choosc.c`. Then link the object files with `bsafe.lib` or the equivalent platform-specific library.

Decrypting the Introductory Example

Decrypting data is similar to encrypting. RC4 is symmetric-key encryption, which means the key that was used to encrypt will be the key needed for decryption.

The example in this section corresponds to the file `dintroex.c`.

Step 1: Creating an Algorithm Object

```
B_ALGORITHM_OBJ rc4Decrypter = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&rc4Decrypter)) != 0)
  break;
```

Step 2: Setting the Algorithm Object

Use the same AI and parameters as for encryption:

```
if ((status = B_SetAlgorithmInfo
      (rc4Decrypter, AI_RC4, NULL_PTR)) != 0)
  break;
```

Step 3: Init

Use the same key data as for encryption. Once again, we must create and set the key object.

Step 3a: Creating the Key Object

As before, we name our key object ***rc4Key*** and declare it as follows:

```
B_KEY_OBJ rc4Key = (B_KEY_OBJ)NULL_PTR;
```

Then we allocate space for the key object using `B_CreateKeyObject`:

```
if ((status = B_CreateKeyObject (&rc4Key)) != 0)
  break;
```

Step 3b: Setting the Key Object

We need to fill our key with the same ten bytes of data we used for encryption. We must make sure that we use the same key as we used to encrypt. For our sample application, we can simply re-create the key data we had before:

```
static unsigned char rc4KeyData[] = {
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10
};
```

Now we can complete the call to `B_SetKeyInfo`:

```
if ((status = B_SetKeyInfo
    (rc4Key, KI_Item, (POINTER)&rc4KeyData)) != 0)
  break;
```

Step 4: Update

Here, we must set the buffer that will store the decrypted data; for RC4, it should be the same size as the encrypted data's buffer:

```
unsigned char *decryptedData = NULL_PTR;

decryptedData = T_malloc (encryptedDataLen);
if ((status = (decryptedData == NULL_PTR)) != 0)
  break;

if ((status = B_DecryptUpdate
      (rc4Decrypter, decryptedData, &decryptedLenUpdate,
      encryptedDataLenTotal, encryptedData, outputLenTotal,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

```
if ((status = B_DecryptFinal
      (rc4Decrypter, decryptedData + decryptedLenUpdate,
      &decryptedLenFinal, encryptedDataLen - decryptedLenUpdate,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

In the “Introductory Example” on page 9, the plaintext was a string. Therefore, we can compute the sum of ***decryptedLenUpdate*** and ***decryptedLenFinal*** to determine how many characters make up the decryption.

Note: For some algorithms, the decrypted data may not be a string — for example, when the `NULL` terminating character was not encrypted. In these cases, if you want to print the decrypted data, you will not be able to because the data is in binary form, not ASCII. You could print the ginary data using `PrintBuf()`, or you can convert the decrypted data. Crypto-C offers encoding and decoding functions to convert between binary and ASCII. See “Converting Data Between Binary and ASCII” on page 154 for more information.

Step 6: Destroy

Always destroy objects when you no longer need them:

```
B_DestroyAlgorithmObject (&rc4Decrypter);

if (decryptedData != NULL_PTR) {
  T_memset (decryptedData, 0, encryptedDataLen);
  T_free (decryptedData);
  decryptedData = NULL_PTR;
}
```

Multiple Updates

An application can do multiple Updates before the Final call. For example, suppose you have data from three different files that you want to encrypt into a single buffer. You could do this in three steps: read the contents of the first file into a buffer; read the next file, appending the contents to the end of the existing buffer; then append the contents of the third. But that would be clumsy if the contents of the three files are already in three buffers.

You do not have to put data together into a single buffer to encrypt it. Instead, call `B_EncryptUpdate` with the first buffer, call it a second time with the second buffer, and one last time with the third buffer. Then call `B_EncryptFinal` once, after you have finished all Updates. Similarly, you can call `B_DecryptUpdate` more than once with blocks of encrypted data.

Multiple updates can also be useful for encrypting or decrypting large amounts of data. If you need to process a one-megabyte file, you could allocate a megabyte of memory, put the entire file into that memory buffer, and call Update once. But using such a large amount of memory is impractical or even impossible in some situations. An application is more robust if it allocates a smaller buffer — say, 64, 128 or 1024 bytes — transfers data from the file in increments, and processes each unit with a separate call to Update. Then it can call Final once for all Updates.

Crypto-C does not always encrypt or decrypt an entire block during an Update call. One reason it might not handle the whole block is because of padding. Padding is used with block ciphers to ensure the data satisfies input restrictions and may add bytes to the original data. See "Padding" on page 36 for more information. Padding and pad operations (encrypting or decrypting the padding and/or stripping the pad) take place in Final, so Crypto-C may keep the last few bytes of any input to an Update call in a buffer. If there is another call to Update, then the bytes in that buffer were not the last bytes of input, and Crypto-C continues to encrypt or decrypt. If the next call is to Final, the bytes in the buffer are the last bytes of input, so Crypto-C adds the pad and encrypts it, or decrypts the final bytes and strips the pad.

Note: The output of a particular Update may very well be larger than the input, because Crypto-C may be processing the current input plus some data in the buffer. Hence, an output buffer of an Update call should always be larger than the input length. For block ciphers, for example, the size of the output buffer may be as large as the length of the input plus the block size.

The following example demonstrates multiple Updates. It corresponds to the file `multencr.c`; a similar example for decryption is in the file `multdecr.c`. Assume that the subroutine ***`GetDataFromFile`*** gets at most a specified number of bytes from a file, places them into the given buffer, and sets a flag indicating whether the bytes returned are the last ones in the file or not. Assume also that the subroutine ***AppendDataToFile*** appends output data to a file. Finally, assume we have already called `B_CreateAlgorithmObject`, `B_SetAlgorithmInfo`, and `B_EncryptInit`:

```
#define UPDATE_SIZE         64
#define UPDATE_OUTPUT_SIZE  (UPDATE_SIZE + 16)

  FILE *inputFile = (FILE *)NULL_PTR;
  FILE *outputFile = (FILE *)NULL_PTR;

  unsigned char dataToEncrypt[UPDATE_SIZE];
  unsigned char blockOfEncryptedData[UPDATE_OUTPUT_SIZE];
  unsigned int dataToEncryptLen, totalBytesSoFar;
  unsigned int outputLenUpdate, outputLenFinal;
  unsigned int sizeToUpdate = UPDATE_SIZE;
  int endFlag, status;

  do {

    totalBytesSoFar = 0;

    while ((status = GetDataFromFile
            (inputFile, sizeToUpdate, dataToEncrypt,
             &dataToEncryptLen, &endFlag)) == 0) {
      printf ("dataToEncryptLen = %i \n", dataToEncryptLen);
      PrintBuf (dataToEncrypt, dataToEncryptLen);
      if ((status = B_EncryptUpdate
           (encryptionObject, blockOfEncryptedData,
            &outputLenUpdate, UPDATE_OUTPUT_SIZE, dataToEncrypt,
            dataToEncryptLen, (B_ALGORITHM_OBJ)NULL_PTR,
            (A_SURRENDER_CTX *)NULL_PTR)) != 0)
        break;

      /*  Save the encrypted data.  */
      if ((status = AppendDataToFile
           (outputFile, blockOfEncryptedData,
            outputLenUpdate)) != 0)
        break;

      totalBytesSoFar += outputLenUpdate;
      if (endFlag == 1)
        break;
    } /*  end while  */
    /*  If there was an error in the above while loop, break out of the
          do-while construct.  */
    if (status != 0)
      break;

    /*  Call B_EncryptFinal once after all Updates.  */
    if ((status = B_EncryptFinal
         (encryptionObject, blockOfEncryptedData, &outputLenFinal,
          UPDATE_OUTPUT_SIZE, (B_ALGORITHM_OBJ)NULL_PTR,
          (A_SURRENDER_CTX *)NULL_PTR)) != 0)
      break;

    /*  Save the encrypted data.  */
    if ((status = AppendDataToFile
         (outputFile, blockOfEncryptedData,
          outputLenFinal)) != 0)
      break;

    totalBytesSoFar += outputLenFinal;

  } while (0);

  /*  Free up any memory allocated, save it to a file or print it out first
      if you need to save it.  */

  T_memset (dataToEncrypt, 0, sizeof (dataToEncrypt));
```

In the above code, we took ***dataToEncryptLen*** bytes of data to encrypt and passed them to `B_EncryptUpdate`. The number of bytes of output may or may not be ***dataToEncryptLen***; check ***outputLenUpdate*** to see. If fewer than ***dataToEncryptLen*** bytes were output, the as-yet-unencrypted input waits in a buffer.

Notice that we did not allocate memory, but used the stack; we did this by declaring our buffers to be arrays of `unsigned char`. This means that the operating system will do the allocating and freeing.

Also notice the call to `T_memset`, another memory management routine from `tstdlib.c`. `T_memset` sets all the bytes of a buffer to a particular value; in this case, it wrote a 0 to every byte in ***dataToEncrypt***. `T_memset` is described in Chapter 4 of the *Library Reference Manual*. When memory is freed, whether by a call to `T_free` or automatically by the operating system, the data still exists at that location; the operating system has simply marked that area as available for use. For security, overwrite any memory that held sensitive data when you are done with it. This prevents attackers from reconstructing secrets by examining your computer’s memory.

Summary of the Six Steps

A typical implementation uses the six steps as follows:

Step 0: Include

Include the necessary header files. In addition, make sure that:

- your compiler can locate the Crypto-C header files
- your compiler can locate and link in the Crypto-C library
- you compile and link in the file containing the definitions for the `T_` functions; an example is provided in `tstdlib.c`.

Step 1: Create

Create an algorithm object by declaring a variable to be an algorithm object and calling `B_CreateAlgorithmObject.`

Step 2: Set

Use `B_SetAlgorithmInfo` to associate the algorithm object with an algorithm and to supply any special information or parameters the algorithm requires.

Step 3: Init

Choose the operations the algorithm object can perform by supplying the desired algorithms methods from the Crypto-C library. If the algorithm requires a key, create and set a key object that will supply the key data that the algorithm needs.

Step 4: Update

Initiate an action. The action depends on the algorithm. Update is the only step that can be performed more than once on the same object. For example:

- For an encryption or decryption algorithm, an Update step encrypts or decrypts all or part of the data. You can use multiple Update steps to encrypt or decrypt data.
- For a message digest, the Update step is used to enter the data to digest.
- For a random number generator, the Update step is used to seed the random number generation.

- For some algorithms, such as generating a public/private key pair, there is no Update step.

Step 5: Final

Finalize the action initiated in Step 4. Again, the finalization depends on the algorithm; for some algorithms, Final is replaced by Generate. For example:

- For an encryption or decryption algorithm, the Final step encrypts or decrypts the final portion of the data. For some algorithms, this data may need special handling, such as “padding,” that is different from the Update step.
- For a message digest, the digest action takes place during Final.
- For a random number generator, the Final (or Generate) step generates the random bytes.
- For generating a public/private key pair, the key pair generation takes place in the Generate step.

Step 6: Destroy

Free any memory allocated in the previous steps and overwrite any sensitive memory with zeroes. The Destroy step is crucial to the security of an application.

Chapter 2

Cryptography

This section presents a brief outline of the basic cryptographic principles and terminology used throughout this manual. The publications listed in , "References and Reading Material", on page 293 provide more comprehensive discussions of cryptographic functions and operations.

Cryptography Overview

Symmetric-Key Cryptography

In *symmetric-key cryptography*, as Figure 2-1 shows, the data used to build the encrypting key is the same data required to build the decrypting key. Using any other key to decrypt will produce incorrect results. Symmetric-key cryptography is also sometimes called secret-key cryptography, because the key used to both encrypt and decrypt must be kept secret.

There are two categories of symmetric encryption algorithms, *block ciphers* and *stream ciphers*. As the name implies, a block cipher processes data in blocks. A stream cipher, on the other hand, processes a unit of data at a time, where a unit is generally a bit or byte. This allows a stream cipher to take in a variable length stream of data, encrypt it, and output a stream of ciphertext the same length as the input. Crypto-C offers DES, Triple DES, DESX, RC2, and RC5 as block ciphers and RC4 as a stream cipher.

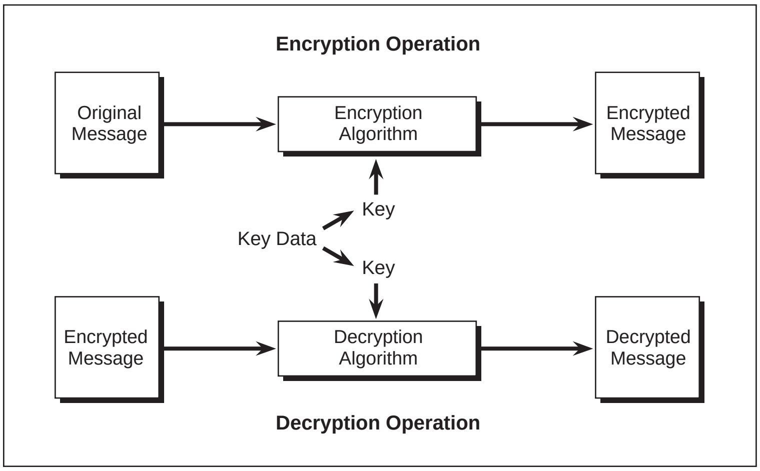

Figure 2-1 **Symmetric-Key Encryption and Decryption**

Block Ciphers

Block ciphers encrypt data block-by-block. They can encrypt each block separately as in ECB mode, or they can use other modes to make the cipher less vulnerable to attacks based on regular patterns. A mode of operation usually combines the underlying cipher with feedback and other simple operations. The security remains a function of the cipher and not of the mode. See "Modes of Operation" on page 40 for more information.

Padding

When you encrypt a message using a block cipher, usually your message length will not be a multiple of the block size. Some modes can deal with variable size blocks, but others require the message be a multiple of the block size. For these modes, padding is a way to deal with this problem. To pad, you add a regular pattern of bytes to the end of the last block to make it a complete block. With padding, the actual number of bytes encrypted can be as much as one block more than the original data.

Ciphers

Crypto-C implements the following block ciphers:

- DES
- Triple DES

- RC2
- RC5
- DESX

DES

The Digital Encryption Standard, DES, is a commercial encryption US standard that has been available for over 15 years. The federal standard document FIPS PUB 46-2 describes the algorithm. DES, in most cases, is not an exportable algorithm.

For DES, the block size is eight bytes. Therefore, the input must be a multiple of eight bytes, or else it must be padded to be a multiple of eight bytes for DES to operate in CBC or ECB modes properly. The key consists of 56 random bits and 8 parity bits, forming a 64-bit, or 8-byte, key.

Triple DES

Triple DES executes DES three times, which triples the number of bits in an encryption key. A number of different methods achieve this function. The technique that Crypto-C uses is depicted in Figure 2-2 on page 38.

This technique is known as EDE, or "Encrypt-Decrypt-Encrypt." The decryption process in the middle stage of Triple DES encryption provides compatibility with DES. If the three keys are the same, the Triple DES operation is equivalent to a single DES encryption. That way, an application that has only DES capabilities can still communicate with applications that use Triple DES. If the three keys are different, the decryption in the middle will scramble the message further; it will not decrypt the first stage. Triple DES decryption is the inverse operation of the above sequence, that is, DES decryption followed by DES encryption and then another DES decryption. Triple DES is generally not exportable.

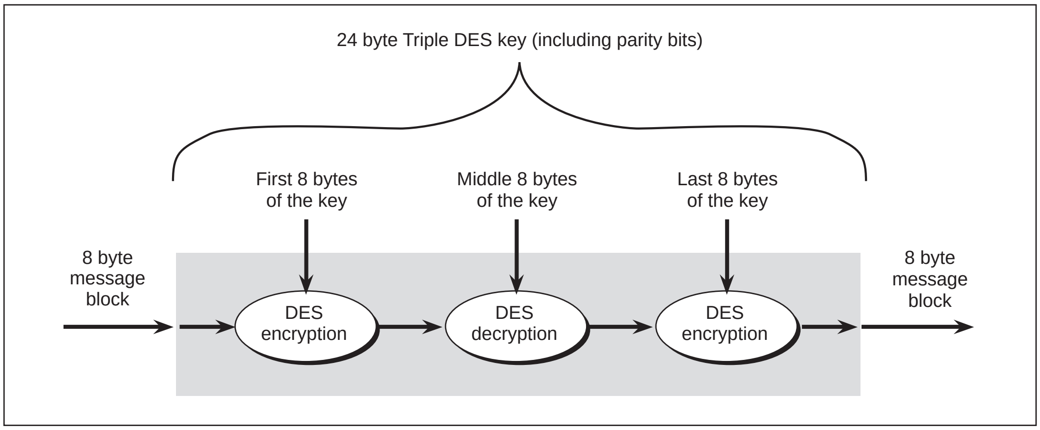

Figure 2-2 **Triple DES encryption as implemented in Crypto-C**

DESX

DESX is an RSA Data Security, Inc. proprietary extension of the DES encryption algorithm that increases the effective number of key bits from 56 to 120 bits. Crypto-C includes DESX for backward compatibility with BSAFE 1.x versions, or as a faster alternative to Triple DES.

RC2

RC2 was developed by Ronald Rivest as an alternative to DES encryption; it is proprietary to RSA Data Security, Inc. RC2 has an eight-byte block size. Therefore, the input must be a multiple of eight bytes, or be padded to be a multiple of eight bytes, for RC2 to operate properly in CBC or ECB modes.

The RC2 input key can be of any length from 1 to 128 bytes. The algorithm uses the input key to generate an *effective key* that is actually used for encryption purposes. Internally, the algorithm builds a key table based on the bits of the key data; the chosen number of effective key bits limits the number of possible key tables. The effective key size is variable and takes values from one bit up to 1024 bits.

Control over your effective key size benefits you as follows:

- You can generate up to 128 bytes of key data and set the algorithm to use a smaller number of effective bits, such as 80. Then, in the future, if you want to increase the effective key bits, you do not have to change the code that generates the key data, only the effective key bit parameter.

- In code that is being exported, you only need to modify the number of effective key bits instead of making extensive modifications to your code. RC2 can generally be approved for export under a limited key size of 40 – 48 bits; applications with 40 bits will usually be expedited.

RC5

RC5 was developed by Ronald Rivest as an alternative to DES encryption; it is proprietary to RSA Data Security, Inc. It is a block cipher with the block being either 4 bytes, 8 bytes, or 16 bytes, depending on the word size. The input must be a multiple of the block size, or it must be padded to a multiple of the block size for RC5 to operate properly. RC5's speed and security are dependent on input parameters determined by the user. These parameters are:

- word size
- rounds
- key size

Word size generally refers to the size of a hardware register. For hardware implementations of RC5, developers can take advantage of larger registers to increase speed. On chips with smaller registers, the word size can be emulated in software. RC5 version 1.0 accepts word sizes of 16, 32, or 64 bits. Crypto-C accepts a word size of 32 or 64 bits; however, the 64-bit implementation is an unoptimized evaluation implementation. The block size is twice the word size. For a word size of 32 bits, the block size is 64 bits, or 8 bytes, the same as for DES and RC2. For a word size of 64 bits, the block size is 128 bits, or 16 bytes.

The number of *rounds* is the number of times the operation employs the inner encryption function. Varying the number of rounds allows developers to make a tradeoff between speed and security. The greater the number of rounds, the greater the security, but the slower the execution. The number of rounds can be anywhere from 0 (zero) to 255. For RC5 with a 32-bit word size, RSA Data Security, Inc. recommends at least 12 rounds for applications; while no practical attacks are known for 12-round RC5-32, recent cryptanalytic work suggests 16 rounds is now a more conservative choice. For RC5 with a 64-bit word size, RSA Data Security, Inc. recommends at least 16 rounds; a conservative choice for the 64-bit version is 20 rounds. Note that the Crypto-C implementation of the 64-bit word is for evaluation purposes only.

The *key size* can be as little as 0 (zero) and as many as 255 bytes. This variable key size is intended to make it easier to obtain export permission. RC5 uses the secret key bytes to generate an expanded key table during the Init phase. The key table is then used during encryption or decryption. Hence, key length will have no appreciable effect on algorithm speed.

RC5 is more formally described as RC5 $w/r/b$. For instance, RC5 with a 32-bit word, 12 rounds, and a 10 byte key would be described as RC5 32/12/10.

Modes of Operation

When you use a block cipher to encrypt a message of arbitrary length, you can also choose a *mode of operation*.

Modes of operation can use techniques such as feedback or chaining to make identical plaintext blocks encrypt to different ciphertext blocks. Modes are designed so that they do not weaken the security of the underlying cipher, but they may have properties in addition to those inherent in the basic cipher.

Most of the modes of operation in Crypto-C are *feedback modes*. Feedback modes use the previous block of output to alter the current block of input before encrypting. In this way, encrypting the same block of plaintext twice will virtually never produce the same ciphertext.

A feedback algorithm requires an *initialization vector*, or IV, to alter the first block. The IV has no cryptographic significance. It is used to alter the first block of data before any encryption takes place; therefore, it does not need to be secret. It should be random, though, so that the first block of encrypted data is not predictable. In order to start the decryption process, it is necessary to use the IV that was employed in the encryption process.

Four Modes

Crypto-C offers four modes:

- Electronic Codebook (ECB) mode
- Cipher Block Chaining (CBC) mode
- Cipher Feedback (CFB) mode
- Output Feedback (OFB) mode

A brief description of these modes follows. Most cryptography texts, such as Bruce Schneier's *Applied Cryptography* [15], provide full descriptions of the various modes.

Electronic Codebook (ECB) Mode

ECB is not a feedback mode; it encrypts each block of input independently of all other blocks. Plaintext patterns are not concealed; instead each identical block of plaintext yields an identical block of ciphertext. This could help an eavesdropper break the code. In addition, the plaintext can be easily manipulated by removing, repeating, or interchanging blocks. The speed of each encryption operation is identical to that of the block cipher. ECB mode is as secure as the underlying block cipher.

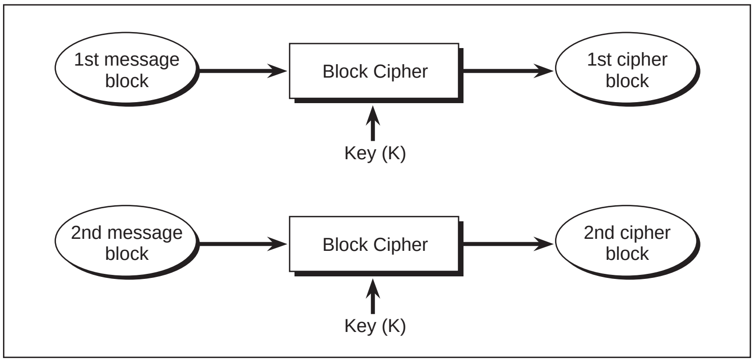

Figure 2-3 **Electronic Codebook (ECB) Mode**

Cipher Block Chaining (CBC) Mode

With CBC mode, each plaintext block is XORed with the previous ciphertext block, then encrypted. CBC mode is as secure as the underlying block cipher against standard attacks. In addition, any patterns in the plaintext are concealed by the XORing of the previous ciphertext block with the plaintext block.

The decryptor follows the same sequence of steps to decrypt, using the same (secret) key and IV.

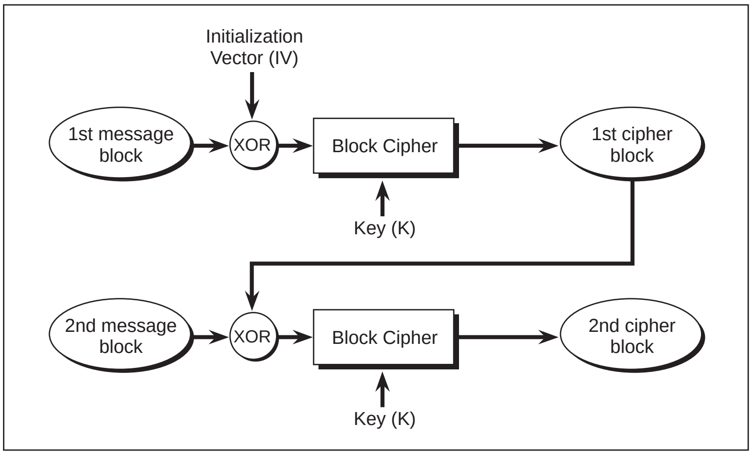

Figure 2-4 **Cipher-Block Chaining (CBC) Mode**

An initialization vector is added to the beginning of the plaintext before encryption. This gives you something to XOR the first block with and ensures that identical plaintexts encrypt to different ciphertexts.

Cipher Feedback (CFB) Mode

In cipher feedback (CFB) mode, the cipher object acts as a byte generator. CFB mode encrypts the previous block of ciphertext, and XORs the plaintext with this block to produce ciphertext. For the first block, the initialization vector is encrypted. CFB mode is as secure as the underlying cipher against standard attacks. In addition, any patterns in the plaintext are concealed by XORing the previous ciphertext block with the plaintext block.

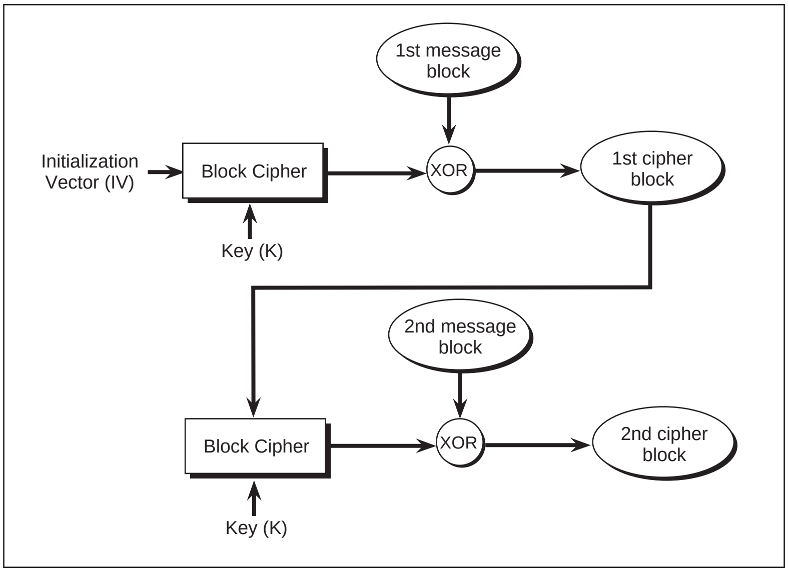

Figure 2-5 **Cipher Feedback (CFB) Mode**

To encrypt a plaintext using CFB mode:

1. Generate your key and your IV.
2. Encrypt the IV with the key to get a block of output, B_1.
3. XOR B_1 with the first block of your plaintext, P_1, to get the first block of ciphertext, C_1.
4. Encrypt C_1 with the key to get the second block of output, B_2.
5. XOR B_2 with the second block of your plaintext message, P_2, to get the second block of ciphertext, C_2.
6. Repeat Steps 4 and 5 until the entire text is encrypted.

To decrypt the ciphertext, the decryptor uses the same (secret) key and initialization vector and follows the same sequence of steps.

CFB mode does not require padding. If your data length is not a multiple of the block size, simply truncate the final block of output to be the same size as the final segment of the data, and then XOR. You can use CFB to encrypt a stream of data.

Output Feedback (OFB) Mode

Output feedback mode is similar to CFB mode, except that the quantity XORed with each plaintext block is generated independently of both the plaintext and the ciphertext.

To encrypt a plaintext using OFB, first generate the "output" used for encryption. This is intermediate data that is used in the encryption process. In OFB, the output depends only on the key and the initialization vector.

1. Generate your key and your IV.
2. Encrypt the IV with the key to get a block of output, B_1.
3. Encrypt B_1 with the key to get the second block of output, B_2.
4. Continue encrypting recursively: encrypt B_i to get B_{i+1}.

This process gives you an arbitrarily long sequence of pseudo-random blocks that you can use to encrypt the data. To use the output to encrypt:

5. XOR your plaintext with the output, block by block. The result of the XOR is the ciphertext.

OFB does not require padding. If your data length is not a multiple of the block size, simply truncate the final block of the output to be the same size as the final segment of the data, and then XOR.

The decryptor can use the same (secret) key and IV to generate the same sequence of output blocks, and XOR the sequence with the ciphertext to recover the plaintext.

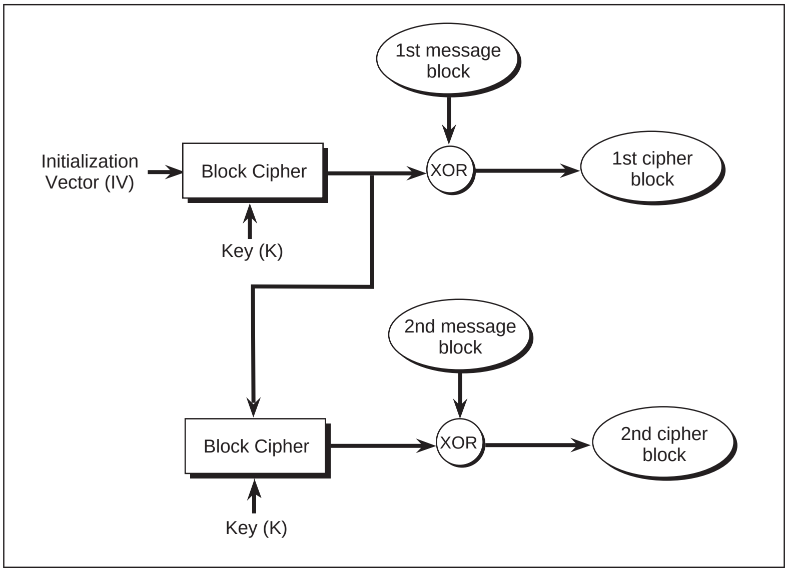

Figure 2-6 **Output Feedback Mode (OFB)**

Stream Ciphers

A *stream cipher* processes the input data a unit at a time. A unit of data is generally a byte, or sometimes even a bit. In this way, encryption or decryption can execute on a variable length of input. The algorithm does not have to wait for a specified amount of data to be input before processing, or append and encrypt extra bytes.

RC4

RC4 is a symmetric stream-encryption algorithm developed by Ronald Rivest and proprietary to RSA Data Security, Inc. It is actually a keyed pseudo-random sequence. It uses the provided key to produce a pseudo-random number sequence which is then XORed with the input data. This means that the encryption and decryption operations are identical.

The number of key bits is variable and ranges from eight to 2048 bits. An application that uses RC4 with a key size of 40 – 48 bits is generally exportable; a key size of 40 bits is usually expedited. RC4 with a key size less than 40 bits is not recommended.

Because RC4’s encryption is an XOR between the message bytes and the pseudo-random byte stream generated from the key, the same key should not be used more than once. Otherwise, if some of the bytes of one input message are known (or easy to guess), an attacker would be able to determine some of the original message bytes by XORing two sets of cipher bytes.

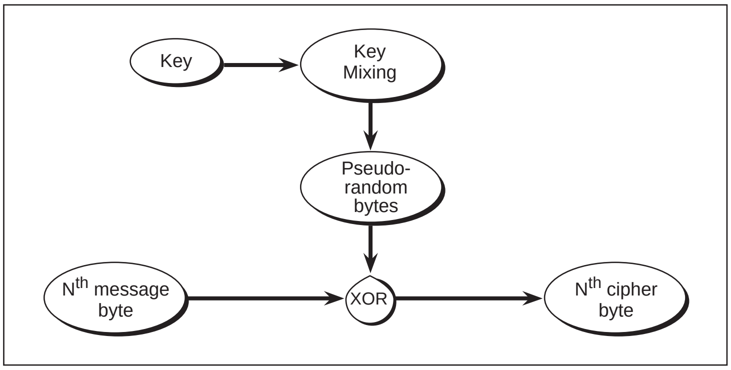

Figure 2-7 **RC4 Encryption or Decryption**

RC4 with MAC

The RC4 with MAC algorithm is an extension of RC4. It provides data integrity by using a Message Authentication Code (MAC) in conjunction with the RC4 encryption algorithm. The authentication code does not provide cryptographic authentication; rather, it provides the equivalent of a checksum that can be used to determine if any errors were introduced within the cipher bytes. The MAC guards against transmission or retrieval errors but may not detect deliberate tampering with the data.

Message Digests

A message digest (also sometimes referred to as a one-way hash function) is a fixed-length computationally unique identifier corresponding to a set of data. That is, each unit of data (a file, a string, a buffer, etc.) will map to a particular short block, called a message digest. It is not random: digesting the same unit of data with the same message digest algorithm will always produce the same short block.

A good message digest algorithm possesses the following qualities:

- The algorithm accepts any input data length.
- The algorithm produces a fixed length output for any input data.

- It is computationally infeasible to produce data that has a specific digest. In other words, given a particular block of the proper size, it will be virtually impossible to determine a unit of data that will digest to that particular block.
- It is computationally infeasible to produce two different units of data that produce the same digest. In other words, given some data, it is virtually impossible to create different data that will digest to the same block as the first. This quality is also called *collision-free*.

Message digests have many uses. They can authenticate data, for instance. To create an digest for authentication, digest the data and save the digest. Later, if you need to see if the data has been altered, digest it again and compare the new digest to the old. If the digests are different, the data is different. Although there will exist other sets of data that will digest to the original value, it is virtually impossible to find them. Minor changes in data will produce very different digests.

Crypto-C includes the MD, MD2, MD5, and SHA1 message digest algorithms. MD is included for backward compatibility with BSAFE 1.x. MD, MD2, and MD5 produce a 16-byte digest for any input message; SHA1 produces a 20-byte digest. MD5 is the fastest message digest algorithm implemented in Crypto-C.

Recent cryptanalytic work has discovered a collision in MD2's internal compression function, and there is some chance that the attack on MD2 may be extended to the full hash function. The same attack applies to MD. Another attack has been applied to the compression function on MD5, though this has yet to be extended to the full MD5. RSA Data Security, Inc. recommends that before you use MD, MD2, or MD5, you should consult the RSA Laboratories web site at `http://www.rsa.com/rsalabs` to be sure that their use is consistent with the latest information. One bulletin that discusses this issue is *Recent Results for MD2, MD4, and MD5*; it can be found at `http://www.rsa.com/rsalabs/html/bulletins.html`.

Message Digests and Pseudo-Random Numbers

Random number generation (for software implementation, usually pseudo-random number generation) is a key component of cryptographic operations. Random numbers are usually used as cryptographic keys or as a basis for generating keys. Crypto-C uses message digest algorithms with a random seed for generating random numbers. See "Pseudo-Random Numbers and Seed Generation" on page 92 for a discussion of the security considerations of random number generation.

Hash-Based Message Authentication Codes (HMAC)

A hash-based message authentication code (HMAC) combines a secret key with a message digest to create a message authentication code. This method of creating a MAC makes it possible to update the underlying message digest if a new attack makes the original message digest unsecure. Crypto-C provides an HMAC implementation based on SHA1.

Recall that SHA1 produces a 20-byte digest; in addition, we need to know that SHA1 takes input in 64-byte blocks.

Given a message *M* and a key *k*, the HMAC of *M* is computed as follows:

1. Create two different fixed strings that are used in the calculation:

   ipad = the byte `0x36` repeated 64 times

   opad = the byte `0x5C` repeated 64 times

2. Extend *k* to 64 bytes in length by appending zeros to the end of *k*. For example, if *k* is 25 bytes, append 39 copies of the zero byte `0x00`. We will call the extended key *k*′.

3. Compute the following:

   $\mathrm{SHA1}(k' \text{ XOR } opad \;\|\; (\mathrm{SHA1}(k' \text{ XOR } ipad) \;\|\; M))$

   where $\|$ denotes concatenation.

The same key can be used for multiple authentications, but the key should be replaced periodically. For security considerations, the key should be at least as long as the message digest output. For SHA1, this means an HMAC key should be at least 20 bytes. If the key is “weakly random”, that is, if knowing some of the key bits might help an attacker generate other key bits, then a longer key should be used.

Password-Based Encryption

Password-Based Encryption (PBE) generates a symmetric key from a password, and encrypts data using that generated key. Usually, though, a password will not have enough effective random bits to qualify as a candidate for a key or even a random seed to generate a key. For example, each character of an 8-byte alphanumeric password that also allows case-sensitive letters has the equivalent of slightly less than six bits of randomness. For eight-character passwords, this is far less than the required key size of a block cipher such as DES.

Therefore, a good PBE implementation not only uses the password, but mixes in a random number, known as a *salt*, to create the key. Normally, the mixing is a message digest. This makes the task of getting from password to key very time-consuming for an attacker. Digesting a password with a salt helps thwart *dictionary attacks*. An attacker could put together a “dictionary” of keys generated from likely passwords, and try out each key on encrypted data. This would greatly reduce the amount of work necessary to find the key and may make it feasible to recover encrypted material. With a salt, the attacker would have to create a dictionary of keys generated from each password, but each password would then have to have a dictionary of each possible salt.

Crypto-C uses the methods described in the PKCS document #5 to implement password-based encryption. The methods use a message digest algorithm with a specific means of padding to increase the search space for dictionary attacks against the key.

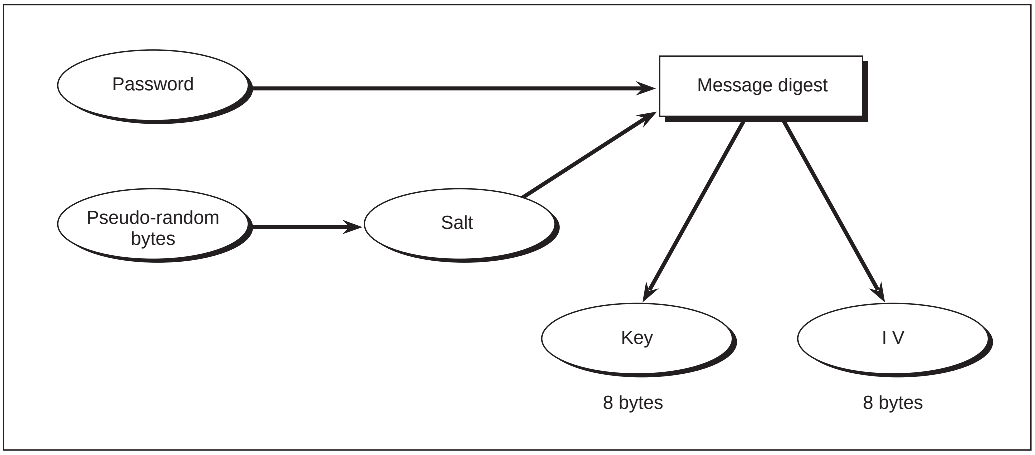

Figure 2-8 **DES Key and IV Generation for Password Based Encryption**

Public-Key Cryptography

In 1976, Stanford graduate student Whitfield Diffie and Stanford professor Martin Hellman invented *public-key cryptography*. In this system, each person owns a pair of keys, called the public key and the private key. The key pair's owner publishes the public key and keeps the private key secret.

Suppose Alice wants to send a message to Bob. She finds his public key and encrypts her message using that public key. Unlike symmetric-key cryptography, the key used for encryption will not decrypt the message. That is, knowledge of Bob's public key will not help an eavesdropper. To decrypt a message, Bob uses his private key. If Bob wants to respond to Alice, he can encrypt his message using her public key.

To get a flavor of this idea, think of taking a number to a power. For instance, given values x and y, compute $z = x^y$. To recover x, you would not compute z^y, but rather $z^{1/y}$. You end up with the original x, because $z^{1/y} = (x^y)^{1/y} = x^{y \cdot 1/y} = x^1 = x$. You need two values to perform this exercise, a "public key," y, to compute the encrypted value, and the inverse of the public key, or a "private key," $1/y$, to recover the original value.

This example, of course, is not practical because if you made y public, anyone could easily compute $1/y$ and know your private key. Therefore, a good public-key cryptosystem relies on a key pair for which it is impossible (or at least intractable) to derive the private key from the public key.

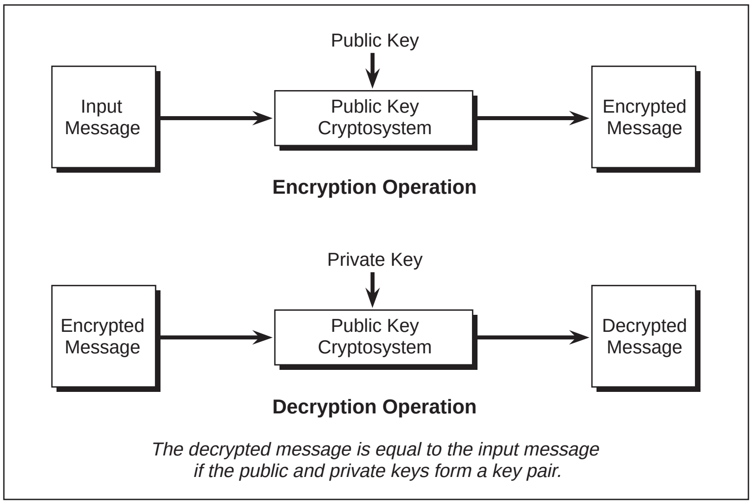

Figure 2-9 **Public-Key Cryptography**

In practice, public-key algorithms are slow compared to symmetric-key algorithms. Therefore, they are more often used for shorter messages, such as encrypting the symmetric key for a message encrypted with a symmetric cipher, or for encrypting a digest.

The RSA Algorithm

RSA is a public-key cryptosystem for both encryption and authentication that MIT professors Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman invented in 1977. It is actually similar to the example in the previous section that takes numbers to a power, except that it works in *modular math*.

Modular Math

Modular math uses a positive integer as a *modulus*; the only numbers under consideration are the integers from 0 to one less than the modulus. So for mod n, only the integers from 0 to $(n–1)$ are valid operands and the results of operations will always be numbers from 0 to $(n–1)$. When an operation such as addition or multiplication would give a result that is greater than the modulus, the remainder of the result after division by n is used instead. Therefore, two numbers are equal mod n if and only if their difference is an even multiple of n.

For example, think of military time where the modulus is 2400. For instance, 2200 hours (10:00 P.M.) plus 4 hours is not 2600, but 0200 hours, or 2:00 in the morning. Likewise, if we start at 0, or midnight, 6 times 5 hours (say six 5-hour shifts) is not 3000, but 0600, or 6:00 A.M. the following day.

Another aspect of modular math is the concept of an *inverse*. Two numbers are the inverse of each other if their product equals 1. For instance, $7 \cdot 343 = 2401$, but if our modulus is 2400, the result is $(7 \cdot 343) \bmod 2400 \equiv 2401 - 2400 = 1 \bmod 2400$.

Prime Numbers

The RSA algorithm also employs *prime numbers*, or primes. A prime number is a number that is evenly divisible by only 1 and itself. For instance 10 is not prime because it is evenly divisible by 1, 2, 5, and 10. But 11 is prime, because its only factors are 1 and 11.

The RSA Algorithm

The RSA algorithm works as follows: take two large primes, p and q, and find their product $n = pq$; n will be the modulus. Choose a public value, e (also known as the *public exponent*), that is less than n. There are other constraints on e described later. To compute ciphertext c from a plaintext message m, find

$$c = m^e \bmod n$$

To decrypt, determine the private key d, the inverse of e, and compute

$$m = c^d \bmod n$$

The relationship between e and d insures that the algorithm correctly recovers the original message m, because

$$c^d = (m^e)^d = m^{ed} \equiv m^1 = m \bmod n$$

Only the entity that knows d can decrypt.

The security of the system relies on the fact that if you know p, q and e, it is easy to compute d, but if you know only n and e it is more difficult to determine d. This is due to the following property of the math: the value d is actually not the inverse of e mod n, but rather the inverse of e mod $(p–1)(q–1)$. The value you pick for e must be relatively prime to $(p–1)(q–1)$, which means e and $(p–1)(q–1)$ share no common factors, so that there exists d such that

$$ed \equiv 1 \bmod (p-1)(q-1)$$

In other words, you find the private value using a modulus of $(p–1)(q–1)$, but when you apply the RSA algorithm to encryption or decryption, you use a modulus of $n = p \cdot q$.

Why, if d is the inverse of e mod $(p–1)(q–1)$, does $c^d = (m^e)^d = m^{ed} = m^1 = m \bmod n$? Aren't we mixing moduluses? That is the quirk of the math; it may seem counterintuitive, but that "mixing of moduluses" is what makes the algorithm work. A complete proof of this fact is beyond the scope of this publication, so if you want to learn more about the underlying mathematical principle, find a math book that discusses Euler's phi-function.

Incidentally, in practice you would generally pick e, the public exponent first, then find the primes p and q which satisfy the requirement that e be relatively prime to $(p–1)(q–1)$.

Consider the following example with small numbers. Choose public exponent $e = 3$. Then, let $p = 5$ and $q = 11$, which means $n = 55$ and $(p–1)(q–1) = 40$. This is a valid p and q combination because 3 is relatively prime to 40. The inverse of 3 mod 40 is 27.

$$(3 \cdot 27) = 81$$
$$81 - (2 \cdot 40) = 81 - 80 = 1$$
$$3 \cdot 27 = 1 \bmod 40$$

Apply the RSA algorithm with these parameters to the "plaintext message" $m = 2$.

$$c = m^e = 2^3 = 8 \bmod 55$$

This yields an encrypted message of 8.

To decrypt, raise the message to the power of the inverse of 3, which is 27.

$$c^d = 8^{27} \bmod 55$$

Rather than computing 8^{27} directly, a shortcut would be to compute:

$$8^{16+8+2+1} = 8^{16} \cdot 8^8 \cdot 8^2 \cdot 8^1 = 2 \bmod 55$$

The calculation is shown in Table 2-1:

Table 2-1 **Calculation of 8^{27} mod 55**

8^0			1 mod 55
8^1			8 mod 55
8^2	$8^1 \cdot 8^1 = 8 \cdot 8 = 64$	$64 – 55 = 9$	9 mod 55
8^4	$8^2 \cdot 8^2 = 9 \cdot 9 = 81$	$81 – 55 = 26$	26 mod 55
8^8	$8^4 \cdot 8^4 = 26 \cdot 26 = 676$	$676 – (12 \cdot 55) = 16$	16 mod 55
8^{16}	$8^8 \cdot 8^8 = 16 \cdot 16 = 256$	$256 – (4 \cdot 55) = 36$	36 mod 55
$8^1 \cdot 8^2$	$8 \cdot 9 = 72$	$72 – 55 = 17$	
$(8^1 \cdot 8^2) \cdot 8^8$	$17 \cdot 16 = 272$	$272 – (4 \cdot 55) = 52$	52 mod 55
$(8^1 \cdot 8^2 \cdot 8^8) \cdot 8^{16}$	$52 \cdot 36 = 1872$	$1872 – (34 \cdot 55) = 2$	2 mod 55

Summary

Take two large primes, p and q, and find their product $n = p \cdot q$. Set n to be the modulus. Choose a public exponent, e, less than n and relatively prime to $(p–1)(q–1)$. Find d, the inverse of e mod $(p–1)(q–1)$, that is, $ed \equiv 1$ mod $(p–1)(q–1)$. The pair (n,e) is the public key; d is the private key (or the private exponent). The primes p and q must be kept secret or destroyed.

To compute ciphertext c from a plaintext message m, find $c = m^e$ mod n. To recover the original message, compute $m = c^d$ mod n. Only the entity that knows d can decrypt.

Note: In public-key cryptography, it is also possible to encrypt using a private key. In this case, the sender takes the plaintext input and the private key and follows the same steps need to decrypt an encrypted file. This creates a ciphertext that can be read using the public key; to read it, the recipient follows the same steps needed to encrypt with the public key and restores it to the plaintext. This is used in authentication and digital signatures.

Security

The security of the RSA algorithm relies on the difficulty of factoring large numbers. In theory, it is possible to obtain the private key d from the public key (n,e), by factoring n into p and q. In order to find d, one must know the product $(p–1)(q–1)$. But to find that value, one must know p and q. For example, in the earlier example, an eavesdropper would know that $p \cdot q = 55$, but what is $(p–1)(q–1)$? Factoring 55 into its component primes is easy: the answer is 5 and 11.

However, for very large numbers, factoring is very difficult. The RSA Laboratories publication, *Frequently Asked Questions About Today's Cryptography* (the *FAQ*), describes the state of the art in factoring. Factoring numbers takes a certain number of steps, and the number of steps increases exponentially as the size of the number increases. Even on supercomputers, the time to execute all the steps is so great that for large numbers it could take years to compute. Within a short period of time, the current threshold of general numbers that can be factored will probably rise to 155 digits, approximately the size of a 512-bit RSA modulus. Currently, the limit to the size of an RSA modulus in Crypto-C is 2048 bits.

Digital Envelopes

A *digital envelope* is a way of combining the advantages of symmetric- and public-key cryptography. In general, public-key algorithms are slower than symmetric-key ciphers, and for some applications may be too slow to be of practical use, while for symmetric-key ciphers, there is the problem of transmitting the key. A digital envelope provides a solution to this dilemma. The sender encrypts the message using a symmetric-key encryption algorithm, then encrypts the symmetric key using the recipient's public key. The recipient then decrypts the symmetric key using the appropriate private key and decrypts the message with the symmetric key. In this way, a fast encryption method processes large amounts of data, yet secret information is never transmitted unencrypted.

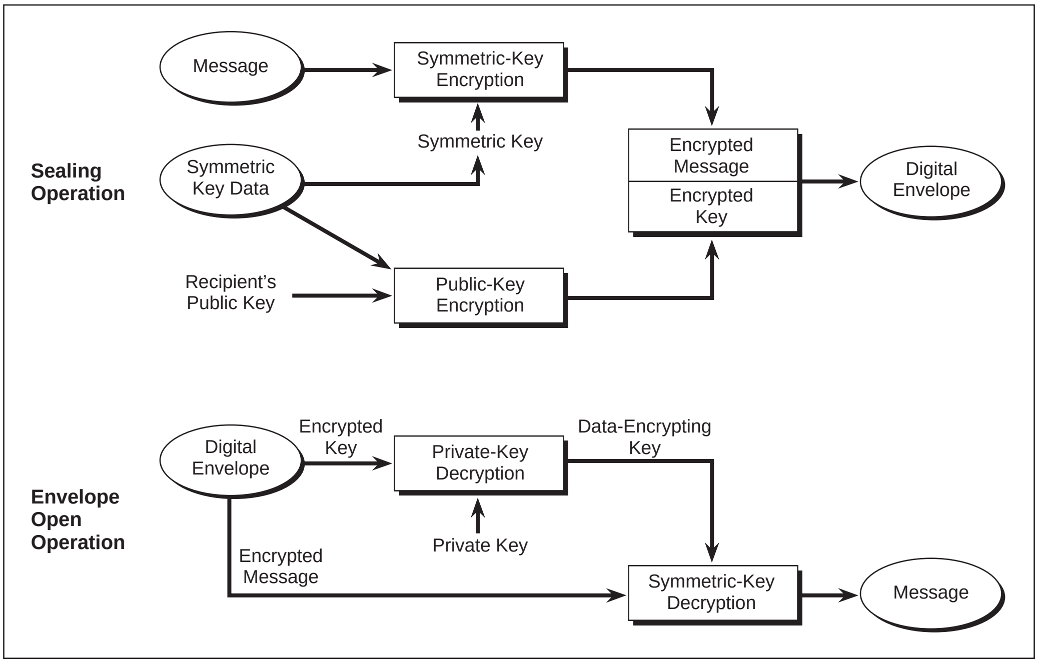

Figure 2-10 **Digital Envelope**

Authentication and Digital Signatures

Suppose Alice and Bob are disputing a contract. Alice says that Bob must uphold certain obligations because he agreed to them in a contract. Bob says that this is not the contract he signed. He offers as evidence his copy of the contract and sure enough, it differs from Alice's. One of them has altered their copy of the contract, but who? Or maybe the dispute centers on Bob's assertion that he never signed a contract, that the signature at the bottom is not his. In that case, either Bob is not telling the truth or Alice forged his signature.

If the contract was signed physically, there are ways to determine the truth. Contracts are often filed with government agencies, so comparing Bob's and Alice's copies with the third party's copy reveals who made alterations. Witnesses may also sign the contract and later testify that both parties did sign it, and the signatures are not forgeries. For electronic documents, there is also a method to determine if a document has been altered or if someone truly did sign it. This method is the *digital signature*.

There are two types of signature algorithms. The first is a public-key cryptosystem that can perform block encryption, while the second is only capable of digital signatures. The RSA algorithm is an example of the first type. The Digital Signature Algorithm, DSA, is an example of an algorithm of the second type. Crypto-C includes the RSA and DSA signature methods.

A digital signature uses a public/private key pair to sign a document. First the signer digests the document, as described in “Message Digests” on page 46, then encrypts it with their private key. A good digital signature algorithm possesses the following properties:

- Only the owner of a private/public key pair can generate a signature. Knowledge of the public key does not enable anyone else to forge a signature.
- Knowledge of the public key enables anyone to verify the signature.
- The digital signature guarantees the authenticity of the message and its author.

  The digital signature is computationally unique for each message and signer. While a normal signature can be imitated, a digital signature is immune to imitation.
- Any altering of the message renders the signature invalid.

Note: If a digital signature is invalid, you cannot be sure it was a deliberate forgery. Transmission errors will also produce errors in a digital signature.

For example, to create a digital signature on a contract:

1. Alice and Bob compose a contract in digital format. The file can be in any form, such as a word processing file or an ASCII file.
2. Each party digests the file and encrypts the digest with their private key.
3. That encrypted digest is their digital signature.
4. The contract now consists of the file and the two copies of the encrypted digest, one using Alice’s private key, the other using Bob’s private key. Everyone gets copies of this contract.

The digital signature can be used to verify the data at a later time. Suppose that Bob produces a file that is different from Alice’s. To discover which copy has been altered:

1. Digest the new copy.
2. Decrypt each party’s encrypted digest with the corresponding public key.
3. Compare the new digest to the old one.
4. If one of the new digests does not match the old one, that is the altered file.

If a file has been altered, it will produce a different digest, because it is virtually impossible to produce data that will digest to a given value. Even if someone could manipulate the digest, it would be extremely difficult to produce data that has value to anyone.

The digital signature can also be used to verify that a message came from a given person. What if Bob claims Alice forged his digital signature on the original document? He might say her copy of his encrypted digest is not the true version. That is very unlikely. To do that, Alice would have had to have encrypted the digest of her choice with Bob's private key, to which she has no access.

The following example shows how to verify a message and its signature. Suppose you have the following information:

- a message
- an entity who claims to have sent the message
- a block of data 96 bytes long that purports to be the encrypted digest

To verify the message and the sender:

1. Request the possible sender's 768-bit (96-byte) RSA public key from a certification authority.
2. Use that public key to decrypt the 96-byte block of data.
3. If the decryption process results in a 16-byte output, you can say it is a message digest. There is a message that will digest to those 16 bytes, but you do not yet know what it is.
4. Digest the message file.
5. If the digest matches the 16 bytes you obtained from decrypting the original 96-byte block, the message is verified. That is, you can assume the 96-byte block is the file's digest encrypted with the RSA private key associated with the public key you used. It would have been computationally infeasible to produce that 96-byte block any other way.

There are other uses for a digital signature. Suppose that Bob wishes to buy something from Alice over the Internet. He emails her a credit card number. Alice can easily find out from the credit card issuer that the number she received is valid and indeed belongs to Bob. But how does she know that it was Bob who sent the number and not someone posing as Bob? She sends the purchaser a randomly generated message and asks him to digitally sign it with his private key. She then retrieves his public key from a certification authority and verifies the signature. Only the person with access to Bob's private key will be able to generate a digital signature from the message she generated in such a way that Bob's public key will verify it properly. In this way, Alice authenticates Bob's identity.

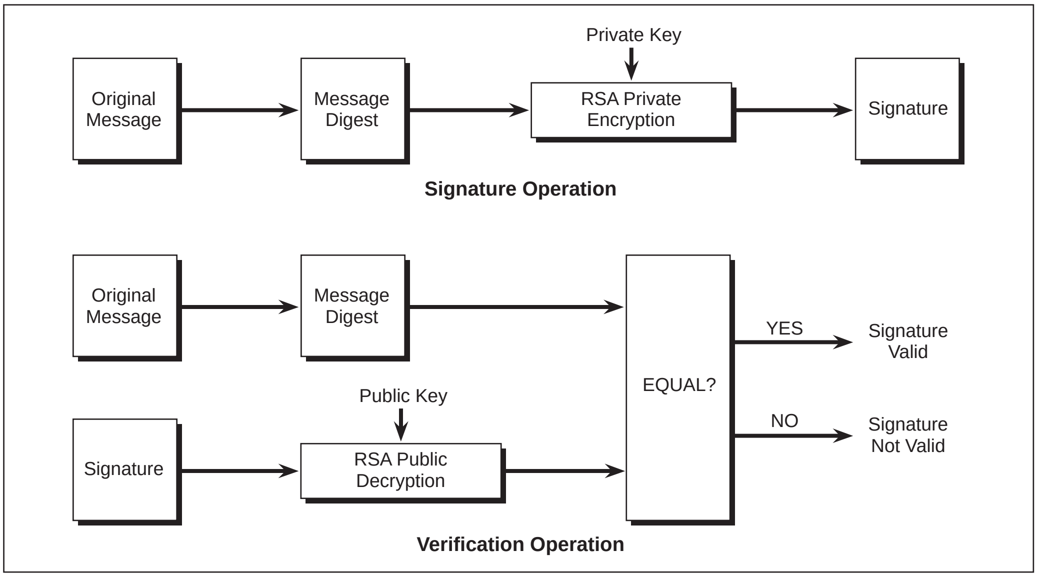

Figure 2-11 **RSA Digital Signature**

Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA) is part of the Digital Signature Standard (DSS), published by the National Institute of Standards and Technology (NIST), a division of the US Department of Commerce. It is the digital authentication standard of the US government. The DSS specifies the Secure Hash Algorithm (SHA1) as the message digest to use with DSA when generating a digital signature.

To generate a DSA key pair:

1. Find a prime, p, at least 512 bits long.
2. Find a second prime, q, exactly 160 bits long, that satisfies the property $q \mid (p–1)$. q is called the *subprime*.
3. Generate a random value, h, the same length as p but less than p.
4. Compute $g = h^{(p-1)/q} \bmod p$. g is called the *base*.
5. Generate another random value, x, 160 bits long. x is the private value.
6. Compute the public value: $y \equiv g^x \bmod p$.

Note: The three values p, q, and g above (the prime, subprime, and base, respectively) are called the DSA *parameters*. The parameters are public and must be generated before you can sign a message.

To sign a message using DSA:

1. Digest the message using SHA1. This yields a 20-byte (160-bit) digest.
2. Generate a random value, k, 160 bits long and less than q.
3. Find the following values:

   $k_{inv} = k^{-1} \bmod q$

   $r = (g^k \bmod p) \bmod q$
   $xr = (x \cdot r) \bmod q$
   $s = [k_{inv} \cdot (\text{digest} + xr)] \bmod q$
4. Output the signature (r,s).

To verify a message:

1. Digest the message using SHA1.
2. From the signature (r,s), compute:

   $s_{inv} = s^{-1} \bmod q$
   $u_1 = (\text{digest} \cdot s_{inv}) \bmod q$
   $u_2 = (r \cdot s_{inv}) \bmod q$

   $a = g^{u_1} \bmod p$

   $b = y^{u_2} \bmod p$
   $v = (a \cdot b \bmod p) \bmod q$
3. If $v = r$, the signature is verified. If $v \neq r$, the signature is invalid.

The Math

To see that this is indeed the signature, consider the following. We have the values:

$$y = g^x \bmod p$$

and

$$u_2 = r \cdot s_{inv} \bmod q$$

Make the following algebraic substitutions:

$$a \cdot b \bmod p = g^{u_1} \cdot g^{x \cdot u_2} \bmod p$$
$$= g^{u_1 + x \cdot u_2} \bmod p$$

$$= g^{digest \cdot s_{inv} + x \cdot r \cdot s_{inv}} \bmod p$$
$$= g^{s_{inv}(digest + x \cdot r)} \bmod p$$
$$= g^{k} \bmod p$$

Recall that:

$$r = (g^{k} \bmod p) \bmod q$$

This means that:

$$v = (a \cdot b \bmod p) \bmod q$$
$$= (g^{k} \bmod p) \bmod q$$
$$= r$$

Digital Certificates

Suppose you own an RSA public/private key pair. You must make your public key public, so that others can use it to verify your digital signature or to encrypt session keys when creating an RSA envelope. How do you publicize your key?

Probably the best way is to register public keys with a trusted authority. Then, this trusted authority can certify that a particular public key belongs to a particular entity. Currently, such a public key registration infrastructure exists in the form of *digital certificates*.

A certificate connects an entity to a public key. For instance, it can list an individual's name, address, and public key. When people want to use a person's public key, they look up the certificate associated with that person's name and address. A certificate can contain a wide variety of information on its owner, such as the person's organization or job title. This helps differentiate between people who have the same name. The certificate can also contain information on when it was issued or when the public key expires.

For a certificate system to work, there need to be individuals or organizations that issue and maintain the certificates. These are known as a *certificate authorities*, or CAs-. An individual can request a certificate by presenting a CA with a public key and a name and any other identifying information. It is then the CA's responsibility to verify that the entity making the request is indeed the person identified by the information or is authorized to be associated with that key. The level of trust users place in a CA will depend on the level of verification it performs.

When you ask for an individual's public key, the CA sends the certificate and signs it with the digest of the certificate encrypted with the CA's private key. To verify that the certificate is genuine, you must digest the certificate and decrypt the signature using the CA’s public key. Compare the two results: if they are the same, you have a proper certificate.

If the CA you deal with does not have a certificate for the individual in question, that CA can communicate with another CA that might have the right certificate. In fact, to find a particular certificate, a CA may have to go through a chain of CAs until it finds one that possesses the desired certificate.

Names that uniquely distinguish users are necessary for digital certificates to be of real use. The CCITT X.500 series of documents offer more discussion regarding naming conventions and related topics.

Diffie-Hellman Public Key Agreement

The Diffie-Hellman Public Key Agreement, invented by Whitfield Diffie and Martin Hellman in 1976, was the first true public-key algorithm. It provides a method for *key agreement*; that is, it allows two parties to each compute the same secret key without exchanging secret information. Diffie-Hellman key agreement does not provide encryption or authentication.

The Algorithm

The Diffie-Hellman algorithm is made up of three parts (see Figure 2-12 on page 62):

- Parameter Generation
- Phase 1
- Phase 2

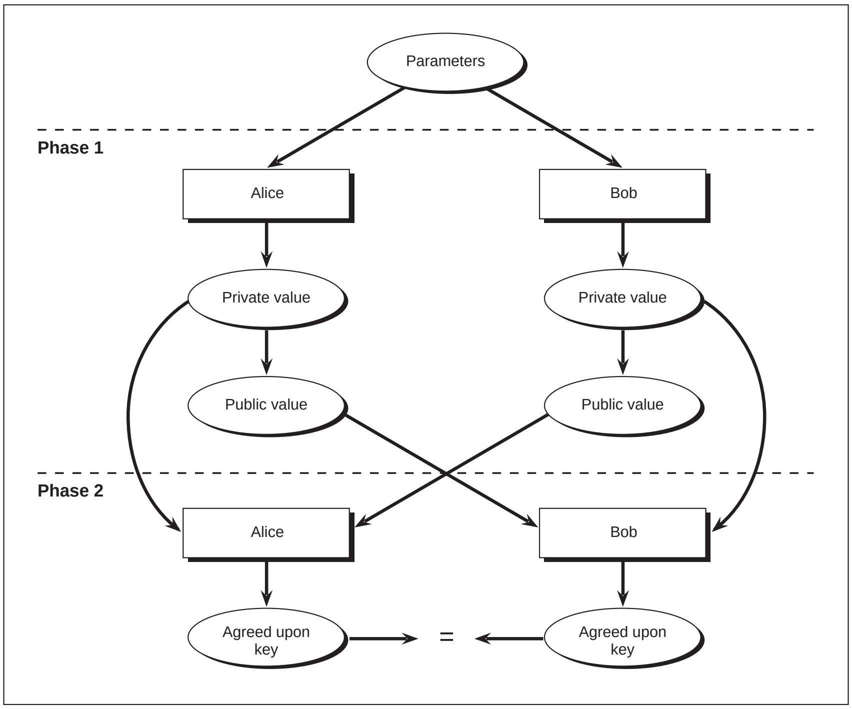

Figure 2-12 **The Diffie-Hellman Key Agreement Protocol**

Parameter Generation

A central authority selects a prime number p of length k bytes, and an integer g greater than 0 but less than p, called the *base*. The central authority may optionally select an integer l, the private-value length in bits, that satisfies $2^{l-1} \leq p$.

Phase 1

Each of the two parties executing the Diffie-Hellman protocol does the following:

1. Each party, i, $i = 1$ or 2, randomly generates a private value, which is a number, x_i, greater than 0 but less than the prime. If the central authority has specified the length l, the private value shall satisfy $2^{l-1} \leq x_i < 2^l$.
2. Each party computes a public value $y_i = g^{x_i} \bmod p$.
3. The two parties exchange the public values.

These private and public values correspond to the private and public key components of a key pair. The public value is generated in such a way that computing the private value from the public number is computationally infeasible.

Phase 2

Each participant computes the agreed-upon secret key, z, using: the other's public value, y', their own private value, x, and the prime, p, as follows:

$$z = (y')^x \bmod p.$$

Even with knowledge of the parameters and both public keys, an outside individual will not be able to determine the secret key. You must have one of the private values to determine the secret key. This means secret information is never sent over unsecure lines.

The Math

Even though the two parties involved are making computations using different private values, they will both end up with the same secret key, as illustrated by the following.

p: prime
g: base
x_1: 1st party's private value
x_2: 2nd party's private value
y_1: 1st party's public value
y_2: 2nd party's public value
z: secret key

In Phase 1, each party computes a private value, x_n, and a public value, y_n:

$$y_1 = g^{x_1} \bmod p$$

$$y_2 = g^{x_2} \bmod p$$

In Phase 2, the parties trade public values and compute the same secret key:

$$z = {y_2}^{x_1} \bmod p$$

$$z = {y_1}^{x_2} \bmod p$$

They both compute the same z, because:

$$y_2^{x_1} = (g^{x_2})^{x_1} = (g^{x_1})^{x_2} = y_1^{x_2} \bmod p$$

Security

The security of Diffie-Hellman key agreement relies on the difficulty of computing *n*th roots modulo a prime number. It takes very little time to exponentiate a number modulo a prime, but it takes a great deal of time to compute its roots. This problem in modular arithmetic is called the *discrete logarithm problem*. (Recall that, in the real numbers, if you can compute the logarithm of a number, you can easily compute all of its roots.) The RSA Laboratories publication, *Frequently Asked Questions About Today's Cryptography*, states, "The best discrete log problems have expected running times similar to that of the best factoring algorithms." That is, the time it takes to compute discrete logs modulo a prime of a certain length is approximately equivalent to the time it takes to factor a number of that same length. See "The RSA Algorithm" on page 50 for a discussion of factoring.

Multiple-Party Key Agreement

The above protocol can be extended to more than two parties. For a multiple-party agreement, each individual chooses a private value, then uses the collection of public values from other parties to generate a common secret key.

Elliptic Curve Cryptography

Elliptic curves are mathematical constructs that have been studied by mathematicians for over 100 years. The application of elliptic curves to cryptosystems is more recent; in 1985, Neal Koblitz and Victor Miller independently devised a public-key system using a group of points on an elliptic curve.

The core of elliptic curve cryptosystems rests on the difficulty of a particular type of calculation. For some public-key algorithms, such as Diffie-Hellman key agreement, the security is based in part on the fact that given a modulus n, a number g, and g^k mod n, it is difficult to determine k. This is called the discrete logarithm problem. Elliptic curve cryptosystems rest on a similar problem: given an elliptic curve E and two points on the curve, $\boldsymbol{P}$ and $\boldsymbol{Q}$, such that $\boldsymbol{Q} = k \cdot \boldsymbol{P}$ for some number k, it is difficult to determine k. This is called the *elliptic curve discrete logarithm problem*. (See the next section, "Elliptic Curve Parameters", for a discussion of these terms.) Many algorithms that are based on the discrete logarithm problem can be translated to analogous algorithms based on the elliptic curve discrete log problem.

Elliptic curves can be used for a variety of public-key cryptosystems. Crypto-C supports the following elliptic curve features:

- Generation of elliptic curve parameters
- Elliptic curve key pair generation
- Elliptic Curve Signature Schemes (ECDSA)
- Elliptic Curve Authenticated Encryption Scheme (ECAES)
- Elliptic Curve Diffie-Hellman key agreement (ECDH)

Crypto-C also allows you to generate precomputed acceleration tables to speed up certain elliptic curve operations. For more information, see the example "Public-Key Acceleration Table" on page 247.

Elliptic Curve Parameters

A number of parameters are necessary for elliptic curve cryptosystems. These parameters must be generated before you generate a key pair, create an acceleration table, initiate encryption, or perform key agreement with these systems. You can use the same parameters to generate more than one key. These parameters include:

- the finite field, F_q, over which the elliptic curve is defined
- two elements of F_q, a and b, which define the elliptic curve; a and b are also called the coefficients of the curve
- a point $\boldsymbol{P}$ of prime order on the elliptic curve E
- the order, n, of $\boldsymbol{P}$
- the cofactor $h = \#E(F_q)/n$. Here, $E(F_q)$ means the set of points on the elliptic curve and $\#E(F_q)$ means the number of points in that set. See "The Order of an Elliptic Curve" on page 69 for more information

Note: In all discussions of elliptic curves, the upper case letters $\boldsymbol{P}$ and $\boldsymbol{Q}$ are used to denote points on an elliptic curve. The lower case letter p is used to denote a prime.

The next section discusses these terms in detail. We will try to give enough of the math to give you a feel for what the underlying concepts are without going too deeply into the details. A full discussion of elliptic curve cryptography is far beyond the scope of this manual. For background on elliptic curves, see the book by J. Silverman and J. Tate, *Rational Points on Elliptic Curves* [20]. For more information on elliptic curves in cryptography, see the ANSI X9.62 and X9.63 Draft standards [13], the IEEE *Standard Specifications for Public-Key Cryptography* [14], and A. Menezes book, *Elliptic Curve Public Key Cryptosystems* [19].

The Finite Field

The elliptic curves used in cryptography are always defined over a *finite field*, denoted F_q. There are two choices for this field:

- An odd prime field, F_p, where p is an odd prime.
- A field of even characteristic, F_{2^m}.

For more information about finite fields, see the book by A. Menezes, I. Blake, X. Gao, R. Mullin, S. Vanstone, and T. Yaghoobian, *Applications of Finite Fields* [18] and also Chapter 2 of Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone' s book, *Handbook of Applied Cryptography* [17].

Odd Prime Fields

The *odd prime field* F_p is simply Z_p, the integers mod p. Modular math is described in the section "The RSA Algorithm" on page 50. Recall that in modular math, we have addition and multiplication, with the additional twist that the numbers loop around, so that, for example, $p+1 = 1$ mod p.

Although you don't need it to use the cryptosystem, a little background may help. Because p is prime, F_p has an interesting property that not all modular math systems have: except for 0, every number in F_p has a multiplicative inverse. That is, given any number c between 1 and $p–1$, there is another number d in the same range such that $cd = 1$ mod p. This is the crucial property that distinguishes F_p from other modular math systems and makes it a field.

Not all moduli will give you a field. For instance, our earlier example, arithmetic mod 55, is not a field. You can see this by looking at the number 5 in this system. The first ten multiples of 5 are: 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50. When we multiply 5 by 11, we get 55, which is just 0 mod 55. Now, when we multiply 5 by 12, we just fall back down to 60 = 60–55 = 5 mod 55. In fact, no matter what we multiply 5 by, we will just get a multiple of 5, which will reduce back down to the ten numbers listed above. There is no way we can get to 1 as a multiple of 5 in this particular modular system.

In fact, the only numbers that will give a field in modular arithmetic are the primes. So you can see that fields are fairly special. The crucial thing to remember is:

An odd prime field, F_p, is just modular arithmetic, where the modulus p is prime.

Fields of Even Characteristic

The fields of *even characteristic*, also known as characteristic 2, are more complicated. If you were looking for a field of that size, you might start with the integers mod 2^m. However, it turns out that integers mod 2^m cannot be a field for any m>1.

Why is this? Remember, we said every element in a field, except 0, has a multiplicative inverse. But, for example, 2^{m-1} cannot be invertible in the integers mod 2^m (except for $m = 1$). To see this, consider the product $2 \cdot 2^{m-1} = 2^m \equiv 0 \bmod 2^m$. If 2^{m-1} did have an inverse, I, then we would have:

$$\begin{aligned}
&0 = 0 \cdot I\\
&\equiv (2 \cdot 2^{m-1}) \cdot I \bmod 2^m\\
&= 2 \cdot (2^{m-1} \cdot I)\\
&\equiv 2 \cdot 1 \bmod 2^m\\
&= 2
\end{aligned}$$

Instead, we create the field F_{2^m} in a completely abstract manner. We start by letting the elements of the finite field F_{2^m} be the bit strings of bit-length m. Mathematicians have shown that it is possible to create an addition and a multiplication that make these strings, called m-tuples, into a field.

Addition is easy to define: to add two strings, just XOR them. This is the same as adding them bit by bit, with no carry. Notice that with this field addition rule, for every x in F_{2^m}, we have that $x + x = 0$. That is already very different from addition in the integers mod 2^m.

Note: If you look closely, you will see that we are trying to create a system where 2 can equal 0. In fact, it is because of this property — that the number 1 added to itself two times gives us 0 — that we say this is a field of "characteristic 2" or "even characteristic". The amazing thing is that not only can this can be done, but that we can get something useful out of it.

Multiplication is even more difficult to define. When you multiply two m-tuples, you can't just multiply them bit-by-bit, or else you would never be able to invert any string that had a 0 in it somewhere. Instead, multiplication in F_{2^m} is a complicated operation involving ordinary multiplication and addition of cross terms.

The mathematics underlying the construction of F_{2^m} is deep, but it is very well-understood by mathematicians. For an in-depth discussion of this field, see [18] and [17].

Elliptic Curve Coefficients

An elliptic curve, E, can be thought of as a particular type of equation. Elliptic curves look slightly different in the two different cases.

Coefficients Over an Odd Prime Field

An elliptic curve E over an odd prime field F_p is all the pairs of points (x,y) that satisfy the equation:

$$y^2 = x^3 + ax + b$$

In this equation, x and y are elements of F_p, and so are a and b. The whole equation is evaluated over F_p. For computational reasons, there is also a "point at infinity", O, that is included as well.

The numbers a and b are called the *coefficients* of the elliptic curve; they are part of the elliptic curve parameters.

Coefficients Over a Field of Even Characteristic

An elliptic curve E over a field of even characteristic F_{2^m} is all the pairs of points (x,y) that satisfy the equation:

$$y^2 + xy = x^3 + ax^2 + b$$

In this equation, x and y are elements of F_{2^m}, and so are a and b. The whole equation is evaluated over F_{2^m}. For computational reasons, there is also a "point at infinity", O, that is included as well.

The numbers a and b are called the *coefficients* of the elliptic curve; they are part of the elliptic curve parameters.

Note: Note that the equation over F_{2^m} is different from the equation over F_p. Over F_{2^m} there is a quadratic term, ax^2, instead of the linear term ax in the odd prime case, as well as a new cross-term, xy. The differences in the equation arise because of the differences in arithmetic between the two types of fields.

The Point *P* and its Order

Obviously, you can't create a cryptosystem out of just any equation. The elliptic curve equation is important because it has special properties. One of these properties is that it is possible to set up an addition system that lets you add one point on the elliptic curve to another. The addition is complex and non-obvious, but it is possible to set up a system of equations that determine the sum of two points. Adding two points on an elliptic curve involves several operations in the underlying field, F_q, including multiplications, additions, and the computation of inverses. The complexity of the addition is what makes elliptic curve cryptosystems work — if you add a point $\boldsymbol{P}$ to itself k times to get $k\boldsymbol{P}$, there is no known fast way to get k.

To implement an elliptic curve cryptosystem, we need to specify a point $\boldsymbol{P}$ on our curve that has some special properties. To understand these properties, we need some more concepts: the points on a curve, the order of a curve, and the order of a point on the curve.

The Points of an Elliptic Curve

For our field, F_q, and our elliptic curve *E*, determined by *a* and *b*, we can consider all the pairs (*x,y*) in F_q that satisfy the elliptic curve equation. Each such pair is called a *point* of the elliptic curve. The collection of all the points that satisfy the equation, along with the special point *O* mentioned earlier, is called the points of *E* over F_q; this is written $E(F_q)$.

The Order of an Elliptic Curve

The addition system that makes the points on the elliptic curve into what is called a group has a number of properties. The first thing to notice is that there can only be a finite number of points on the curve. Even if every possible pair (*x,y*) were on the curve, there would be only p^2 or $(2^m)^2 = 2^{2m}$ possibilities. The total number of points, including the point *O*, is called the *order* of the elliptic curve. The order is written as $\#E(F_q)$.

The special point *O* plays the role of the additive identity, zero, in the group of the elliptic curve.

The Order of a Point

Given any point on the curve, ***P***, the addition rule lets you add that point to itself. Then you can add your new point to the old point, and so on. When you add a point to itself a number of times, it is called *scalar multiplication*. Although this is not multiplication in the usual sense — it is an iteration of point addition *k* times — it still has the usual math properties like commutativity and associativity over addition. Adding a point ***P*** to itself *k* times gives another point denoted *k**P***.

No matter what ***P*** is, there is always some *n* such that $nP = O$. The smallest *n* that works for a given ***P*** is called the *order* of ***P***. Not only does *n* exist, but it is always true that *n* evenly divides the order of the elliptic curve, $\#E(F_q)$.

The order *n* of ***P*** is important because it means that when we use ***P*** as the starting point of our calculations, we can apply the rules of arithmetic modulo *n*. That is, we have the following important fact:

$$r = r' \bmod n \text{ if and only if } r\mathbf{P} = r'\mathbf{P}$$

A Point of Prime Order

Now we have those concepts, we can go on to the next parameter. Given our elliptic curve, E, defined over our finite field, F_q, we want to fix a special point that will be used to mask the private key in a public/private key pair. The properties of $\boldsymbol{P}$ are important to the security of our system. Not just any point will do: we need a point $\boldsymbol{P}$ whose order n is prime; the larger the prime, the more secure the cryptosystem.

Remember, $\boldsymbol{P}$ is of the form $\boldsymbol{P} = (x,y)$ where x and y satisfy the elliptic curve equation. To show that x and y are specific to $\boldsymbol{P}$, we usually write them as $x_{\boldsymbol{P}}$ and $y_{\boldsymbol{P}}$. Therefore, the special point $\boldsymbol{P}$ gives us two parameters:

- A point $\boldsymbol{P} = (x_{\boldsymbol{P}}, y_{\boldsymbol{P}})$ of prime order
- The order n of $\boldsymbol{P}$

$\boldsymbol{P}$ is sometimes called the *base point*.

The Cofactor

We mentioned above that the prime number n that is the order of $\boldsymbol{P}$ must evenly divide the order of the elliptic curve. That is, we know that the number $h = \#E(F_q)/n$ is an integer. We call h the *cofactor*, and set it as our last parameter:

- The cofactor $h = \#E(F_q)/n$

Summary of Elliptic Curve Terminology

Table 2-2 lists the elliptic curve parameters and gives a short description of each parameter. For a brief description, see above; for a detailed discussion, see [13], [14], and [19] in the list of references.

Table 2-2 **Elliptic Curve Parameters**

Notation	Name	Description
F_q	base field	Either: F_p : {0,1,...,p–1} with arithmetic mod p or F_{2^m} : strings of m bits. Addition is bitwise XOR, multiplication exists, but has no quick description
a, b	coefficients of the curve	*a* and *b* are elements of F_q. They determine an equation, which depends on the base field: For F_p: $y^2 = x^3 + ax + b$ For F_{2^m}: $y^2 + xy = x^3 + ax^2 + b$

Table 2-2 **Elliptic Curve Parameters**

Notation	Name	Description
P	point of prime order or base point	(x_P, y_P) The pair x_P, y_P satisfies the curve equation.
n	order of *P*	The smallest nonzero number such that *P* added to itself *n* times is the zero point, *O*, on the curve. *n* is prime.
h	cofactor	The order of the curve divided by the order of *P*: $\#E(F_q)/n$

Representing Fields of Even Characteristic

For fields of even characteristic (fields of the form F_{2^m}), Crypto-C allows you to choose how you want the field to be represented. The representation you choose is internal to Crypto-C and affects how field arithmetic is performed. The choice of representation is also one of the formal elliptic curve parameters that must be transmitted along with the public key. Some representations lead to more efficient implementations in hardware or software.

When we talk about representations of F_{2^m}, we use the term *basis* to reflect the original mathematics underlying the construction of F_{2^m}. From our point of view, it is most important to know that a different basis corresponds to a different representation in Crypto-C. Crypto-C offers two types of representation for fields of even characteristic:

- *Polynomial basis:* this representation closely reflects how the field was originally constructed by mathematicians. Every field of even characteristic has a polynomial basis representation.
- *Optimal normal basis (ONB):* this representation is constructed to optimize certain multiplicative operations. Not all fields have an ONB representation; it can be constructed only for certain values of *m*.

The difference in the choice of basis shows up most clearly in how multiplication is defined. For example, for any polynomial basis representation, the multiplicative identity is represented as (000…01). For any optimal normal basis, the multiplicative identity is (111…11).

Note: Although arithmetic looks different when you choose a different representation, the field is still the same. Just as you can represent “normal” arithmetic using a hexadecimal or a decimal system, you can represent F_{2^m} in more than one way.

Elliptic Curve Key Pair Generation

Elliptic curve parameters can be used to generate a public/private key pair. Elliptic curve parameters can either be common to several key pairs or specific to one key pair. The elliptic curve parameters can be public; the security of the system does not rely on these parameters being secret.

Creating the Key Pair

To compute a public/private key pair:

1. Generate a random value, d, between 1 and $n-1$.
2. Compute the elliptic curve point $d\boldsymbol{P}$, that is, $\boldsymbol{P}$ added to itself d times. Call this point $\boldsymbol{Q}$; it is a pair of field elements $(x_{\boldsymbol{Q}},y_{\boldsymbol{Q}})$.

The key pair is $(\boldsymbol{Q},d)$: $\boldsymbol{Q}$ is the public key, d is the private key. As mentioned above, even if you know $\boldsymbol{P}$ and $\boldsymbol{Q}$, you cannot easily calculate d.

ECDSA Signature Scheme

Once you have generated elliptic curve parameters and created a public/private key pair, you can use this information to create an elliptic curve analogue of the Digital Signature Algorithm (DSA).

Signing a Message

The holder of the private key can sign a message as follows:

1. Digest the outgoing message using SHA1. This yields a 20-byte (160-bit) digest, e.
2. Compute a random value, k, between 1 and $n-1$.
3. Compute the elliptic curve point $k\boldsymbol{P} = (x_1,y_1)$.
4. Currently, the first coordinate, x_1, is an element of the finite field. To perform further calculations, we must convert x_1 to an integer, called $\overline{x_1}$. We do this as follows:

For F_p, x_1 is an integer α in the range *0* to *p–1*. Let $\overline{x_1} = \alpha$. (Essentially, no conversion is required.)

For F_{2^m}, x_1 is a bit string of length *m* bits: $s_1 s_2 \ldots s_m$. Because F_{2^m} has a very strange arithmetic, we need a way to think of its elements as integers. To do this, let the integer $\overline{x_1}$ be a weighted sum of the bits of x_1:

$$\overline{x_1} = \sum_{i=1}^{m} 2^{(m-i)} \cdot s_i$$

In either case, once you have calculated $\overline{x_1}$, set $r = \overline{x_1}$.

Note: Although this lets you take a member of the field F_{2^m} and represent it as an integer, it has some limitations. If you perform any arithmetic operations on $\overline{x_1}$, you will be using regular arithmetic. This is so different from arithmetic in F_{2^m} that, for example, $\overline{x_1 + x_2} \neq \overline{x_1} + \overline{x_2}$. However, if you convert two field elements and perform operations on them that show they are equal after conversion, then they were equal before conversion.

5. Compute $s = k^{-1}(e+dr) \bmod n$. Again, you must check that *s* is nonzero.

The signature for this message is the pair *r* and *s*. Notice that, as with DSA, the signature depends on both the message and the private key. This means no one can substitute a different message for the same signature.

Note: The above equation is merely an outline. For cryptographic purposes, it is necessary to verify that certain numbers are nonzero, or that they satisfy other conditions. Crypto-C makes the appropriate verifications when it generates your key pair.

Verifying a Signature

When a message is received, the recipient can verify the signature using the received signature values and the signer's public key, ***Q***. Because the pair (*r*,*s*) that has been received may not actually be a valid signature pair, it is customary to call the received pair (*r′*,*s′*) instead.

To verify a signature:

1. First verify that r' and s' are between 1 and $n-1$. If they are not, the output is invalid.
2. Digest the received message using SHA1. This yields a 20-byte (160-bit) digest, e.
3. Compute $c = (s')^{-1}$. Remember, s' is an integer mod n, so its inverse is also an integer mod n.
4. Compute $u_1 = ec \bmod n$ and $u_2 = r'c \bmod n$.
5. Compute the elliptic curve point $(x_1,y_1) = u_1\mathbf{P} + u_2\mathbf{Q}$.
6. Convert x_1 to an integer, $\overline{x_1}$. See Step 5 on page 73 for details.
7. Compute $v = \overline{x_1} \bmod n$

If $v = r'$, the signature is verified. If they are different, the signature is invalid.

The Math

The ECDSA algorithm depends in part on the fact that if $r = r' \bmod n$, then $r\mathbf{P} = r'\mathbf{P}$. (See "The Point P and its Order" on page 68.)

The following calculations are really just a series of substitutions that can be made by looking back at the definition. You may find it more convincing to go through the substitution steps yourself, by glancing back at the sections "Creating the Key Pair", "Signing a Message", and "Verifying a Signature" immediately above.

If the message has been signed correctly, then $s = s'$. Expanding the elliptic curve point $(x_1,y_1) = u_1\mathbf{P} + u_2\mathbf{Q}$ calculated by the recipient, we see that:

$$u_1\mathbf{P} + u_2\mathbf{Q} = es^{-1}\mathbf{P} + rs^{-1}\mathbf{Q}$$
$$= s^{-1}(e\mathbf{P} + r\mathbf{Q})$$

Recall that $\mathbf{Q} = d\mathbf{P}$, so:

$$u_1\mathbf{P} + u_2\mathbf{Q} = s^{-1}(e\mathbf{P} + r\mathbf{Q})$$
$$= s^{-1}(e\mathbf{P} + rd\mathbf{P})$$
$$= s^{-1}(e + rd)\mathbf{P}$$
$$= s^{-1}(e + dr)\mathbf{P}$$

Now recall that $s = k^{-1}(e+dr) \bmod n$, so:

$$u_1\mathbf{P} + u_2\mathbf{Q} = s^{-1}(e + dr)\mathbf{P}$$
$$= [k^{-1}(e+dr)]^{-1}(e + dr)\mathbf{P}$$

$= (k^{-1})^{-1}(e+dr)^{-1}(e+dr)\boldsymbol{P}$
$= k\boldsymbol{P}$

This is the point calculated by the recipient. But this is also the point generated by the sender. The recipient then checks that the *x*-coordinate of the calculated point is in fact the *x*-coordinate that was received.

Elliptic Curve Authenticated Encryption Scheme (ECAES)

You can use elliptic curves to create an authenticated encryption scheme with a public/private key pair.

As always with elliptic curves, we assume that the elliptic curve parameters have been defined in advance. Suppose Bob has a key pair based on these parameters. The pair is $(\boldsymbol{Q},k_2)$, where $\boldsymbol{Q} = k_2\boldsymbol{P}$, where $\boldsymbol{P}$ is the base point of prime order specified in the elliptic curve parameters. The point $\boldsymbol{Q}$ is the public value and the number k_2 is the private value.

Encrypting a Message Using the Public Key

Anyone who wishes to send Bob an encrypted message can do so using the elliptic curve parameters and $\boldsymbol{Q}$. To encrypt a message *M*, where the length (in bytes) of the message is *f*, another party follows these steps:

1. Compute a random value, k_1, between *1* and $n - 1$.
2. Compute the elliptic curve point $\boldsymbol{Q}_1 = k_1\boldsymbol{P}$. This will be transmitted along with the encrypted message.
3. Compute the elliptic curve point $\boldsymbol{S}_1 = k_1\boldsymbol{Q}$. $\boldsymbol{S}_1$ is a pair (x_1,y_1). This is the secret information the sender uses to encode the message.
4. Compute a one time pad, *otp*, of length *f*, from x_1 using a key derivation function (KDF). *otp* is a concatenation of a series of hashes; it is constructed using *f*, x_1, and SHA1. *otp* is described below. The description uses the following notation: (1) $||$ denotes the concatenation of two numbers, (2) for a number *a*, [*a*] denotes the integer part of *a*. In particular, [*f*/160] denotes the integer part of *f*/160.
   a. Initiate a 32-bit, big-endian bit string *counter*. In hex, *counter* is intialized to 00000001_{16}.
   b. For i = 1 to [*f*/160], create a series of hashes, as follows:

Compute $Hash_i$ = SHA1($x_1 \parallel counter$), that is, the SHA1 hash of the concatentation of x_1 and *counter*.

Increment *counter*.

Increment i.

c. We want the length of the pad to be exactly the same as the length, f, of the message M. If $f/160$ is not an integer, we need to truncate the last hash to make the lengths equal. Therefore, we define $Hash'_{[f/160]}$ as follows:

$$Hash'_{[f/160]} = \begin{cases} Hash_{[f/160]} & \text{if } f/160 \text{ is an integer} \\ \text{the } [f/160] - (160 \times [f/160]) \text{ leftmost bits of } Hash_{[f/160]} & \text{if } f/160 \text{ is not an integer} \end{cases}$$

d. Set *otp* to be the concatenation of the series of hashes:

$otp = Hash_1 \parallel Hash_2 \parallel \ldots \parallel Hash_{[f/160]-1} \parallel Hash'_{[f/160]}$

5. Compute $M' = otp$ XOR M.
6. Compute an authentication tag, tag = SHA1 ($x_1 \parallel M'$). That is, *tag* is the SHA1 hash of concatenation of the x-coordinate of the secret point $k_1\boldsymbol{Q}$ and the message M'. Since *tag* is an SHA1 hash, *tag* is 20 bytes long.
7. Transmit the ciphertext $c = (\boldsymbol{Q}_1, M', tag)$. The total length of c in bytes is: 21+2 · (the length of a field element in bytes) + f.

Decrypting a Message Using the Private Key

A message that has been encrypted as above can be decrypted using the private key, as follows:

1. Parse the received ciphertext $c = (\boldsymbol{Q}_1, M', tag)$ into its components, $\boldsymbol{Q}_1$, M', and *tag*.
2. Use the private key k_2 to compute the elliptic curve point $\boldsymbol{S}_2 = k_2\boldsymbol{Q}_1$. $\boldsymbol{S}_2$ is a pair (x_2, y_2). If the message was transmitted correctly and encoded with the correct public key, $\boldsymbol{S}_2$ is equal to $\boldsymbol{S}_1$.
3. To verify that $\boldsymbol{S}_2$ is equal to $\boldsymbol{S}_1$, compute tag' = SHA1 ($x_2 \parallel M'$). If tag' is different from *tag*, output an error and stop.
4. Compute a one time pad, otp', of length f, from x_2 using the key derivation function outlined in Step 4 on page 75. Use x_2 instead of x_1. Since $x_1 = x_2$, $otp' = otp$.

5. Compute $M = otp$ XOR M'.

Elliptic Curve Diffie-Hellman Key Agreement

It is possible to construct a version of the Diffie-Hellman key agreement that uses elliptic curves. (For more information on Diffie-Hellman key agreement, see "Diffie-Hellman Public Key Agreement" on page 61.) Like Diffie-Hellman, EC Diffie-Hellman provides for key agreement, but not encryption or authentication.

The elliptic curve Diffie-Hellman key agreement algorithm provides a method for two parties to each compute the same secret key without exchanging secret information. The algorithm is made up of two parts: Phase 1 and Phase 2. Before they begin, the two parties must agree on the elliptic curve parameters: a base field, an elliptic curve over the base field, and point $\boldsymbol{P}$ of prime order, along with its order n. See the section "Elliptic Curve Parameters" on page 65 for details.

Phase 1

The first party randomly generates a private value, a number k_1, greater than 0 but less than n. Similarly, the second party generates a random private value, k_2.

Each party then computes a public value. To do this, they each compute $\boldsymbol{R}_i = k_i\boldsymbol{P}$. For each party, this is an elliptic curve point. The two parties exchange their public values.

These private and public values correspond to the private and public key components of a key pair. The public value is generated in such a way that computing the private value from the public value is computationally infeasible.

Phase 2

Each participant computes the agreed-upon secret key, z, from the other's public value, $\boldsymbol{R}_j$, and their own private value, k_i. The parties compute $k_i\boldsymbol{R}_j$ to get the elliptic curve point $\boldsymbol{S}$. This is a pair, (x_S, y_S). They then use the first coordinate of $\boldsymbol{S}$, x_S, as their secret value.

Even with knowledge of the parameters and both public keys, an outside individual will not be able to determine the secret key. One must have one of the private values to determine the secret key. This means secret information is never sent over unsecure lines.

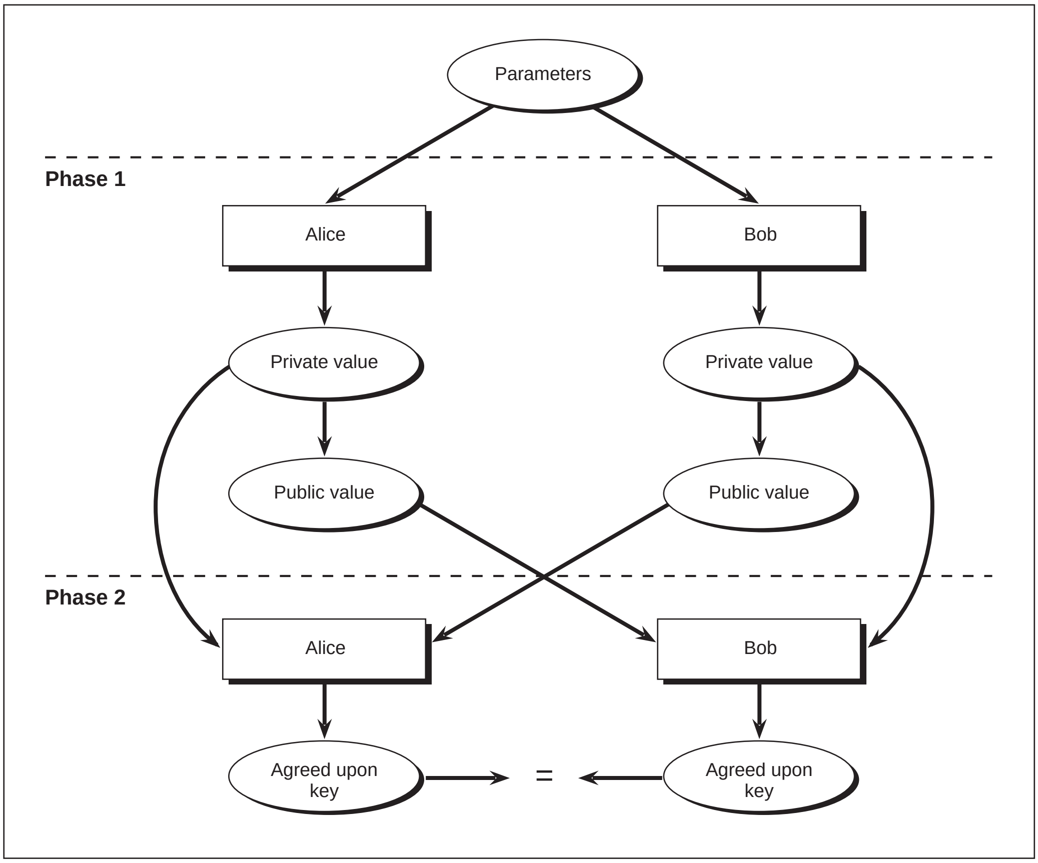

Figure 2-13 **Elliptic Curve Diffie-Hellman Key Agreement**

The Math

Even though the two parties involved are making computations using different private values, they will both end up with the same secret key, as illustrated by the following.

$\boldsymbol{P}$: point on the elliptic curve
k_1: 1st party's private value
k_2: 2nd party's private value
$\boldsymbol{R}_1$: 1st party's public value

$\boldsymbol{R}_2$: 2nd party's public value
x_S: secret key

In phase 1, each party computes a private value, k_i, and then a public value, $\boldsymbol{R}_i$:

$$\boldsymbol{R}_1 = k_1\boldsymbol{P}$$

$$\boldsymbol{R}_2 = k_2\boldsymbol{P}$$

In phase 2, the parties trade public values and compute the same elliptic curve point $\boldsymbol{S}$:

$$\boldsymbol{S} = k_1\boldsymbol{R}_2 = k_1k_2\boldsymbol{P}$$

$$\boldsymbol{S} = k_2\boldsymbol{R}_1 = k_2k_1\boldsymbol{P}$$

The first coordinate of $\boldsymbol{S}$, x_S, is their agreed-upon secret key.

Secret Sharing

Secret sharing, also known as a *threshold scheme*, takes a message or other data and divides it up into pieces in such a way that while each piece means nothing individually, some or all of the pieces can be assembled to retrieve the secret. Typically, the secret is a key used for encrypting sensitive data.

A good secret-sharing algorithm allows an application to share the secret among a variable number of shares. It should also be possible to set how many of the shares are needed to recover the secret. That is, if the total number of shares is N, you should be able to decide in advance that any K of them can recover the secret. The number K, the required number of shares, is known as the *threshold*.

With secret sharing, access can be split among several individuals, with reconstruction requiring a threshold number of shares. In this way, if one or more of the individuals are not available, it is still possible to recover the data. In addition, secret sharing contains some level of checks and balances: no one can recover data without at least one other individual knowing about it.

The algorithm used in Crypto-C is Bloom-Shamir secret sharing.

Figure 2-14 and Figure 2-15 show the schema for secret sharing and recovery.

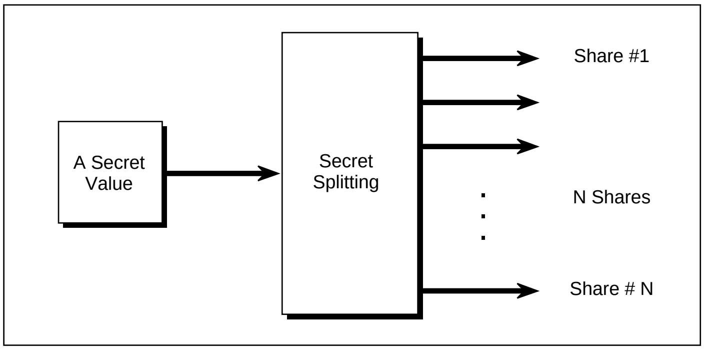

Figure 2-14 **Secret Sharing — Key Share Assignment**

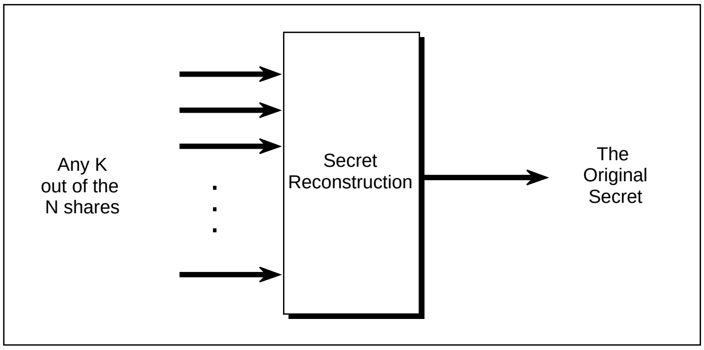

Figure 2-15 **Secret Sharing — Full Key Generation From Shares**

Working with Keys

Key Generation

The techniques for generating public/private key pairs and symmetric keys are quite different. Symmetric-key algorithms generally require an arbitrary random-byte sequence, while a public/private key pair must satisfy a mathematical formula. Key generation depends on the availability of a good random number generator, and the security of a random number generator depends on the seed. See “Pseudo-Random Numbers and Seed Generation” on page 92 for more information.

Key Management

The term *key management* refers to the collection of processes and methods for assigning the right keys to communication sessions, providing the right keys to the right persons, and making sure unauthorized personnel cannot gain access to keys. Key management is the most difficult security problem. To manage keys properly, an application must address the following issues.

- Generating keys
- Choosing appropriate values for the keys
- Guarding the privacy of keys transmitted between nodes
- Verifying the authenticity of keys transmitted between nodes
- Using keys in a software environment in an open system
- Keeping backup keys
- Dealing with compromised keys
- Destroying old keys
- Changing keys

Often, the bulk of a security application's focus will be on key management. Crypto-C provides a rich suite of cryptographically secure algorithms, but it is up to the application designer to carefully consider how to manage the keys.

Key Escrow

Key escrow allows a designated authority or authorities to recover keys belonging to someone else. This can be a desirable feature when users lose access to their keys because they leave an organization or simply forget a password. Key escrow can be implemented through secret sharing or by encrypting keys with a security officer's RSA public key and storing the encrypted copy. To recover the escrowed key, you must either assemble the necessary shares or have the security officer decrypt the encrypted key using the appropriate RSA private key.

Key escrow is never automatic with Crypto-C. There is no Crypto-C encryption method that offers key escrow as part of the algorithm; the developer must make key escrow part of the application. Crypto-C offers the techniques to implement key escrow, but it is the developer's responsibility to decide whether it will be part of the application, and if so, how it will be executed.

ASCII Encoding and Decoding

ASCII encoding and decoding is required when you need to send encrypted or signed data using communication protocols that allow transmission of printable characters only. In this case, the application must convert the encrypted 8-bit values to a string of printable characters. Crypto-C uses the Internet RFC1113 method for implementing ASCII-encoding. The Internet Draft RFC1113 is a publication that describes this system.

Applications of Cryptography

Crypto-C offers application developers the tools to add privacy and authentication features to software and hardware systems. This section discusses a number of areas where such features are useful.

Historically, *privacy* has been the main use of cryptographic techniques. In these applications, cryptography is used to hide critical information from eavesdroppers or unauthorized personnel. Crypto-C provides algorithms and methods for encrypting data in a variety of applications.

Authentication is a cornerstone of the forever-pursued paperless office. Authentication enables users to prove authenticity and authorship of messages and non-tampering of data.

Cryptography can be useful in any of the following situations:

- Local applications, to control access and prevent tampering.
- Point-to-point applications, to protect the privacy of communications.
- Client-server applications, to control access and provide authentication.
- Peer-to-peer applications, to protect privacy between nodes.

Local Applications

One of the most basic applications of cryptography is local file encryption. There are many reasons why one would find it useful to encrypt files even if they are not being transmitted. For example, you can use cryptographic techniques to:

- Save files in encrypted form to protect against unauthorized access.
- Ensure file integrity and protect against tampering. Cryptographic techniques can be used to guarantee that only authorized personnel can modify or install certain files.
- Archive important data so that it can be accessed only by authorized personnel.
- Protect intellectual property.

Point-To-Point Applications

Applications that require establishing a secure link between two nodes are very common and may have different topologies. However, their similarities allow them to be treated in a comparable manner. Secure point-to-point communication is needed if:

- Communication takes place between exactly two nodes.
- The primary security consideration is to allow the two nodes to communicate privately and to prevent others from eavesdropping on the traffic.

Here are some applications that require secure point-to-point data communication:

- Computer hardware links connecting two nodes
- Satellite or cellular communications
- A single transaction between two nodes in a larger network

Here is a typical scenario for implementing applications in this class, using key agreement with stream-cipher encryption.

1. Compute the Diffie-Hellman parameters for both nodes. This must be done before a communication session is established. When a link is requested, the parameters should be waiting for the nodes.

   A new Diffie-Hellman parameter set is not necessary each time you generate a session key; it is safe to use one set of Diffie-Hellman parameters for many key-agreement sessions. In addition, either of the nodes can generate the parameters and transmit the values over any channel.
2. Establish an agreed-upon secret value using Phase 1 and Phase 2 of the Diffie-Hellman key-agreement protocol. See “Diffie-Hellman Public Key Agreement” on page 61 for an overview of this process.
3. Compute an RC4 key for the session using the agreed-upon secret value. The RC4 key may be shorter than a Diffie-Hellman secret value. The application must determine the procedure for extracting the required bits. A single Diffie-Hellman agreement may also be used to generate multiple RC4 keys.
4. Perform the encryption and decryption using RC4 with the established key. If the application requires multiple session keys, use a message digest on the agreed-upon secret value and a counter to generate a new key.

There is an attack against this kind of protocol known as “man-in-the-middle.” Someone could intercept all messages between the two parties and pose as each individual’s other participant. For example, if Alice wants to communicate with Bob, she sends a message to initiate a session. The man-in-the-middle intercepts Alice’s message, builds a secure session with Alice, and initiates his own session with Bob. Now, all messages Alice sends to Bob go through the attacker. The man-in-the-middle decrypts Alice’s messages based on the session he created with Alice and saves the results to examine later. He then reencrypts the message based on the session he created with Bob. If a particular application is vulnerable to such an attack, it is advisable to use a peer-to-peer protocol (see page 86) instead.

Client-Server Applications

A client-server application is distinguished by one central server node that provides services to several client nodes. Many client-server applications have a need for cryptographic tools. For example:

- *Network applications*: Any network that connects several computer nodes to one central server, such as a local or wide area network, can use cryptography to establish secure communications between the clients and the server. The network can also employ authentication to guarantee that intruders do not have access to the network.
- *Database applications:* Multiple clients — in this case, database queries — need access to a server — the database. To ensure that not all fields in the database are accessible to all clients, restricted fields can be encrypted or signed. In addition, by distributing secret shares among authorized personnel, you can ensure that very sensitive data can be accessed only according to the security rules.
- *Cryptographic smart cards:* Here, you must authenticate users to service providers such as banks. A smart card holds the individual private keys and includes a processor that runs the cryptographic algorithms needed to achieve the appropriate authentication level.

In all these applications, the server generates a public/private key pair for use with all clients requiring secure communications. The server uses the private key to sign digital certificates for all nodes that require access to the server and its resources.

It also starts a public key table to register client RSA public keys. Each client computes an RSA public/private key pair when it is first established as a secure client. The public key is communicated to the server and an entry is made in the table maintained by the server for the public keys.

As an alternative, the server can certify the public keys of the client nodes by generating a digital certificate to be signed by the server's private key. In this case, the server only trusts messages from previously-certified keys. There is no table to maintain because the digital certificate can be used to verify the identity of a node each time a connection or request is needed.

There are two approaches to establishing a link between a client and the server.

In the first approach, the server and a client determine a session key using a Diffie-Hellman key agreement protocol. The Diffie-Hellman parameters are established once at the initial setup of the server, and communicated publicly to each client when a secure connection is requested. The session key is used for bulk-data encryption; the established client RSA key pair is used for authentication or for envelope communications. Any block or stream cipher can be used for encryption with the session key. For stream ciphers, a new key should be computed for each session; this prevents attacks that compare blocks of data encrypted with the same key.

In the second approach, the server uses the client’s RSA public key (contained in the digital certificate) to generate a digital envelope for the encrypted data sent from the server to the client. Likewise, the client uses the server’s public key (known to all nodes) to create a digital envelope. In addition, each message contains digital signatures to authenticate the originator.

Peer-To-Peer Applications

Unlike a client-server application, a peer-to-peer network application provides each node with access to any other node in the network. For example, users may wish to communicate privately with other known or unknown users through secure email. In the peer-to-peer situation, there is no single node capable of authenticating other client nodes.

Digital signatures can be used to provide proof of authorship to any recipient. Each node must generate its public/private key pair and obtain a digital certificate from some approved central authority. VeriSign can provide details about how to obtain a digital certificate.

Each message between any two or more nodes can be authenticated by attaching the originator’s digital certificate to the message. The recipient can verify the authenticity of the message and the originator by verifying the validity of the certificate.

Nodes on peer-to-peer applications can encrypt using digital envelopes. To do so, the sender obtains the digital certificate of each recipient and extracts the public key.

Choosing Algorithms

In some cases, an application's constraints determine the algorithm. In other cases, the developer can choose among a number of options and still produce a viable solution. This section presents suggestions to help you determine the best choice.

Public-Key vs. Symmetric-Key Cryptography

Because symmetric-key encryption algorithms are much faster than public-key algorithms, they are most suited for bulk data encryption.

Public-key encryption should not be used for encrypting large amounts of data. It is best used to encrypt keys for either a digital envelope method or for key escrow applications.

Stream vs. Block Symmetric-Key Algorithms

Crypto-C has only one stream encryption algorithm, RC4. RC4 produces an encrypted output the same size as the original input message and is significantly faster than block-encryption algorithms. However, once a key has been used to encrypt a particular message, it should not be used again. Hence, employing RC4 requires using many keys. If managing many keys is difficult, RC4 may not provide the easiest solution.

Some applications do not save keys outside of the session. For these applications, RC4 will generally be a good choice. For instance, in encrypted phone conversations, the symmetric key is a session key. It encrypts for one call; once the session is over, the key is discarded. Another example would be an email application where the session key is encrypted with an RSA public key and is a part of the data package.

RC4 has a variable length key. If you set the key to be long enough, RC4 offers greater security than DES. The key can also be set to a level low enough to obtain export approval.

Block-encryption algorithms are best used for applications that require repeated encryptions without changing the value of the key. In addition, DES is a standard used by many applications. If an application must be able to communicate with other applications, DES is a safe choice for universal support.

Block Symmetric-Key Algorithms

The following considerations may help when choosing between DES, DESX, Triple DES, RC2, and RC5 block algorithms.

DES is a standard algorithm in use by many applications. Using DES ensures widespread connectivity. However, DES is limited to an effective key size of 56 bits. The cryptography community expects that, because of the continued increase in computing power, within a few years, DES will not be strong enough to withstand attacks. Triple DES is gaining in acceptance as a substitute for DES to counter this problem.

DESX is viewed as a cheap and secure alternative to Triple DES.

RC2 is faster in software than DES and Triple DES and has gained momentum in the marketplace, although it is not as widely implemented as DES. In addition, RC2 employs a variable key size, which allows you to increase the security beyond that supplied by DES or Triple DES, or to decrease security to the level necessary to obtain export permission.

RC5 is even faster than RC2; its speed and security can be increased or decreased through the word size, rounds, and key length parameters. It is a new algorithm, so does not have a history of withstanding attacks and analysis. Although the early reports are that it is just as secure, if not more so, than RC2, some developers may shy away from using it because of its youth.

If communication with other applications is not an issue, RC2 and RC5 offer greater security and are much faster in software than DES. RC2 is exportable, and RC5 is likely to receive export permission as well.

Key Agreement vs. Digital Envelopes

Both key agreement and digital envelopes allow two nodes communicating over an unsecure medium to establish a secret symmetric-encryption key.

Key agreement is easier and faster when the two nodes are in current contact, such as in a phone conversation. Crypto-C employs the Diffie-Hellman key agreement algorithm and the implementation requires an interactive session.

Digital envelopes are more convenient when the contact between nodes is not interactive, such as email. One node can send a message to another without waiting for the other node to respond.

To thwart man-in-the-middle attacks, authentication by digital signatures should be built into any communication system.

Secret Sharing and Key Escrow

Also known as *emergency access*, secret sharing and key escrow both allow for recovery of keys by parties other than the owner. Without some form of emergency access, data that is encrypted using a session key that is itself protected by password-based encryption is inaccessible or even lost if the owner forgets the password or is unavailable.

To enable recovery using key escrow, you can encrypt all session keys with a security officer's RSA public key. Any time access is required, the officer can decrypt the session key with the appropriate RSA private key. This method is the easiest to implement and execute. However, it requires trust in the security officer not to abuse this power, and it requires that a single individual be available.

With secret sharing, access can be split among several individuals, with reconstruction requiring a threshold number of shares. In this way, if one or more of the individuals are not available, it is still possible to recover the data. In addition, secret sharing contains some level of checks and balances: no one can recover data without at least one other individual knowing about it.

Elliptic Curve Algorithms

Elliptic curve cryptosystems have recently come into strong consideration, particularly by standards developers, as alternatives to established standard cryptosystems such as the RSA cryptosystem, Diffie-Hellman, and DSS. Elliptic curve cryptosystems have a number of interesting properties, which may make them appropriate tools for meeting security requirements in some cases, and not in others.

From a cryptographic perspective, the primary motivation for development of elliptic curve cryptosystems is that they are based on a different hard mathematical problem than established systems, and appear to have a reasonable expectation of security, without significant additional cost. In particular, in certain applications, elliptic curve cryptosystems can provide security where other systems currently do not fit. However, the range of applications where they make a significant difference is limited. In typical applications of cryptography, public-key operations are employed in combination with other techniques. In particular, public-key operations often represent only a minor overhead in the total processing, whether in storage or in computation time. A "faster" or "smaller" public-key technique thus may have little overall impact in many applications.

Elliptic curve cryptosystems have, at this point, relatively fewer cryptanalytic results than established systems. It could be that the systems are stronger, or it could be that they are just not that well understood. In either case, this is an observation that calls for further study.

In conclusion, RSA Data Security, Inc., is currently recommending that elliptic curve cryptosystems continue to be studied as additional tools in the public-key repertoire, and that they be considered as near-term solutions in the particular cases where the alternative would be to have no security at all.

For more information about elliptic curve cryptosystems, see the RSA Laboratories technical note, *Recommendations on Elliptic Curve Cryptosystems*, at `http://www.rsa.com/ecc/html/recommendations.html`.

Interoperability

Elliptic curve public-key methods can be constructed in a number of ways. Parameters can be chosen over odd prime fields or fields of even characteristic. The underlying mathematics of these implementations is different enough that a successful implementation of only one of these approaches could not handle another implementation. In essence, this means that one could build two different cryptosystems, both using elliptic curve cryptography, but unable to interoperate with each other.

The two main interoperability issues for elliptic curve cryptosystems are:

- the choice of finite field over which the elliptic curve is defined
- the representation of elements in the finite field.

There are two types of finite fields: finite fields with p elements, where p is an odd prime, denoted F_p, and called "odd prime fields," and a finite field with 2^m elements for some integer m, denoted F_m, and called "even characteristic." It is not possible to convert between the two types of finite field, so the choice of finite field is critical to interoperability.

The even characteristic implementations offer greater gains in hardware implementation. However, the odd prime implementations can use the same special-purpose circuitry that is available for implementations such as RSA. This can make the odd characteristic a better choice for situations where RSA hardware is already in place, or where a hardware developer wants to be able to provide a platform that supports both RSA and elliptic curve encryption.

For the even characteristic finite field, F_{2^m}, there is also a choice of representation. For these fields, elements can be represented using a polynomial basis, a normal basis, or some other basis. For some values of *m*, elements can also be represented in an optimal normal basis, which is generally more efficient than an ordinary normal basis. In order for systems that use different bases to communicate, they need to convert from one representation to another. Each representation has advantages and disadvantages, including efficiency and potential patent coverage, so in current elliptic curve standards the choice is typically left to the implementation.

Elliptic Curve Standards

The elliptic curve algorithms in Crypto-C are based on a number of draft standards. Several standards bodies are already working on various elliptic curve cryptographic standards. These include the IEEE P1363, the ANSI X9 Financial Standards, and ISO/ IEC SC27.

Security Considerations

Handling Private Keys

In public-key cryptography, only the owner of a private key can create a digital signature or open digital envelopes. However, if someone other than the owner is able to obtain the private key, the security fails. To ensure that no one other than the owner has access to a private key, it should be stored encrypted, generally with a password-based encryption method. An application will decrypt the private key when it is needed. Always overwrite the memory that held a private key with zeroes or random bytes immediately after the key has performed its function.

Operating systems will frequently use the hard disk space as virtual memory and so an unencrypted private key may, through no intention of a user, end up on a hard disk. Hence, for key buffers, an application should use the operating system's mechanisms that ensure allocation of core memory, and not virtual memory.

It is a good idea to generate new public/private key pairs every so often to thwart long-term factoring attacks. Material encrypted using the old key pair should be re-encrypted with the new. However, an application may not have access to all material protected by an old key pair, so it may be necessary to retain old key pairs in a secure environment.

Temporary Buffers

Even though a temporary buffer may not contain a private key, it still may hold sensitive data, such as a message to be encrypted or a symmetric key. Such temporary buffers require the same security as private-key buffers. After using the data, overwrite the buffer with zeroes or random bytes. Make sure the operating system uses core memory and not hard disk virtual memory.

Pseudo-Random Numbers and Seed Generation

Crypto-C uses pseudo-random number algorithms for generating both symmetric keys and public/private key pairs. The random number generation algorithms are the same as the message digest algorithms, and are verified to have very high degree of randomness.

Any method that is employed to generate random values begins with a random seed.

The security issue then becomes one of making sure that an attacker cannot determine the seed. Generally, any random number generator will produce pseudo-random numbers, given any seed. Therefore, to generate random number, you do not need to start with a seed that is itself random. However, the seed should be "unrepeatable." That is, no one should be able to apply some sort of algorithm which can "guess" the seed in a reasonable amount of time.

For instance, suppose that a message was encrypted using RC2 with 80 effective key bits from 10 bytes of key data, but that the key data was generated using an MD5 random byte generating algorithm with a 4-byte seed. An attacker could try every possible 10-byte key combination to crack the message, or could try every 4-byte seed combination to generate 10 bytes of key data. Further, suppose that 4-byte seed was the time of day. Now the attacker has an even smaller range of possible seeds to test before finding the right one.

The seed should contain at least as many unrepeatable bits as the key. If the seed is based on a user's typing a series of letters and characters on the keyboard, then an attacker can predict two or three of the bits in each seed byte. Bit 7, for instance, will always be 0. Furthermore, many of the keystrokes can be predicted: they will probably be lower-case letters that alternate between the left and right hand. Analysis of this issue has determined that there is only one bit of entropy from each keystroke (think of the term "entropy" as "unrepeatability"). When using keystrokes, use at least one for each bit of key size.

There are other schemes for finding seed bytes, including tracking mouse movements, timing keystrokes, "listening" to hardware noise, or capturing machine state information. Many schemes will provide more than one bit of entropy per byte of seed; however, it is an easy-to-remember rule of thumb to use as many bytes of seed data as bits of key.

Whatever the scheme, it may seem unusual to expend more effort to produce a seed than it will take to produce the random key data itself. Why not simply use the seed data in the key? The strength of cryptography relies on key data that is random or pseudo-random. If an attacker knows that the key data is not random, cracking the cipher becomes easier. The seed will almost certainly not be random. The eavesdropper may not be able to repeat the seed gathering process exactly, but non-random key data leaves a cipher algorithm as a whole open to various attacks. Hence, use a large unrepeatable seed to generate pseudo-random data.

Choosing Passwords

In almost any security application, users are required to have passwords that indicate authorized access to the system. Often, when given a choice, users choose the same password for various applications. For instance, they may use their login password to encrypt a private key. Many times, users will choose passwords an attacker can easily deduce. Therefore, it is a good idea for developers to build good password protocols into their applications. The following are a list of possible guidelines in choosing passwords.

- Enforce a minimum password length, generally eight characters.
- Inform users to avoid “easy to guess” passwords, such as common names or birthday dates.
- Check an entered password against a dictionary.
- Require a combination of numeric, special, and upper- and lower-case alphabetic characters.
- Include support for password expiration dates to limit the available searching time an attacker has to break into the system.

Initialization Vectors and Salts

Although IVs and salts are not secret information, it is still a good idea to use random values. If a salt is not random, an attacker will have much fewer precomputations to make in generating keys from possible password/salt combinations.

An IV should also be used for only one message. Using the same IV with the same key on two separate messages may provide an attacker with useful information.

DES Weak Keys

Researchers over the years have found that some DES keys are more susceptible to attack than others. Some of these keys are known as “weak,” others, when used in pairs, as “semi-weak.” Using a weak key or a semi-weak pair may not necessarily pose a security risk; it could depend on the mode of DES. However, it is simply easier to avoid these keys (listed in Table 2-3) altogether.

Table 2-3 **DES weak and semi-weak keys**

0101010101010101
FEFEFEFEFEFEFEFE
1F1F1F1F1F1F1F1F
E0E0E0E0E0E0E0E0

Table 2-3 **DES weak and semi-weak keys**

01FE01FE01FE01FE
1FE01FE00EF10EF1
01E001E001F101F1
1FFE1FFE0EFE0EFE
011F011F010E010E
E0FEE0FEF1FEF1FE
FE01FE01FE01FE01
E01FE01FF10EF10E
E001E001F101F101
FE1FFE1FFE0EFE0E
1F011F010E010E01
FEE0FEE0FEF1FEF1

Stream Ciphers

A stream cipher (such as RC4) will create a stream of pseudo-random bytes based on the secret key; this is known as the key stream. To encrypt, you XOR the plaintext with the key stream, byte by byte. The XOR operation has the property that the ciphertext XORed with the same key stream decrypts, restoring the plaintext. This also means that an XOR operation between the plaintext and the ciphertext will reproduce the key stream. Hence, knowing or guessing part of the plaintext allows an attacker to determine the corresponding part of the key stream. This still will not enable the attacker to deduce the entire key or any more of the key stream, but this does pose a threat if the same key is used in two different messages.

The same key always produces the same key stream. Therefore, if two messages use the same key, knowing part of the key stream in one message means knowing the same part of the key stream in the second message. An attacker can thus determine some of the plaintext in the second message. That is why you should never use the same stream cipher key twice.

Incidentally, this attack does not work on block ciphers. Knowledge of part of the plaintext does not automatically grant to the attacker knowledge of the key.

Another stream cipher attack involves a dictionary of key streams. Suppose you had an application you wanted to export and so kept the key size to 40 bits. An attacker could create a dictionary of the first eight bytes of the key stream from every possible 40-bit (5-byte) key. Then, the attacker “decrypts” the first eight bytes of an intercepted message with each possible key stream, until one produces a reasonable result. The key that generated the stream that worked is the right one.

To thwart this attack, you can implement salting. Design the application to use an 80-bit (10-byte) key, but send 40 bits in the clear. That 40 bits in the clear is known as a salt. For example, in an email application, encrypt the message using RC4 with a 10-byte key. Then encrypt the first five bytes of the key using the recipient's RSA public key. Now the RSA digital envelope consists of the public-key-encrypted five secret bytes, five salt bytes sent in the clear and the RC4-encrypted message. In this way the attacker's dictionary is rendered useless. That dictionary is valid for 40-bit keys, but the message used an 80-bit key. Still, only 40 bits are kept secret to comply with export regulations. A dictionary of 80-bit key streams is not feasible — it would require 2^{80} entries. That is about $1.2 \cdot 10^{24}$, or 1.2 times one trillion times one trillion.

Timing Attacks and Blinding

If the time it takes to execute a cryptographic operation varies based on the input parameters, then in theory, an attacker with access to accurate timings can determine unknown values. This is the case for RSA, Diffie-Hellman, and DSA operations. For instance, in an RSA signing operation, purportedly an attacker who knows the message being signed and exactly how long it takes to create the digital signature can determine the signer's RSA private key.

Currently, there is no known successful implementation of such a procedure. Proposed algorithms under scrutiny either require several absolutely exact timings or thousands of inexact (but still accurate to the millisecond) timings to succeed. However, there are two simple ways to guard against this attack. One is to "equalize" private key operations, by padding shorter transactions with a few extra milliseconds to make sure that all times are the same. The second method is known as *blinding*.

For a timing attack to succeed, the eavesdropper must know that the input being processed is only altered before the operation is performed and that the true answer is recovered after the operation by reversing the alteration procedure.

For example, in an RSA signature operation, the input is the BER-encoding of the digest of the data to sign and some pad bytes. To blind the attacker, that input is modular multiplied by a secret random number. Then the product, not the input, is modular exponentiated. To produce the actual signature, the result is modular multiplied by the inverse of the random number.

In mathematical terms, instead of performing the usual RSA encryption process:

$$sig = m^{d} \bmod n$$

pick a random value r and compute:

$$m' = mr^e \bmod n$$

where e is the public exponent. Now find:

$$s = (m')^d \bmod n$$

Then to compute the actual signature, find:

$$sig = (r^{-1}) \cdot s \bmod n$$

In this way, the timing attack fails because the attacker does not know what value was exponentiated.

To see that the signature is the same in both cases, note that:

$$\begin{aligned} r(mr^e)^d \bmod n &= (r^{-1})(m)^d(r^e)^d \\ &= (\ r)(r^{ed})(m^d) \\ &= (\ r^{-1})(r)(m^d) \\ &= (1)(m^d) \bmod n \end{aligned}$$

Crypto-C offers both blinding and non-blinding RSA private operations through separate algorithm methods. It currently offers no blinding technique in Diffie-Hellman or DSA operations.

Crypto-C uses MD5 random number generation to produce the random value r. The seed is the following digest:

$$\text{MD5}(p \parallel padP \parallel \text{MD5}(q \parallel padQ \parallel m))$$

where p and q are the two primes, *padP* and *padQ* are paddings of zeros to make sure the length is a multiple of 64 bytes, and the symbol $\parallel$ means concatenation. An attacker will not know what r is without knowing what the seed is, and will not know what the seed is without knowing what p and q are. An attacker who knows p and q is not going to implement a timing attack to determine the private key, because knowledge of p and q is equivalent to knowledge of the private key already.

Choosing Key Sizes

In cryptography, security is measured in key size: the bigger the key, the greater the security. Key size, in turn, is measured in bits. However, that bit number might not describe the entire key.

For instance, a DES key is 56 bits. However, that size refers to its cryptographic size, not its “physical” size. To build a DES key, you need 64 bits, but because eight of those bits are “parity bits,” that is, bits that are known, out of the 64, you really only get 56 secret bits. Hence, a DES key, while consisting of 64 bits of data, is only 56 cryptographic bits large.

An RSA key pair measurement describes the modulus length. When cryptographers talk about a “768-bit RSA key pair,” what they really mean is that the modulus is 768 bits long. The security of an RSA key pair is tied up in how big the modulus is; hence, the measurement used is the bit size of the modulus. The actual keys themselves will contain more information than the modulus, so the “physical” size will be much larger.

In choosing a key size, if larger keys offer greater security, why not simply always choose the largest possible key? Larger RSA, Diffie-Hellman, DSA, and elliptic curve keys can slow down cryptographic operations. In addition, there are restrictions on key size for applications seeking export.

For RC2, RC4, and RC5, larger keys generally do not significantly degrade performance. However, larger keys do require more management.

Table 2-4 gives a summary of the recommended key sizes for the algorithms supported in Crypto-C. These recommendations were current at the time this manual went to press. Please note, however, that such recommendations are always provisional and can be affected by changes in the cryptographic state of the art.

Table 2-4 **Summary of Recommended Key Sizes**

Algorithm	User Key	Organizational or Long-Term Key	Root Key
Diffie-Hellman	768-bit prime	1024-bit prime	2048-bit prime
DSA	768-bit prime	1024-bit prime	2048-bit prime
ECAES	160-170-bit modulus	Not recommended at this time	
EC Diffie-Hellman	160-170-bit modulus	Not recommended at this time	
ECDSA	160-170-bit modulus	Not recommended at this time	
RC2	8-128 effective key bits		
RC4	8-128 effective key bits		

Table 2-4 **Summary of Recommended Key Sizes**

Algorithm	User Key	Organizational or Long-Term Key	Root Key
RC5	8-128 effective key bits with 16 rounds for 32-bit word or 20 rounds for 64-bit word		
RSA	768-bit modulus	1024-bit modulus	2048-bit modulus

RSA Keys

The security of the RSA algorithm is based on the difficulty of factoring large integers. Therefore, the choice for the key size depends on the efficiency of integer-factoring algorithms. Because users will probably want a key pair to last a few years, it is advisable to choose a size that will not only remain secure against current state of the art factoring, but will probably defeat improved factoring attempts of the future. The RSA Laboratories publication, "*Frequently Asked Questions About Today's Cryptography*" describes current factoring capabilities.

For normal user data, RSA Data Security, Inc. recommends a modulus size of 768 bits. For organization keys or for long-term applications, a 1024-bit modulus is advisable. For root keys, RSA Data Security, Inc. recommends a 2048-bit modulus. This safeguards against progress in factoring algorithms and improvements in computing power.

Diffie-Hellman Parameters and DSA Keys

The security of the Diffie-Hellman algorithm and DSA are both dependent on the complexity of computing logarithms modulo a prime number. Generally, this is equivalent to the complexity of the factoring problem, because modern factoring algorithms generally apply to the discrete logarithm problem. Therefore, the designer is advised to use similar sizes for the Diffie-Hellman parameters and DSA keys as for RSA operations: a 768-bit prime for user keys, 1024-bit prime for organizational keys and a 2048-bit prime for root keys.

Note: The Digital Signature Standard lists a maximum of 1024 bits for DSA, but the algorithm does not have an inherent limit. Crypto-C's implementation allows up to 2048-bit DSA keys.

RC2 Effective Key Bits

A key with 80 to 128 effective key bits is sufficient for most applications using the RC2 algorithm. Export regulations may limit the size to 48 effective bits. A key size of 40 bits generally expedites the export permission.

RC4 Key Bits

An 80- to 128-bit key is sufficient for most applications using the RC4 algorithm. Export regulations may limit the size to 48 bits. A key size of 40 bits generally expedites the export permission.

RC5 Key Bits and Rounds

An 80- to 128-bit key is sufficient for most applications using the RC5 algorithm. Note also that the security of the RC5 algorithm is dependent on the number of rounds. For RC5 with a 32-bit word size, RSA Data Security, Inc. recommends at least 12 rounds for applications; while no practical attacks are known for 12-round RC5-32, recent cryptanalytic work suggests 16 rounds is now a more conservative choice. For RC5 with a 64-bit word size, RSA Data Security, Inc. recommends at least 16 rounds; a conservative choice for the 64-bit version is 20 rounds. Note that the Crypto-C implementation of the 64-bit word is for evaluation purposes only.

Triple DES Keys

It is possible to implement Triple DES with one, two, or three keys. One key in EDE mode (encrypt-decrypt-encrypt) is equivalent to DES, and is used to provide compatibility with applications that only understand DES. There are known attacks against Triple DES using two keys, so RSA Data Security, Inc. recommends using three keys.

Elliptic Curve Keys

For prototyping and evaluation, RSA Data Security, Inc. recommends setting the order of the base point to be between 160 and 170 bits. Currently, RSA Data Security, Inc. does not recommend using elliptic curve cryptography for long-term applications.

Chapter 3

Using Crypto-C

Algorithms In Crypto-C

Whatever algorithm Crypto-C performs, it does so from an algorithm object. An algorithm object is used to hold information about an algorithm's parameters and to keep a context during cryptographic operations.

To build an algorithm object, create an empty object with `B_CreateAlgorithmObject`. Then, use `B_SetAlgorithmInfo` to fill the object with the information necessary to distinguish it as an object performing the desired operation. The information for `B_SetAlgorithmInfo` consists of two elements: an Algorithm Info Type, or AI, and its specific accompanying *info*. This chapter gives a brief summary of the AIs categorized by function.

Chapter 2 of the *Crypto-C Library Reference Manual* (LRM) gives a complete listing of AIs in alphabetical order. Each entry in the *Library Reference Manual* contains a description of information that must accompany the AI, including keys and algorithm methods.

Information Formats Provided by Crypto-C

There are four types of AIs in Crypto-C. These AIs differ in the format in which they provide information:

- Basic algorithm info types, which provide information in Crypto-C's internal format.
- BER-based algorithm info types, which provide information in a format that complies with Open Systems Interconnection's Basic Encoding Rules.
- PEM based algorithm info types, which provide information in a format that complies with the Privacy Enhanced Mail draft standard.
- BSAFE1 algorithm info types, which provide information in a format that is backward compatible with BSAFE 1.x.

Basic Algorithm Info Types

The basic algorithm is used to start a new process because its *info* (the accompanying information specific to the AI) is the simplest to format.

BER-Based Algorithm Info Types

BER-based algorithms are algorithms that comply with Basic Encoding Rules, as defined in ANSI X.690. BER-based algorithms are necessary because the format of the *info* in a basic AI is not standard. Much of the data in cryptography is passed between two or more individuals. Not every cryptographic application uses Crypto-C, and other packages may not organize the necessary information the same way. When one person needs to tell another person which algorithm was used to encrypt, for instance, there needs to be a standard way to present the information. The standard description of information is known as Basic Encoding Rules, or BER, which is a product of Open Systems Interconnection and is defined in ANSI X.690.

BER-based algorithms end with the letters BER. Such AIs will read in or output information according to the BER.

Unfortunately, BER is often complicated and it is difficult to determine the proper BER encoding without a translator. Therefore, it is simpler to use `B_SetAlgorithmInfo` to define algorithm objects with the basic algorithm AI, get the information in BER format using `B_GetAlgorithmInfo`, and send the BER-encoding to those who need the information. The recipient will translate the BER information into something they can understand.

When a Crypto-C application receives information in BER format, it can set using the BER AI and build an algorithm object to match that information.

PEM-Based Algorithm Info Types

The Privacy Enhanced Mail (PEM) draft standard is a product of the Internet

Activities Board, Network Working Group (see RFC 1421-1424). It defines the proper formatting of information passed between entities in electronic mail. Formatting information to follow this standard is fairly simple.

BSAFE1 Algorithm Info Types

The fourth kind of AI ends with `BSAFE1`. These algorithm info types are only for backward compatibility with applications using the BSAFE 1.x formats.

Summary of AIs

Table 3-1 **Message Digests**
Not all message digests are recommended. See “Message Digests” on page 46 for details.

Algorithm Info Type	Description	Standards	BER	PEM
AI_MD2	MD2 message digest	RFC 1319		
AI_MD2_BER	MD2 message digest; BER-encoded algorithm identifier	RFC 1319	✓	
AI_MD2_PEM	MD2 message digest with PEM	RFC 1423		✓
AI_MD5	MD5 message digest	RFC 1321		
AI_MD5_BER	MD5 message digest; BER-encoded algorithm identifier	RFC 1321	✓	
AI_MD5_PEM	MD5 message digest, PEM-encoded algorithm identifier	RFC 1423		✓
AI_MD	Supplied for backwards compatibility with the BSAFE 1.x message digest algorithm	none		
AI_SHA1	SHA1 message digest	FIPS PUB 180-1		
AI_SHA1_BER	SHA1 message digest; BER-encoded algorithm identifier	FIPS PUB 180-1		

Table 3-2 **Message Authentication**

Algorithm Info Type	Description	Standards
AI_MAC	BSAFE 1.x message authentication code; supplied for backwards compatibility with BSAFE 1.x	
AI_HMAC	Hashed-based Message Authentication Code	SET Draft

Table 3-3 **ASCII Encoding**

Algorithm Info Type	Description	Standards
AI_RFC1113Recode	ASCII/binary conversion	RFC1113/RFC1421; RFC1521; MIME Base64

Table 3-4 **Pseudo-Random Number Generation**

Algorithm Info Type	**Description**
AI_MD2Random	MD2 pseudo-random number generator
AI_MD5Random	MD5 pseudo-random number generator
AI_SHA1Random	Identical to AI_X962Random_V0. For forward compatibility, we recommend that you use AI_X962Random_V0.
AI_X931_Random	Generates pseudo-random numbers for RSA key generation in conformance with ANSI X9.31 standard. This AI is intended for use with AI_RSAStrongKeyGen only.
AI_X962Random_V0	SHA1 pseudo-random number generator based on X9.62 Draft

Table 3-5 **Symmetric Stream Ciphers**
Some stream ciphers include message authentication codes to detect tampering with the data stream.

Algorithm Info Type	**Description**	**BER**	**MAC**
AI_RC4	RC4		
AI_RC4_BER	RC4	3	
AI_RC4WithMAC	RC4 with message authentication code		3
AI_RC4WithMAC_BER	RC4 with message authentication code; BER-encoded algorithm identifier	3	3

Table 3-6 **Symmetric Block Ciphers**

Algorithm Info Type	**Description**	**Padding**	**BER**	**PEM**
General Purpose				
AI_FeedbackCipher	DES, Triple DES, DESX, RC2, or RC5 in ECB, CBC, CFB, or OFB feedback modes			
DES				
AI_DES_CBC_IV8	DES-CBC, 8-byte IV	none		
AI_DES_CBCPadIV8	DES-CBC, 8-byte IV	PKCS #5		

Table 3-6 **Symmetric Block Ciphers (Continued)**

Algorithm Info Type	Description	Padding	BER	PEM
AI_DES_CBCPadBER	DES-CBC, 8-byte IV, BER-encoded algorithm identifier	PKCS #5	✓	
AI_DES_CBCPadPEM	DES-CBC, 8-byte IV, PEM-encoded algorithm identifier	RFC 1423		✓
AI_DES_CBC_BSAFE1	DES-CBC, 8-byte IV, padding optional; backward compatibility with BSAFE 1.x			
Triple DES				
All 3DES algorithms in Crypto-C use the encrypt-decrypt-encrypt (EDE) sequence.				
AI_DES_EDE3_CBC_IV8	3DES-CBC			
AI_DES_EDE3_CBCPadIV8	3DES-CBC, 8-byte IV	PKCS #5		
AI_DES_EDE3_CBCPadBER	3DES-CBC, 8-byte IV, BER-encoded algorithm identifier	PKCS #5	✓	
DESX				
AI_DESX_CBC_IV8	DESX-CBC, 8-byte IV			
AI_DESX_CBCPadIV8	DESX-CBC, 8-byte IV	PKCS #5		
AI_DESX_CBCPadBER	DESX-CBC, 8-byte IV, BER-encoded algorithm identifier	PKCS #5	✓	
AI_DESX_CBC_BSAFE1	DESX-CBC, 8-byte IV, padding optional; backward compatibility with BSAFE 1.x			
RC2				
AI_RC2_CBC	RC2-CBC, 8-byte IV			
AI_RC2_CBCPad	RC2-CBC, 8-byte IV	PKCS #5		
AI_RC2_CBCPadBER	RC2-CBC, 8-byte IV, BER-encoded algorithm identifier	PKCS #5	✓	
AI_RC2_CBCPadPEM	RC2-CBC, 8-byte IV, PEM-encoded algorithm identifier	RFC 1423		✓
AI_RC2_CBC_BSAFE1	RC2-CBC, 8-byte IV, padding optional; backward compatibility with BSAFE 1.x			
RC5				
AI_RC5_CBC	RC5-CBC, 8-byte IV			
AI_RC5_CBCPad	RC5-CBC, 8-byte IV	PKCS #5		

Table 3-6 **Symmetric Block Ciphers (Continued)**

Algorithm Info Type	Description	Padding	BER	PEM
Initialization Vector				
AI_CBC_IV8	Resets the IV in a CBC algorithm during an Update or a Final for all CBC AIs except AI_FeedbackCipher			
AI_RESET_IV	Resets the IV in a CBC algorithm during an Update or a Final for all CBC implementations of AI_FeedbackCipher			
Password-Based Encryption				
These composite algorithms generate a symmetric key by digesting a password with a salt, then use the key for block cipher encryption. Not all message digests are recommended. See “Message Digests” on page 46 for details.				
AI_MD2WithDES_CBCPad	MD2 digest followed by DES-CBC	PKCS #5		
AI_MD2WithDES_CBCPadBER	MD2 digest followed by DES-CBC, BER-encoded algorithm identifier	PKCS #5	✓	
AI_MD2WithRC2_CBCPad	MD2 digest followed by RC2-CBC	PKCS #5		
AI_MD2WithRC2_CBCPadBER	MD2 digest followed by RC2-CBC, BER-encoded algorithm identifier	PKCS #5	✓	
AI_MD5WithDES_CBCPad	MD5 digest followed by DES-CBC	PKCS #5		
AI_MD5WithDES_CBCPadBER	MD5 digest followed by DES-CBC, BER-encoded algorithm identifier	PKCS #5	✓	
AI_MD5WithRC2_CBCPad	MD5 digest followed by RC2-CBC	PKCS #5		
AI_MD5WithRC2_CBCPadBER	MD5 digest followed by RC2-CBC, BER-encoded algorithm identifier	PKCS #5	✓	
AI_MD5WithXOR	MD5 digest followed by XOR for encryption	not needed		
AI_MD5WithXOR_BER	MD5 digest followed by XOR for encryption, BER-encoded algorithm identifier	not needed	✓	
AI_SHA1WithDES_CBCPad	SHA1 digest followed by DES-CBC	PKCS #5		
AI_SHA1WithDES_CBCPadBER	SHA1 digest followed by DES-CBC, BER-encoded algorithm identifier	PKCS #5	✓	

Table 3-7 **RSA Public-Key Cryptography**

Algorithm Info Type	Description	Pad	BER	PEM
Key Generation				
AI_RSAKeyGen	Key generation for RSA key pair			
AI_RSAStrongKeyGen	Key generation for RSA key pair; the generated moduli are in accordance with the strength criteria of the FIPS X9.31 standard			
Encryption and Decryption				
AI_PKCS_OAEP_RSAPrivate	RSA private-key encryption/decryption with OAEP in accordance with PKCS #1 v2	PKCS #1 v2 OAEP		
AI_PKCS_OAEP_RSAPrivateBER	RSA private-key encryption/decryption with OAEP in accordance with PKCS #1 v2, BER-encoded algorithm identifier	PKCS #1 v2 OAEP	✓	
AI_PKCS_OAEP_RSAPublic	RSA public-key encryption/decryption with OAEP in accordance with PKCS #1 v2	PKCS #1 v2 OAEP		
AI_PKCS_OAEP_RSAPublicBER	RSA public-key encryption/decryption with OAEP in accordance with PKCS #1 v2, BER-encoded algorithm identifier	PKCS #1 v2 OAEP	✓	
AI_SET_OAEP_RSAPrivate	RSA private-key encryption with OAEP in accordance with the SET v1 protocol	SET v1 OAEP		
AI_SET_OAEP_RSAPublic	RSA public-key encryption with OAEP in accordance with the SET v1 protocol	SET v1 OAEP		
AI_PKCS_RSAPrivate	RSA private-key encryption/decryption according to PKCS #1	PKCS #1 v1.5		
AI_PKCS_RSAPrivateBER	RSA private-key encryption/decryption according to PKCS #1, BER-encoded algorithm identifier	PKCS #1 v1.5	✓	
AI_PKCS_RSAPrivatePEM	RSA private-key encryption/decryption according to PKCS #1, PEM-encoded algorithm identifier	PKCS #1 v1.5		✓
AI_PKCS_RSAPublic	RSA public-key encryption/decryption according to PKCS #1	PKCS #1 v1.5		

Table 3-7 **RSA Public-Key Cryptography (Continued)**

Algorithm Info Type	Description	Pad	BER	PEM
AI_PKCS_RSAPublicBER	RSA public-key encryption/decryption according to PKCS #1, BER-encoded algorithm identifier	PKCS #1 v1.5	✓	
AI_PKCS_RSAPublicPEM	RSA public-key encryption/decryption according to PKCS #1, PEM-encoded algorithm identifier	PKCS #1 v1.5		✓
AI_RSAPrivate	Raw RSA private-key encryption; input must be a multiple of word size	none		
AI_RSAPublic	Raw RSA public-key encryption; input must be a multiple of word size	none		
AI_RSAPrivateBSAFE1	BSAFE 1.x RSA private-key encryption, padding optional			
AI_RSAPublicBSAFE1	BSAFE 1.x RSA public-key encryption			
Digital Signatures				
Composite operations for signing data: digest the data, then encrypt the BER-encoding of the digest with RSA.				
BER-encoded digest is 34 bytes for 16-bit digests (MD2, MD5); min RSA modulus is 45 bytes long; BER-encoded digest is 35 bytes for 20-bytes digests (SHA1); min RSA modules is 46 bytes long.				
AI_MD2WithRSAEncryption	MD2 digest with RSA encryption	PKCS #1		
AI_MD2WithRSAEncryptionBER	MD2 digest with RSA encryption, BER-encoded algorithm identifier	PKCS #1	✓	
AI_MD5WithRSAEncryption	MD5 digest with RSA encryption	PKCS #1		
AI_MD5WithRSAEncryptionBER	MD5 digest with RSA encryption, BER-encoded algorithm identifier	PKCS #1	✓	
AI_SHA1WithRSAEncryption	SHA1 digest with RSA encryption	PKCS #1		
AI_SHA1WithRSAEncryptionBER	SHA1 digest with RSA encryption, BER-encoded algorithm identifier	PKCS #1	✓	

Table 3-8 **DSA Public-Key Cryptography**

Algorithm Info Type	Description	BER
Parameter Generation		
AI_DSAParamGen	DSA parameter generation	
Key Generation		
AI_DSAKeyGen	DSA key generation	
Digital Signatures		
AI_DSA	DSA sign/verify a 20-byte input	
AI_DSAWithSHA1	SHA1 digest with DSA sign/verify	
AI_DSAWithSHA1_BER	SHA1 digest with DSA sign/verify, BER-encoded algorithm identifier	3

Table 3-9 **Diffie-Hellman Key Agreement**

Algorithm Info Type	Description	BER
Parameter Generation		
AI_DHParamGen	Diffie-Hellman parameter generation	
Key Agreement		
AI_DHKeyAgree	Diffie-Hellman key agreement	
AI_DHKeyAgreeBER	Diffie-Hellman key agreement, BER-encoded algorithm identifier	✓

Table 3-10 **Elliptic Curve Public-Key Cryptography**

Algorithm Info Type	Description
Parameter Generation	
AI_ECParamGen	EC parameter generation
AI_ECParameters	EC parameter use and access

Table 3-10 **Elliptic Curve Public-Key Cryptography (Continued)**

Algorithm Info Type	Description
Acceleration Tables	
AI_ECAcceleratorTable	Acceleration table use and access
AI_ECBuildAcceleratorTable	Generates auxiliary data to speed EC operations
AI_ECBuildPubKeyAccelTable	Generates auxiliary data to speed EC operations, including ECDH-specific operations
AI_ECPubKey	Generates auxiliary data to speed EC operations for a specific public-key
Key Generation	
AI_ECKeyGen	EC key pair generation
Elliptic Curve Diffie-Hellman	
AI_EC_DHKeyAgree	Two-phase EC Diffie-Hellman key agreement
Elliptic Curve DSA	
AI_EC_DSA	Raw ECDSA signature/verification
AI_EC_DSAWithDigest	SHA1 digest followed by ECDSA signature/verification
Elliptic Curve Authenticated Encryption System	
AI_EC_ES	EC Authenticated Encryption System

Table 3-11 **Bloom-Shamir Secret Sharing**

Algorithm Info Type	Description
AI_BSSecretSharing	Bloom-Shamir secret sharing

Table 3-12 **Hardware Interface**

For use with hardware devices, when present.

Algorithm Info Type	Description
AI_HW_Random	Provides access to random bytes generated by a hardware device
AI_KeypairTokenGen	Generates the token form of an RSA or DSA public/private key pair
AI_SymKeyTokenGen	Generates the token form of a DES, RC2, RC4, RC5, or TDES symmetric key
AI_PKCS_OAEPRecode	RSA raw or hardware encryption/decryption with OAEP according to PKCS #1 v2
AI_PKCS_OAEPRecodeBER	RSA raw or hardware encryption/decryption with OAEP according to PKCS #1 v2, BER-encoded algorithm identifier

Keys In Crypto-C

The key object is used to hold any key-related information and to supply this information to functions that require it. To build a key, create an empty key object with `B_CreateKeyObject`. Then, use `B_SetKeyInfo` to fill it with the information necessary to distinguish it as the desired key. That information for `B_SetKeyInfo` is made up of two items, a Key Info Type (KI) and its specific accompanying ***`info`***.

Chapter 3 of the *Crypto-C Library Reference Manual* (LRM) gives a complete listing of KIs in alphabetical order. Each entry in the *Library Reference Manual* contains a description of the information that must accompany the KI.

Summary of KIs

Table 3-13 **Generic Keys**

Key Information Type	Description
KI_8Byte	Generic 8-byte key
KI_Item	Generic variable-length key

Table 3-14 **Block Cipher Keys**

Key Information Type	Description
KI_DES8	8-byte DES key satisfying DES parity requirement
KI_DES8Strong	8-byte DES key satisfying DES parity requirement; checks for weak DES keys
KI_24Byte	24-byte 3DES key
KI_DES24Strong	24-byte 3DES key; checks for weak 3DES keys
KI_DES_BSAFE1	8-byte DES in BSAFE1.x format
KI_DESX	DESX key
KI_DESX_BSAFE1	DESX key in BSAFE 1.x format
KI_RC2_BSAFE1	RC2 key in BSAFE 1.x format
KI_RC2WithBSAFE1Params	RC2 key with additional parameters in BSAFE 1.x format

Table 3-15 **RSA Public and Private Keys**

Key Information Type	Description	BER
KI_PKCS_RSA_Private	PKCS #1-compatible RSA private key	
KI_PKCS_RSA_PrivateBER	BER encoding of an RSA private key of type PKCS #8 PrivateKeyInfo	✓
KI_RSAPrivate	RSA private key	
KI_RSAPrivateBSAFE1	RSA private key in BSAFE 1.x format	
KI_RSA_CRT	RSA key with Chinese Remainder Theorem (CRT) parameters	
KI_RSAPublic	RSA public key	
KI_RSAPublicBER	BER encoding of an RSA public key of type X.509 SubjectPublicKeyInfo	✓
KI_RSAPublicBSAFE1	RSA public key in BSAFE 1.x format	

Table 3-16 **DSA Public and Private Keys**

Key Information Type	Description	BER
KI_DSA_Private	DSA private key	
KI_DSA_PrivateBER	BER-encoding of a DSA private key of type PKCS #8	✓
KI_DSA_Public	DSA public key	
KI_DSA_PublicBER	BER-encoding of a DSA private key of type X.509 SubjectPublicKeyInfo	✓
KI_DSAPrivateX957BER	BER encoding of a DSA private key of type ANSI X9.57 PrivateKeyInfo that contains an RSA Data Security, Inc. DSAPrivateKey type	✓
KI_DSAPublicX957BER	the encoding of a DSA public key that is encoded as ANSI X9.57 SubjectPublicKeyInfo type.	✓

Table 3-17 **Elliptic Curve Keys**

Key Information Type	Description
KI_ECPrivate	EC private key and underlying EC parameters
KI_ECPrivateComponent	Private component of an EC private key
KI_ECPublic	EC public key and underlying EC parameters
KI_ECPublicComponent	Public component of an EC public key

Table 3-18 **Token Keys**
For use with hardware devices, when present.

Key Information Type	Description
KI_ExtendedToken	Software-based token form of symmetric keys
KI_KeypairToken	Software-based token forms of RSA or DSA public and private keys
KI_Token	Hardware-based token forms of symmetric and public/private keys

System Considerations In Crypto-C

Algorithm Choosers

When you use an AI, it in turn calls one or more algorithm methods. An algorithm method (or AM) is the underlying code that will actually perform the cryptography. Because many AIs can perform more than one cryptographic function (for instance, both encryption and decryption, as with `AI_FeedbackCipher`), an application will often have a choice of which underlying cryptographic code to link in. An algorithm chooser lists all the AMs the application can use. That is, it chooses in advance which AMs to link in.

Crypto-C comes with a demonstration application containing the algorithm chooser `DEMO_ALGORITHM_CHOOSER`. You can use this algorithm chooser in any Crypto-C application as long as the module that defines it (`choosc.c`) is compiled and linked in. However, `DEMO_ALGORITHM_CHOOSER` will link in all the algorithm methods available, even though an application may use only two or three. A developer can write an algorithm chooser tailored for the specific application to make the executable image smaller.

The section "Defining an Algorithm Chooser" in the *Library Reference Manual* says:

> An algorithm chooser is an array of pointers to `B_ALGORITHM_METHOD` values. The last element of the array must be `(B_ALGORITHM_METHOD *)NULL_PTR`.

From this we see that an algorithm chooser is a pointer to an array. This array contains pointers to algorithm methods, which are the AMs the application will use.

To determine which AMs to include in your algorithm chooser, you need to know which AIs you will use in your application. Then, for each AI, find the Chapter 2 entry in the *Library Reference Manual* and look at the AMs listed under "Algorithm methods to include in application's algorithm chooser." Then, based on how your application uses the given AI, decide which of those AMs you need to include in your algorithm chooser.

An Encryption Algorithm Chooser

The section "Introductory Example" on page 9 describes a program that encrypted data and did nothing else. It did not decrypt data, generate random numbers, execute the Diffie-Hellman key agreement protocols, or use elliptic curve cryptography.

Therefore, the only cryptographic tools the program needed was encryption code. And the only kind of encryption code it needed was RC4 encryption, not DES, RC2, RC5, or RSA encryption. So we could have built an algorithm chooser that included only one AM, the one we used for RC4 encryption.

To find the AM we need, look at the *Library Reference Manual*, Chapter 2, for the entry on the AI in use. We used `AI_RC4`. The *Library Reference Manual* states that for this AI, the possible AMs are `AM_RC4_ENCRYPT` for encrypting and `AM_RC4_DECRYPT` for decrypting. Because we did not decrypt, our algorithm chooser only needs to include `AM_RC4_ENCRYPT`:

```
B_ALGORITHM_METHOD *INTRODUCTORY_CHOOSER[] = {
  &AM_RC4_ENCRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};
```

The last entry of an algorithm chooser must be `(B_ALGORITHM_METHOD *)NULL_PTR.`

As an argument in a Crypto-C function call, it would look like this.

```
if ((status = B_<function> (
      <arguments>, INTRODUCTORY_CHOOSER,
      <other arguments>)) != 0)
  break;
```

An RSA Algorithm Chooser

In this example, we will build an algorithm chooser for the example in “Performing RSA Operations” on page 186. We want to include all the AMs for generating an RSA key pair, encrypting, and decrypting. We need: a random number generator, a key pair generator, an RSA public encryption algorithm, and an RSA private decryption algorithm. (Although the example doesn’t directly include a random-number generator, it calls on the one from “Generating Random Numbers” on page 147.)

The AIs used in the example are: `AI_X962Random_V0` (also known as `AI_SHA1Random`), `AI_RSAKeyGen`, `AI_PKCS_RSAPublic`, and `AI_PKCS_RSAPrivate`.

Note: `AI_SHA1Random` is identical to `AI_X962Random_V0`. The name `AI_SHA1Random` is used in the demo applications; however, `AI_SHA1Random` may change in future versions of Crypto-C. For forward compatibility, we recommend that you do *not* use the name `AI_SHA1Random` in your applications; use `AI_X962Random_V0` instead.

From the corresponding entries in Chapter 2 of the *Library Reference Manual*, you can construct the following algorithm chooser. Note that you should reference the description of `AI_X962Random_V0` instead of `AI_SHA1Random`:

```
B_ALGORITHM_METHOD *RSA_SAMPLE_CHOOSER[] = {
  &AM_SHA_RANDOM,
  &AM_RSA_KEY_GEN,
  &AM_RSA_ENCRYPT,
  &AM_RSA_CRT_DECRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};
```

Note: The above algorithm chooser lists `AM_RSA_CRT_DECRYPT`. This AM will not perform blinding (see “Timing Attacks and Blinding” on page 96). If you want your application to perform blinding, use `AM_RSA_CRT_ENCRYPT_BLIND` or `AM_RSA_CRT_DECRYPT_BLIND`.

The Surrender Context

Some Crypto-C functions are time-consuming. When an application calls one of these functions, it can appear as if the computer has crashed or frozen. A lengthy Crypto-C function can tie up the computer, forcing other applications or programs to wait until the Crypto-C function is finished to continue their execution. The surrender context is a way for an application to allow Crypto-C to surrender control.

In general, it is a good idea to include a surrender context whenever a function takes several seconds to execute. The following functions are extremely time-consuming:

- functions for parameter generation
- functions for key generation
- functions for creating acceleration tables

Other functions are less time-consuming and might not need a surrender context in your application. These include many of the block- and stream-cipher symmetric-key operations as well as message digests.

The surrender context information is contained in an `A_SURRENDER_CTX` structure. `aglobal.h` gives the definition; this is described in Chapter 1 of the *Library Reference*

Manual:

```
typedef struct {
  int (*Surrender) (POINTER);             /* surrender function callback */
  POINTER handle;                       /* application-specific information */
  POINTER reserved;                             /* reserved for future use */
} A_SURRENDER_CTX;
```

Chapter 1 also gives the form that a surrender function must have:

```
int (*Surrender) (
  POINTER handle                        /* application-specific information */
);
```

If you define a surrender function within the surrender context, Crypto-C functions will call it at regular intervals during execution. Depending on the application, the surrender function can perform one of a number of operations.

For example, a surrender function can:

- Notify the user of the current status of execution, either once at the beginning or once every second, for instance.
- Allow the user to cancel the operation.
- Suspend the Crypto-C function to allow other operations to execute.

Even when you do not need a surrender function to manage lengthy function calls, you can create one to perform other tasks. For example, you could use a surrender function to allow other applications to cut into a Crypto-C routine, no matter how quickly it executed. A surrender context can be a potent tool in debugging as well.

A Sample Surrender Function

As an example, we will construct a surrender function that announces the start of a Crypto-C function, and prints out a dot on the screen every second.

```
#include <time.h>

int GeneralSurrenderFunction (handle)
POINTER handle;
{
  static time_t currentTime;
  time_t getTime;
  if ((int)*handle == 0) {
    printf ("\nSurrender function ...\n");
    *handle = 1;
    time (&currentTime);
  }
  else {
    time (&getTime);
    if (currentTime != getTime) {
      printf " .");
      currentTime = getTime;
    }
  }
  return (0);
}
```

A routine that calls Crypto-C functions would use the above surrender function as follows:

```
A_SURRENDER_CTX generalSurrenderContext;
int generalFlag;
generalSurrenderContext.Surrender = GeneralSurrenderFunction;
generalSurrenderContext.handle = (POINTER)&generalFlag;
generalSurrenderContext.reserved = NULL_PTR;
generalFlag = 0;

if ((status = B_<function>
     (<other arguments>, &generalSurrenderContext)) != 0)
  break;
```

For this surrender function, the ***handle*** contains a flag passed from the user. If ***handle*** is 0, this is the first time the surrender function has been called by the particular Crypto-C routine currently executing. Then the surrender function will reset the flag and the next time it is called, it will skip the `if` clause and execute the `else` clause.

The coding examples in this manual use the example in this section as their surrender context. The examples also note when a routine is lengthy enough to warrant use of a surrender context. When a surrender context is not necessary, you can pass a properly cast `NULL_PTR`.

When to Allocate Memory

Whenever you use Crypto-C, you will produce output. The output might be encrypted or decrypted data, or information you are retrieving concerning keys or algorithms. This output must go somewhere; there must be memory that is allocated to hold it. If memory is not allocated for a particular output, the computer will still try to put the output somewhere, possibly in a location that already contains other data or programs. When is it the application's responsibility to allocate memory and when will Crypto-C do the allocating?

The application must allocate memory whenever a Crypto-C function produces output and the prototype indicates that the output argument is a pointer (for instance, `POINTER` or `unsigned char *`). In this situation, Crypto-C asks for a pointer and places the output at the address indicated by the pointer. It is the application's responsibility to make sure that the pointer points to allocated memory.

Crypto-C allocates memory whenever a function produces output and the prototype indicates the output argument is a pointer to a pointer (for instance, `POINTER *`). Here, Crypto-C asks for the address of a pointer. Crypto-C goes to that address and deposits a pointer there. If the application goes to where the pointer points, it will find the information it is looking for. This information, though, belongs to Crypto-C; subsequent Crypto-C calls can alter or erase it. If an application needs to save the information, it should copy it into its own buffer or allocated space. See "Distributing Diffie-Hellman Parameters" on page 222 for an example.

Note: Crypto-C will sometimes call for an `unsigned int` argument and other times an `unsigned int *`. For `unsigned int`, Crypto-C is expecting a number; for `unsigned int *`, Crypto-C will supply the number, so you just supply the address of an `int` variable.

Memory-Management Routines

Crypto-C uses the following memory-management routines:

- `T_malloc`
- `T_realloc`
- `T_free`
- `T_memset`
- `T_memcpy`
- `T_memmove`
- `T_memcmp`

Sample implementations of these routines reside in the memory management file, `tstdlib.c`, supplied with Crypto-C. See the final section of Chapter 4 in the *Library*

Reference Manual for descriptions and prototypes of these routines. You can also write your own versions of these routines to satisfy the needs of your operating system or application. It is a good idea to examine `tstdlib.c` before writing your own code.

Supplying memory management routines with Crypto-C provides several advantages:

- Reduced dependency on standard C libraries
- Increased control over memory allocation
- Increased ability to handle binary data

Memory-Management Routines and Standard C Libraries

The memory-management routines in `tstdlib.c` organize the arguments to the standard call to best suit Crypto-C's purposes. They do type checking and verify that the arguments follow the Crypto-C conventions. This helps you to keep your code portable, because any call to these routines will behave uniformly, regardless of platform. This uniform behavior best suits the needs of Crypto-C.

Some applications may need to be completely autonomous; that is, they should have no need to link in any external libraries. As far as possible, the Crypto-C library is autonomous, but Crypto-C does need the functionality of certain standard C library routines, such as `malloc`. In order for Crypto-C to remain autonomous, the user must supply these routines.

The routines in `tstdlib.c` do call the standard C library routines, so to use `tstdlib.c` you must still link in the standard C library. If your application does not need to be autonomous, you can use these supplied versions of the `T_` routines. If, however, your application will eventually require autonomy, you can supply versions of the `T_` routines that do not call the standard C library.

If a particular platform and compiler offers an optimized version or simply a platform-specific version of one or more of the memory management routines, Crypto-C can call that routine without requiring a change in the source code. You only have to modify the module containing the memory management routines.

Memory Allocation

For security reasons, it is often important that space be allocated from core memory, not a hard disk virtual memory. If an application makes a call to the standard `malloc` or `alloc`, the operating system may decide to use virtual memory. The `T_malloc` call can be made to guarantee core memory allocation and never virtual memory.

Binary Data

Remember that the C calls beginning with “`str`”, such as `strlen` and `strcpy`, operate on strings. Length is not a necessary input argument; instead, the function acts on everything from the beginning of the string to the `NULL` terminating character. However, the output from a Crypto-C call is a block of memory, not a string. Even if the data to encrypt is a string, the encrypted data is not. Similarly, data that has been decrypted will not be a properly terminated string unless the `NULL` terminating character was encrypted as well.

The “`mem`” routines supplied with Crypto-C, such as `T_memcpy` and `T_memset`, address this problem. They operate on blocks of memory and need to know how many bytes to act on. Whether or not there is a `NULL` terminating character in the block of memory does not matter.

BER/DER Encoding

Much of the data in cryptographic applications needs to be passed between two or more individuals. For example, users may need to transmit a public key, elliptic curve parameters, or an algorithm name. Not everyone uses Crypto-C, and how information is processed in Crypto-C may be different from another company’s package. There needs to be a standard for describing certain information. BER/DER is such a standard.

Open Systems Interconnection (OSI, described in ANSI’s X.200) is an internationally standardized architecture that governs the interconnection of computers from the physical layer up to the user-application layer. OSI’s method of specifying abstract objects is called ASN.1 (Abstract Syntax Notation One, defined in X.680), and one set of rules for representing such objects as strings of ones and zeros is called BER (Basic Encoding Rules, defined in X.680). There is generally more than one way to BER-encode a given value, so another set of rules, called the Distinguished Encoding Rules (DER), which is a subset of BER, gives a unique encoding to each ASN.1 value. The PKCS document includes “A Layman’s Guide to a Subset of ASN.1, BER and DER” which is more accessible than the actual standard.

If your application must transfer information to another computer or software package, you may need to convert the data into BER-encoded format before you send it. Crypto-C offers a way to get information into DER format, using `B_GetAlgorithmInfo` or `B_GetKeyInfo` with the BER version of the AI or KI used to set the algorithm or key object.

The following example corresponds to the file `berder.c`.

In the "Introductory Example" on page 9, we set the algorithm object to AI_RC4. The *Library Reference Manual* Chapter 2 entry on this AI reports that a compatible representation is AI_RC4BER. That AI provides the BER-encoded algorithm identifier for the RC4 algorithm. Look up the *Library Reference Manual* Chapter 4 entry for B_GetAlgorithmInfo. This function takes three arguments: an address for Crypto-C to deposit a pointer to the ***info***, the algorithm object from which we are getting the ***info*** and the info type.

The *Library Reference Manual* Chapter 2 entry on AI_RC4BER tells us that the *info* returned by B_GetAlgorithmInfo is a pointer to an ITEM. The type ITEM is defined in aglobal.h as:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

We will declare a variable to be a pointer to an ITEM and use its address as the ***info*** argument. The prototype calls for the address of a POINTER, not the address of a pointer to an ITEM, so type casting is necessary.

Crypto-C returns a pointer to the location where we can find the ***info***, not the ***info*** itself. When we destroy the object, that ***info*** will disappear. If we want to save it, we have to copy it over into our own buffer, the memory for which we must allocate.

```
ITEM *getInfoBER;
ITEM infoBER;

infoBER.data = NULL_PTR;

if ((status = B_GetAlgorithmInfo
     ((POINTER *)&getInfoBER, encryptionObject,
      AI_RC4_BER)) != 0)
  break;

infoBER.len = getInfoBER->len;
infoBER.data = T_malloc (infoBER.len);
if ((status = (infoBER.data == NULL_PTR)) != 0)
  break;

T_memcpy (infoBER.data, getInfoBER->data, infoBER.len);
```

Remember to use T_free to free any memory you allocated with T_malloc when you are done with it.

Now, if we need to let anyone know what algorithm we used to encrypt, we can send the BER-encoded algorithm identifier.

For additional examples that use BER, see “Distributing an RSA Public Key” on page 189 and “Distributing Diffie-Hellman Parameters” on page 222.

Note: BER-encoding does not put data into an ASCII string; it is simply a standard way of describing certain universal objects. To convert binary data to and from an ASCII string (to email it, for example) see “Converting Data Between Binary and ASCII” on page 154.

Note: Conversion into BER or DER is known as BER-encoding or DER-encoding; the conversion between binary and ASCII is known as encoding and decoding. This may get confusing, but the word encoding without a BER in front of it generally means binary to ASCII. If the encoding is BER- or DER-encoding, the BER or DER should be explicitly stated.

Input and Output

Some of the AI entries in the *Library Reference Manual* include the categories “Input Constraints” and “Output Considerations”:

- Input constraints generally describe the input requirements of the algorithm specified by the AI.
- Output considerations warn you that there may be more (or fewer) output bytes than input bytes.

Two algorithm types that typically have input constraints or output considerations are symmetric block algorithms and the RSA algorithm.

Symmetric Block Algorithms

Symmetric block algorithms may have both input constraints and output considerations.

Input constraints

- In symmetric block-encryption algorithms, the total number of input bytes must be a multiple of the block size. That does *not* mean the input to each call to an Update function must be a multiple of the block size, just the total.

For instance, with RC2, the block size is eight bytes. You can pass 23 bytes in the first call to Update, then 18, then 7, for a total of 48.

Output considerations

- For a symmetric block-encryption algorithm, the output from each Update call may be different from the input size.

  In the above example, RC2 was able to process 16 of the first 23 bytes, but saved 7 in a buffer. The input was 23, but the output was 16. During the second call to Update, Crypto-C had the 18 new input bytes plus the old 7, or 25 bytes to work with. It could process 24 (and save 1). Hence the input was 18 but the output was 24 bytes long. The last 7 input bytes combined with the saved 1 byte make up the final 8 byte block. It is important to allow for this difference in length between output and input in your application.
- In addition to the difference in size during Updates, the overall data size can differ between input and output.

  Crypto-C offers padding for the symmetric block-encryption algorithms, which have no restrictions on the total input length. Padding means that the total length of the encrypted data can be as many as eight bytes more than the total length of the input.

Note: For algorithm info types that supply padding, Crypto-C will pad even if the input is a multiple of the block size. This way, when decrypting, Crypto-C knows that the last byte is guaranteed to be a pad byte. For AIs that use PKCS #5 padding, the last byte, when decrypted, will be a number: the number of pad bytes Crypto-C should strip.

The RSA Algorithm

The second common area of input constraints is the RSA algorithm. Recall that this algorithm uses modular math.

Input constraints

The following input restrictions apply:

- Whenever modular math is used a calculation, the values passed must be less than the RSA modulus n. For example, if the modulus is 55, the input must be from zero to 54; the number 57 is invalid.

- For PKCS-compatible RSA, the input to encryption or decryption must be no more than *k* – 11 bytes long, where *k* is the modulus length in bytes. For example, with a 768-bit modulus, the input can be no more than 85, or 96 – 11, bytes. This is because the padding scheme needs at least an 11-byte area to work.

  The output will be the same size as the modulus.
- For raw RSA, the application must divide the input to encryption or decryption into blocks. Each block must have the same number of bits as the RSA modulus and, when interpreted as an integer with the most significant byte first, must be numerically less than the modulus. In addition, the size of the total input must be a multiple of the size of the modulus. That is, if the modulus is *k* bits long, each block of input must be *k* bits long, and the total input must be a multiple of *k* bits.

  For example, if the modulus is 768 bits (96 bytes) long, the input must be divided into blocks of 96 bytes and the total input must be a multiple of 96 bytes. See "Raw RSA" on page 197 for more information on how to pass data properly.

  The output of raw RSA is the same size as the input.

  In general, there should be no need for raw RSA encryption or decryption. We do not recommend using raw RSA unless you are familiar with the issues involved.

General Considerations

In general, Crypto-C has mechanisms to keep you aware of input constraints and output considerations.

If your input does not meet the constraints, Crypto-C will return an error message.

For output, Crypto-C requires that you pass the size of the output buffer. In this way, Crypto-C will determine whether there is enough space available before trying to store output. If your buffer is not big enough, Crypto-C will return an error.

Most important of all, when it comes to output, Crypto-C tells you how many bytes it placed into the output buffer. That argument is `unsigned int *partOutLen` in the Update and Final function prototypes. Pass an address to an `unsigned int` and Crypto-C will go to that address and drop a value there. That value is the number of bytes Crypto-C placed into the output buffer. After the call to Crypto-C, you can look at that value to determine how many bytes were processed. It may not be the same number as the input length. It might be more, it might be less. It may even be zero.

Key Size

In cryptography, security is measured in key size: the bigger the key, the greater the security. Key size, in turn, is measured in bits. However, a bit number does not necessarily describe the entire key.

DES Keys

A DES key is 56 bits. However, that size refers to its cryptographic size, not its physical size. To build a DES key, you need 64 bits, but because eight of those bits are “parity bits,” which are known, you really only get 56 secret bits. Hence, a DES key, while consisting of 64 bits of data, is only 56 cryptographic bits large.

RSA Keys

An RSA key-pair measurement describes the modulus length. When cryptographers talk about a “768-bit RSA key pair,” what they really mean is that the modulus is 768 bits long. Because the security of an RSA key pair depends on how big the modulus is, the measurement used is the bit-size of the modulus. However, the actual keys themselves contain more information than the modulus, so the physical size is much larger.

Public Key Size

A public key consists of a modulus and a public exponent. To store that public key requires space for both of those components; so for a 768-bit public key, you need more than 768 bits of storage space.

Almost everyone who uses the RSA algorithm uses F_4 as the public exponent. F_4 is short for Fermat 4, one of a sequence of numbers with special properties first described by the 17th-century mathematician Pierre de Fermat. F_4 = 01 00 01 in hexadecimal notation (65,537 in decimal), and it is 17 bits long. If you use F_4, you need 785 bits of space to store a 768-bit public key and its public exponent. Of course, storage space comes only in bytes, so you actually need 99 bytes of space.

In addition, when you access the public key, you need to know where the modulus ends and the public exponent begins. It would be a good idea to put identifying marks on the data to make it easier to parse. BER/DER encoding standardizes such identifying marks as an industry standard so that people using different software packages can still trade information. Hence, with Crypto-C, the user has the option of storing a 768-bit public key simply as a modulus and public exponent (99 bytes), or in its DER encoded format, which requires 126 bytes.

Private Key Size

At its most basic form, the private key consists of a modulus and a private exponent.

The modulus for the private key is the same as the modulus for the public key. The private exponent is the truly private part of the private key. The private value is usually the same size as the modulus, or 1 bit smaller. Therefore, to store a 768-bit private key, one needs at least 1536 bits (192 bytes) of storage space.

To perform private key operations, you require only the modulus and private exponent. However, the computations can be much faster if you have access to more information.

Recall that, in RSA, the modulus is actually the product of two prime numbers. The private exponent is derived from the two primes and the public exponent. Given only the modulus and the public exponent, an attacker cannot deduce the private exponent.

When computing the key pair, you can find two suitable primes, multiply them together to get the modulus, use the primes to determine the private exponent, and then throw the primes away. Or you can use the primes to compute two prime exponents and a Chinese Remainder Theorem (CRT) coefficient, and save all this information. Then, when executing private key operations with the extra information, you can use the Chinese Remainder Theorem to make the appropriate computations much more quickly.

So when saving a 768-bit private key, you actually need to save the following:

- the modulus: 96 bytes
- the public exponent — it is small and there are advantages to having it saved with the private key: 3 bytes
- the private exponent: 96 bytes
- two primes: 2 × 48 bytes
- two prime exponents: 2 × 48 bytes
- a CRT coefficient: 48 bytes.
- The identifying marks for DER encoding

This adds up to 484 bytes!

In addition, when the most significant bit of the most significant byte of a value is set, DER calls for a prepended 0 byte, so that it is not interpreted as a negative 2’s complement number.

For example, converting the decimal number 3,260,571,825 into hex yields 0xC25860B1. As a byte string, it would be:

```
C2 58 60 B1
```

which is four bytes long. But is that a negative or positive number? Is the sign bit set, or is this an unsigned value? To avoid confusion, we prepend a 0 byte, as follows:

```
00 C2 58 60 B1
```

Our string is now five bytes long.

For a 768-bit key pair, the most significant bit of the most significant byte of the modulus *and* both primes should always be set. So three of the private key's values will have a prepended 0 byte. This increases the total key size to 487 bytes. Sometimes the most significant bit of the most significant byte of the private exponent, prime exponents and CRT coefficient will be set, sometimes not. So the total bytes could be as many as 491.

Note: If the public exponent is F_4 (01 00 01), that value does not need a prepended 0 byte.

All of this means that when you generate your RSA key pair, you do not know in advance how big it is going to be when you store it in DER format. You know the approximate size, but not the exact length.

Crypto-C has the tools to let you know the exact length of your encoded key. When you call `B_GetKeyInfo`, you pass the address of a pointer. Crypto-C drops off a pointer at that address. If you go to the address indicated by the pointer, you will find the key information, which includes the key's length. Use that value to find out exactly how long your key is.

Using Cryptographic Hardware

Interfacing with a BHAPI Implementation

Crypto-C lets you enhance the security and speed of cryptographic operations by exploiting cryptographic hardware that supplies an interface to Crypto-C via the BSAFE Hardware Application Programming Interface (BHAPI). Capabilities include a hardware algorithm method for random number generation and key token types that encapsulate RSA, DSA, and symmetric keys inside of hardware.

When you Create, Set, and Init an algorithm object in a Crypto-C software application, you set an algorithm info type (AI) and the parameters required by the AI. You also choose which algorithm methods to use via the software chooser. The AI itself doesn't perform any cryptographic operations; rather, it is used to store information, allocate space, and to create the necessary points of contact with the underlying Crypto-C functions. Figure 3-1 shows the relation between the algorithm object and the Crypto-C software library.

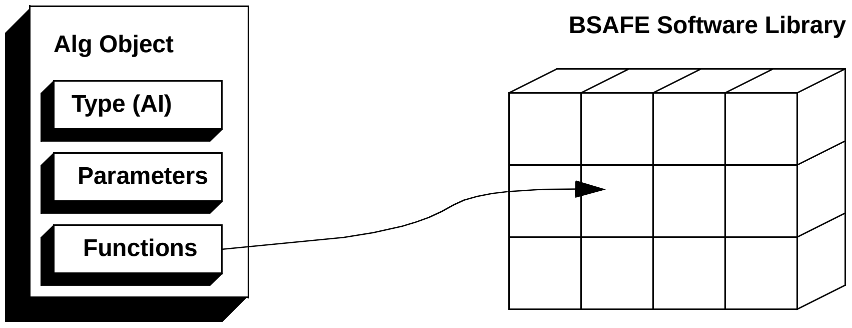

Figure 3-1 **Algorithm Object in a Software Implementation**

A hardware manufacturer can associate a hardware function with a Crypto-C AM and provide these methods to the software developer. You then access the hardware by using `B_CreateSessionChooser` to create a hardware-based chooser, e.g., `FIXED_HARDWARE_CHOOSER`, that lists the available required hardware methods. This substitution is made at link time, and does not change once the application has been compiled.

If more than one hardware method is present for the same AM — for example, if the application includes hardware methods implementing RSA encryption from two different manufacturers — `B_CreateSessionChooser` includes all available hardware methods. When an object's methods are instantiated at initialization, Crypto-C loads the object with the first compatible method from the session chooser. Figure 3-2 shows how an algorithm object operates with a hardware interface.

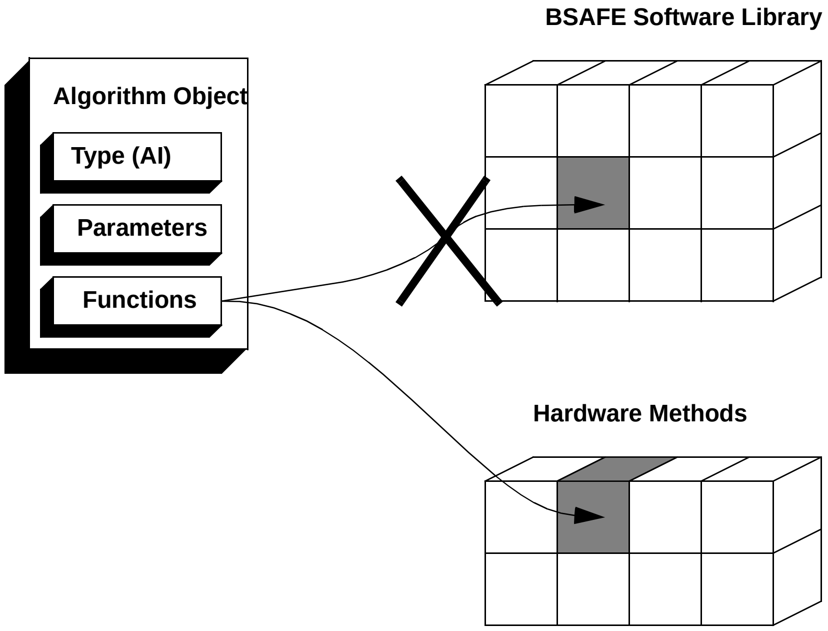

Figure 3-2 **Algorithm Object with Hardware**

During the call to `B_CreateSessionChooser`, Crypto-C tests for the presence of the hardware; if hardware is present, the hardware method is included in the session chooser. If no hardware is present, then the application defaults to the Crypto-C software AM or to a software emulation if one is included in the chooser.

To extend the functionality of the BHAPI interface to include key-token operations, Crypto-C supplies some AIs that are only available when `B_CreateSessionchooser` is used. These AIs have software-emulated versions, but can only be accessed via inclusion in the hardware chooser.

Hardware Issues

Working with hardware devices introduces new issues that must be addressed. A cryptographic key on a hardware device might never leave the device; this is part of the security. For instance, suppose you want to produce a digital envelope. You might use an hardware accelerator to perform DES encryption of the bulk data, then want to encrypt the DES key with the recipient's public key. However, when you make the call to retrieve the key, the hardware might return a handle to the key, rather than the key itself. This enhances security, because the key never appears "in public."

To implement this, the hardware accelerator might require you to call its key-wrapping routines to build a digital envelope. When you request the key in order to store it for later use, the hardware could return a handle to the key. But if you give that data to another cryptographic package, the key will mean nothing.

In other words, once you build a key (symmetric or private) on a hardware device, it is possible that only that hardware device will be able to use the key. Therefore, you should use hardware accelerators only when you thoroughly understand their use.

Chapter 4

Non-Cryptographic Operations

Crypto-C supplies a number of non-cryptographic algorithms that are necessary for cryptographic applications. These include:

- Message digests
- Random number generators
- ASCII-to-binary and binary-to-ASCII encoding

Message Digests

A message digest is a fixed-length statistically-unique identifier that corresponds to a set of data. That is, each unit of data — such as a file, string, or buffer — maps to a particular byte sequence (usually 16 or 20 bytes long). A digest is not random: digesting the same unit of data with the same message-digest algorithm will always produce the same byte sequence.

Digests are used in random-number generation, password-based encryption, and digital signatures.

Creating a Digest

The example in this section corresponds to the file `mdigest.c`.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ digester = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&digester)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

Crypto-C offers four message digest algorithms: MD, MD2, MD5, and SHA1.

Note: Recent cryptanalytic work has discovered a collision in MD2's internal compression function, and there is some chance that the attack on MD2 may be extended to the full hash function. The same attack applies to MD. Another attack has been applied to the compression function on MD5, though this has yet to be extended to the full MD5. RSA Data Security, Inc., recommends that before you use MD, MD2, or MD5, you should consult the RSA Laboratories web site to be sure that their use is consistent with the latest information.

The AI for SHA1 is `AI_SHA1`; the *Library Reference Manual* Chapter 2 entry for this AI states that the format of *info* supplied to B_SetAlgorithmInfo is NULL_PTR:

```
if ((status = B_SetAlgorithmInfo
     (digester, AI_SHA1, NULL_PTR)) != 0)
  break;
```

Step 3: Init

To initialize a message digest, call B_DigestInit. The *Library Reference Manual* Chapter 4 entry on B_DigestInit shows that it requires four arguments. The first argument is the algorithm object. The second is a key object. All Crypto-C message digest AIs call for a properly cast NULL_PTR as the key object; Crypto-C provides this argument for algorithms, like HMAC, that require keys. The third argument is an algorithm chooser. The fourth is a surrender context; this is a fast function, so it is reasonable to pass a properly cast NULL_PTR:

```
B_ALGORITHM_METHOD *DIGEST_CHOOSER[] = {
  &AM_SHA,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_DigestInit
     (digester, (B_KEY_OBJ)NULL_PTR, DIGEST_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Use B_DigestUpdate to enter the data to digest. If you have separate units of data (for example, two or more files or several strings), make a call to B_DigestUpdate for each unit. Message digesting is quick, so unless you are digesting an extremely large amount of data (a megabyte or more), it is reasonable to pass a properly cast NULL_PTR for the surrender context.

Your call will be the following:

```
/* The variable dataToDigest should already point to allocated
   memory and contain the data, dataToDigestLen should
   already be set to the number of bytes to digest. */

unsigned char *dataToDigest;
unsigned int dataToDigestLen;

if ((status = B_DigestUpdate
     (digester, dataToDigest, dataToDigestLen,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

An MD2 or MD5 digest is always 16 bytes; an SHA1 digest is always 20 bytes. Because you are using SHA1, create a 20-byte buffer to hold the output and call `B_DigestFinal`. The *Library Reference Manual* gives the prototype for this function in Chapter 4.

The first argument is the algorithm object. The second is the buffer where Crypto-C will deposit the digest. The third is an address for Crypto-C to return the number of bytes in the digest. Because this value should always be 20, you can use this as a check on the algorithm if you like. The fourth argument is the size of the output buffer. If Crypto-C needs a bigger buffer, this function will return an error. The fifth argument is the surrender context; this is a fast function, so there should be no problem with using a properly cast `NULL_PTR`:

```
#define DIGEST_LEN 20

unsigned char digestedData[DIGEST_LEN];
unsigned int digestedDataLen;

if ((status = B_DigestFinal
     (digester, digestedData, &digestedDataLen, DIGEST_LEN,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects when you are done with them:

```
B_DestroyAlgorithmObject (&digester);
```

BER-Encoding the Digest

If you want to send your digest to someone, you should BER-encode the algorithm identifier and the digest. The Crypto-C function `B_EncodeDigestInfo` offers a way to put together a string containing your information in BER format.

The example in this section corresponds to the file `mdber.c`.

The *Library Reference Manual* Chapter 4 entry for `B_EncodeDigestInfo` shows that this function takes six arguments:

```
int B_EncodeDigestInfo (
  unsigned char *digestInfo,                       /* encoded output buffer */
  unsigned int  *digestInfoLen,                 /* length of encoded output */
  unsigned int   maxDigestInfoLen,             /* size of digestInfo buffer */
  ITEM          *algorithmID,       /* message digest algorithm identifier */
  unsigned char *digest,                             /* message digest value */
  unsigned int   digestLen                               /* length of digest */
);
```

The first argument is an address where Crypto-C can drop the BER-encoded digest information. You will have to allocate the space for this buffer. This buffer will contain the algorithm identifier and the 16- or 20-byte digest, the total for MD2 and MD5 digests is 34; for a SHA1 digest, it is 35 bytes. If you want to be safe, you can make the buffer larger.

The second argument is the address of an `unsigned int`; Crypto-C will place the final length of the BER encoding at that address. The third argument is the buffer size. The fourth is a pointer to an `ITEM` containing the DER encoding of the message digest algorithm; you obtain the DER encoding by calling `B_GetAlgorithmInfo` with the appropriate AI with BER-encoding. The fifth argument is the digest itself; the sixth is the length of the digest.

The following example BER-encodes the sample digest above:

```
#define DIGEST_LEN 20
#define ALG_ID_LEN DIGEST_LEN + 18

ITEM *sha1AlgInfoBER;
unsigned char digestInfoBER[ALG_ID_LEN];
unsigned int digestInfoBERLen;

if ((status = B_GetAlgorithmInfo
     ((POINTER *)&sha1AlgInfoBER, digester, AI_SHA1_BER)) != 0)
  break;

if ((status = B_EncodeDigestInfo
     (digestInfoBER, &digestInfoBERLen, ALG_ID_LEN, sha1AlgInfoBER,
      digestedData, digestedDataLen)) != 0)
  break;
```

To decode BER-encoded information, call `B_DecodeDigestInfo`. Simply pass the addresses you need; Crypto-C will fill the `ITEM`s for you:

```
ITEM retrievedAlgorithmID;
ITEM retrievedDigest;

if ((status = B_DecodeDigestInfo
     (&retrievedAlgorithmID, &retrievedDigest, digestInfoBER,
      digestInfoBERLen)) != 0)
  break;
```

Hash-Based Message Authentication Code (HMAC)

A hash-based message authentication code (HMAC) combines a secret key with a message digest to create a message authentication code. See "Hash-Based Message Authentication Codes (HMAC)" on page 47 for a description of the algorithm.

Crypto-C provides an HMAC implementation based on SHA1. Recall that SHA1 produces a 20-byte digest and takes input in 64-byte blocks.

The example in this section corresponds to the file `hmac.c`.

Step 1: Creating an Algorithm Object

Declare a variable of type `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ HMACDigester = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&HMACDigester)) != 0)
  break;
```

Step 2: Setting the Algorithm Object

There is only one AI for hash-based message authentication codes, `AI_HMAC`. The *Library Reference Manual* Chapter 2 entry for `AI_HMAC` states that the format of *`info`* supplied to `B_SetAlgorithmInfo` is a pointer to a `B_DIGEST_SPECIFIER` structure:

```
typedef struct {
  B_INFO_TYPE digestInfoType;
  POINTER     digestInfoParams;
} B_DIGEST_SPECIFIER;
```

The only choice for *`digestInfoType`* in Crypto-C is `AI_SHA1`. In the case of `AI_SHA1`, *digestInfoParams* should be set to NULL_PTR:

```
  B_DIGEST_SPECIFIER hmacInfo;

  hmacInfo.digestInfoType = AI_SHA1;
  hmacInfo.digestInfoParams = NULL_PTR;

  if ((status = B_SetAlgorithmInfo
       (HMACDigester, AI_HMAC, (POINTER)&hmacInfo)) != 0)
    break;
```

Step 3: Init

For hash-based message authentication, you need a key before you can initialize the object.

Step 3a: Creating the Key Object

```
#define KEY_SIZE 24

  B_KEY_OBJ HMACKey = (B_KEY_OBJ)NULL_PTR;
  unsigned char *keyData;

  /* Create a key object */
  if ((status = B_CreateKeyObject (&HMACKey)) != 0)
    break;
```

Step 3b: Setting the Key Object

Generate a random 24-byte key using KI_24Byte:

```
  keyData = T_malloc (KEY_SIZE);
  if ((status = (keyData == NULL_PTR)) != 0)
    break;

  /* Complete Steps 1-4 of Generating Random Numbers  */
  /* Generate KEY_SIZE bytes of random data for the key. */
  if ((status = B_GenerateRandomBytes
       (randomAlgorithm, keyData, KEY_SIZE,
       (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
```

```
/* Set the key object */
if ((status = B_SetKeyInfo (HMACKey, KI_24Byte, keyData)) != 0)
  break;
```

Once you have a properly initialized the key object, you can call `B_DigestInit`. The *Library Reference Manual* Chapter 4 entry on `B_DigestInit` shows that it requires four arguments. The first argument is the algorithm object; the second is the key object. The third is an algorithm chooser. The fourth is a surrender context; this is a fast function, so it is reasonable to pass a properly cast `NULL_PTR`:

```
B_ALGORITHM_METHOD *HMAC_CHOOSER[] = {
  &AM_SHA,
  &AM_SHA_RANDOM,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_DigestInit
     (HMACDigester, HMACKey, HMAC_CHOOSER,
     (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Once you have set the algorithm object, you can create the message authentication code by calling `B_DigestUpdate` for all of the data to digest:

```
unsigned char dataToDigest[] = "Digest this sentence.";
unsigned int dataToDigestLen = strlen (dataToDigest);

if ((status = B_DigestUpdate
     (HMACDigester, dataToDigest, dataToDigestLen,
     (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

After the data to digest has been processed by calls to `B_DigestUpdate`, call `B_DigestFinal`. You need to pass a pointer to the location where `B_DigestFinal` can store the output. In the case of `AI_HMAC` using SHA1, you need 20 bytes to store the result.

```
unsigned char *digestedData;
unsigned int digestedDataLen;

digestedData = T_malloc (20);
if ((status = (digestedData == NULL_PTR)) != 0)
  break;

if ((status = B_DigestFinal
     (HMACDigester, digestedData, &digestedDataLen,
         20, (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Once you have generated the message authentication code, destroy any objects you used, and free up any memory you allocated:

```
B_DestroyKeyObject (&HMACKey);
B_DestroyAlgorithmObject (&randomAlgorithm);
B_DestroyAlgorithmObject (&HMACDigester);

if (digestedData != NULL_PTR) {
  T_memset (digestedData, 0, 20);
  T_free (digestedData);
  digestedData = NULL_PTR;
  digestedDataLen = 0;
}
```

Generating Random Numbers

In the “Introductory Example” on page 9, we hard-coded the DES key. In an actual application, you would use randomly-generated values. Crypto-C allows you to generate a pseudo-random sequence of bytes using a pseudo-random number generator (PRNG). These PRNGs are based on the message digests MD2, MD5, and SHA1. In fact, because different standards implement random number generation in different ways, there are two random number generators based on SHA1:

- `AI_X962Random_V0` is a SHA1-based pseudo-random number generator. Its implementation can also be used as a model for the MD2 and MD5 random number generators. This model should be used for most random-number generation methods.
- `AI_X931Random` is a SHA1-based pseudo-random number generator that allows multiple streams of randomness. It is intended primarily for use with `AI_RSAStrongKeyGen`, and should not be used for general-purpose random-number generation.

Because there are differences between these two PRNGs, this section shows how to generate random numbers for the two SHA1 implementations.

Generating Random Numbers with SHA1

The example in this section corresponds to the file `genbytes.c`. This example, which uses `AI_X962Random_V0`, can easily be modified to use the PRNGs based on MD2 and MD5, `AI_MD2Random` and `AI_MD5Random`, respectively.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ randomAlgorithm = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&randomAlgorithm)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

You need to supply an appropriate algorithm info type (AI) and the proper associated *info* to B_SetAlgorithmInfo. For random-number generation, you have a choice between AI_MD2Random, AI_MD5Random, AI_X962Random_V0 (also known as AI_SHA1Random), and AI_X931Random, based on the message digest algorithms MD2, MD5, and SHA1 described earlier. For this example, choose AI_X962Random_V0.

Note: AI_SHA1Random is identical to AI_X962Random_V0; the name AI_SHA1Random is used in the demo applications. However, AI_SHA1Random may change in future versions of Crypto-C. For forward compatibility, we recommend that you do *not* use the name AI_SHA1Random in your applications; use AI_X962Random_V0 instead.

Note: Recent cryptanalytic work has discovered a collision in MD2's internal compression function, and there is some chance that the attack on MD2 may be extended to the full hash function. The same attack applies to MD. Another attack has been applied to the compression function on MD5, though this has yet to be extended to the full MD5. RSA Data Security, Inc. recommends that before you use MD, MD2, or MD5, you should consult the RSA Laboratories web site to be sure that their use is consistent with the latest information.

The entry for AI_SHA1Random in Chapter 2 of the *Library Reference Manual* refers you to AI_X962Random_V0; the entry for this second AI states that the *info* supplied to B_SetAlgorithmInfo is NULL_PTR. So the proper way to set your random algorithm object is:

```
if ((status = B_SetAlgorithmInfo
     (randomAlgorithm, AI_SHA1Random, NULL_PTR)) != 0)
  break;
```

Step 3: Init

Initialize *randomAlgorithm* with B_RandomInit. The prototype of this function in Chapter 4 of the *Library Reference Manual* indicates that it takes three arguments: the algorithm object, the algorithm chooser, and the surrender context. The first argument is randomAlgorithm. For the second argument, build an algorithm chooser that contains the AMs listed in the *Library Reference Manual* Chapter 2 entry for AI_X962Random_V0. B_RandomInit is a fast function, so it is reasonable to use a properly cast NULL_PTR for the surrender context as the third argument.

```
B_ALGORITHM_METHOD *RANDOM_CHOOSER[] = {
  &AM_SHA1_RANDOM,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_RandomInit
      (randomAlgorithm, RANDOM_CHOOSER,
       (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

The B_RandomUpdate function mixes in a random seed to the algorithm object. The function prototype in Chapter 4 of the *Library Reference Manual* shows that B_RandomUpdate takes four arguments: an algorithm object, a random seed, the length of the random seed, and a surrender context. So before you can call B_RandomUpdate, you need to procure a random seed.

Step 4a: The Random Seed

The purpose of random number generation is to produce an unpredictable and unrepeatable sequence of bytes. If you do not update a random algorithm object with a random seed, you will generate a default sequence of pseudo-random bytes. In addition, if someone else updates their random algorithm object with the same seed that you used, they will generate the same sequence you did. Because unrepeatability depends on the random seed, you want an unrepeatable seed.

Generating a seed that cannot be predicted or repeated is especially important in cryptography. There are a number of sources for unrepeatable seeds. The best source may be a hardware noise generator. The BSAFE Hardware API (BHAPI) offers a way to interface with a hardware random generator. One such implementation interfaces with Intel's Random Number Generator; see the *RSA BSAFE Crypto-C Intel Security Hardware User's Manual* for more information. Other seed-gathering methods involve tracking mouse movement or timing keystrokes, system time, or processor-elapsed time. There may be other schemes you can devise that do not depend on someone entering a value from the keyboard.

The seed does not necessarily have to be random, but it must be input which is difficult to predict or reproduce. Once you have seeded the random algorithm, the algorithm can produce a sequence of random bytes; these bytes are "more random" and are generated more quickly than the seed. See "Pseudo-Random Numbers and

Seed Generation" on page 92 for more information.

Before you get your seed, you need to set aside memory to hold it. In this example, you will allocate 256 bytes for your seed:

```
POINTER randomSeed = NULL_PTR;
unsigned int randomSeedLen;

randomSeedLen = 256;
randomSeed = T_malloc (randomSeedLen);
if ((status = (randomSeed == NULL_PTR)) != 0)
  break;
```

Now get the random seed. The exact method you use to get the seed will depend on your application and how the seed is generated. Here is a quick method for getting keyboard input. This method is not recommended for an actual application; it is supplied for illustrative purposes only:

```
puts ("Enter a random seed");
if ((status =
     (NULL_PTR ==
      (unsigned char *)gets ((char *)randomSeed))) != 0)
  break;
```

Note: Another method for acquiring a seed would be to use a hardware random number generator, if available, such as the Intel Random Number Generator described in the *Crypto-C Intel Security Hardware User's Guide*. However, even if you have access to random numbers from hardware, you will still want to have a fallback method of seed collection, in case the hardware random number generator is not available or fails for some reason.

Here you are using a 256-byte buffer. When the space was allocated, the contents of the buffer were simply whatever happened to be in that memory location at the time. In this case, when you enter a seed at the keyboard (the `gets` function), you overwrite the first few bytes in the buffer, one byte for each keystroke. Now, the first bytes in the buffer are the input from the keyboard; the rest of the 256 bytes are untouched.

Note: If you want to guarantee a repeatable seed (for example, if you are testing and want to be able to reproduce your data), set the buffer with `T_memset`.

Now that you have a random seed, you can call `B_RandomUpdate`. The length argument tells Crypto-C how many bytes from the random seed buffer to use. See "Pseudo-Random Numbers and Seed Generation" on page 92 for a discussion on how many seed bytes to use. In this example, you will use all 256 bytes from the buffer, even though you probably entered fewer than 256 characters at the keyboard. Once again, it is reasonable to pass a NULL_PTR for the surrender context, because B_RandomUpdate is a fast function:

```
if ((status = B_RandomUpdate
     (randomAlgorithm, randomSeed, randomSeedLen,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Call B_RandomUpdate as many times as you wish with different seeds each time to increase the unrepeatability of your random number generator. After each Update, you may want to overwrite and free your seed immediately.

Step 5: Generate

When generating random bytes, you call B_GenerateRandomBytes instead of a Final function. The function prototype in Chapter 4 of the *Library Reference Manual* calls for the following arguments: a random algorithm object, an output buffer, the number of bytes to generate, and a surrender context. You need to prepare a buffer before calling B_GenerateRandomBytes:

```
#define NUMBER_OF_RANDOM_BYTES 64

unsigned char *randomByteBuffer = NULL_PTR;

randomByteBuffer = T_malloc (NUMBER_OF_RANDOM_BYTES);
if ((status = (randomByteBuffer == NULL_PTR)) != 0)
  break;
```

Now you can generate some random bytes. Generating 64 bytes is quick, so you are still safe in using a NULL_PTR for the surrender context.

```
if ((status = B_GenerateRandomBytes
     (randomAlgorithm, randomByteBuffer, NUMBER_OF_RANDOM_BYTES,
     (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects when done with them. You must also call `T_free` once for each call to `T_malloc`. For security reasons, overwrite the seed buffer with zeros as well:

```
B_DestroyAlgorithmObject (&randomAlgorithm);
T_memset (randomSeed, 0, randomSeedLen);
T_free (randomSeed);
T_free (randomByteBuffer);
```

Generating Independent Streams of Randomness

`AI_X931Random` is a SHA1-based pseudo-random number generator that allows you to generated multiple streams of randomness. This means that the Crypto-C implementation of the X9.31 random algorithm is somewhat different from the implementation of the other PRNGs in Crypto-C. This section describes the modifications you would have to make to the previous example to use `AI_X931Random`. These modifications take place at Step 2, Set, and Step 3, Init.

The example in this section corresponds to the file `x931rand.c`.

Step 1: Create

This step is identical to the previous example.

Step 2: Set

Setting the X9.31 random algorithm object is the main difference working with the other random algorithms. `AI_X931Random` requires you to pass in a structure describing the number of independent streams of randomness and a seed which will be divided between the streams.

```
typedef struct
{
     unsigned int numberOfStreams;        /* number of independent streams */
     ITEM         seed;                              /* additional seeding */
                                 /* to be equally divided among the streams */
} A_X931_RANDOM_PARAMS;
```

For this example, you will specify six streams of randomness, and provide a seed stored in an `ITEM` structure, ***randomSeed***. The amount of seed data passed in the `A_X931_RANDOM_PARAMS` structure must greater than or equal to 20 * (number of streams) bytes and less than or equal to 64 * (number of streams) bytes. With six streams, this means the seed size must be between 120 bytes and 384 bytes. If the amount of seed data is outside this range, Crypto-C will return a `BE_ALGORITHM_INFO` error.

In addition, Crypto-C checks the seed value for the amount of entropy. For example, a constant seed (all zeros or all ones) will return an error.

```
ITEM randomSeed;
A_X931_RANDOM_PARAMS x931Params;

x931Params.numberOfStreams = 6;
x931Params.seed.data = randomSeed.data;
x931Params.seed.len = randomSeed.len;

if ((status = B_SetAlgorithmInfo
     (randomAlgorithm, AI_X931Random, (POINTER)&x931Params)) != 0)
   break;
```

Step 3: Init

Once the structure has been passed in, the Init is essentially the same as in the previous example. The only difference is that `AM_X931_RANDOM` appears in the chooser.

```
B_ALGORITHM_METHOD *RANDOM_CHOOSER[] = {
  &AM_X931_RANDOM,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_RandomInit
     (randomAlgorithm, RANDOM_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Steps 4, 5, 6

These steps are identical to the previous example.

Converting Data Between Binary and ASCII

If you have data in binary format, yet need it in ASCII, or vice versa, Crypto-C offers functions to encode and decode according to the RFC1113 standard.

The example in this section corresponds to the file `encdec.c`.

Encoding Binary Data To ASCII

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ asciiEncoder = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&asciiEncoder)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There is only one ASCII encoding or decoding AI, `AI_RFC1113Recode`. The *Library Reference Manual* Chapter 2 entry for this AI states that the format of *`info`* supplied to `B_SetAlgorithmInfo` is `NULL_PTR`:

```
if ((status = B_SetAlgorithmInfo
      (asciiEncoder, AI_RFC1113Recode, NULL_PTR)) != 0)
  break;
```

Step 3: Init

To initialize ASCII encoding, call `B_EncodeInit`. This function takes only one argument, the algorithm object:

```
if ((status = B_EncodeInit (asciiEncoder)) != 0)
  break;
```

Step 4: Update

Enter the data to encode through `B_EncodeUpdate`. The application is responsible for allocating the space for the output of this routine. When encoding, for each three bytes of input there are four bytes of output. So when allocating space, multiply the input size by 4/3 and round up. If memory is not an issue, you can make the output buffer twice the size of the input length.

Given pre-existing binary input, your calls to the Update functions would be as follows:

```
/* We are assuming binaryData already points to allocated
     space and contains the data to encode into ASCII.
 */
unsigned char *binaryData;
unsigned int binaryDataLen;
unsigned char *asciiEncoding = NULL_PTR;
unsigned int asciiEncodingLenUpdate;

/* Allocate a buffer twice the size of the binary data */
asciiEncoding = T_malloc (binaryDataLen * 2);
if ((status = (asciiEncoding == NULL_PTR)) != 0)
  break;

if ((status = B_EncodeUpdate
     (asciiEncoder, asciiEncoding, &asciiEncodingLenUpdate,
      (binaryDataLen * 2), binaryData, binaryDataLen)) != 0)
  break;
```

Step 5: Final

Finalize the encoding process, writing out any remaining bytes.

```
unsigned int asciiEncodingLenFinal;

if ((status = B_EncodeFinal
     (asciiEncoder, asciiEncoding + asciiEncodingLenUpdate,
      &asciiEncodingLenFinal,
      (binaryDataLen * 2) - asciiEncodingLenUpdate)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects and free up any memory allocated when done:

```
B_DestroyAlgorithmObject (&asciiEncoder);
T_free (asciiEncoding);
```

Decoding ASCII-Encoded Data

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ asciiDecoder = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&asciiDecoder)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There is only one ASCII-encoding or decoding AI, `AI_RFC1113Recode`. The *Library Reference Manual* Chapter 2 entry on this AI states that the format of *`info`* supplied to

`B_SetAlgorithmInfo` is `NULL_PTR`:

```
if ((status = B_SetAlgorithmInfo
     (asciiDecoder, AI_RFC1113Recode, NULL_PTR)) != 0)
  break;
```

Step 3: Init

To initialize decoding, call `B_DecodeInit`. This function takes only one argument, the algorithm object:

```
if ((status = B_DecodeInit (asciiDecoder)) != 0)
  break;
```

Step 4: Update

Enter the data to decode through `B_DecodeUpdate`. The application is responsible for allocating the space for the output of this routine. When decoding, there will be three bytes of output for every four bytes of input. If memory is a concern, you may want to determine the exact number of bytes you will need. If memory is not a concern, make the output size equal to the input length.

Given your pre-existing ASCII input, your call to the Update function would be as follows:

```
/* We are assuming asciiEncoding already points to allocated
     space and contains the data to decode into binary. Also,
     asciiEncodingLenTotal is already set with the length of
     the asciiEncoding.
 */
unsigned char *asciiEncoding;
unsigned int asciiEncodingLenTotal;
unsigned char *binaryDecoding = NULL_PTR;
unsigned int binaryDecodingLenUpdate;

/* Allocate a buffer the same size as the ascii data. */
binaryDecoding = T_malloc (asciiEncodingLenTotal);
if ((status = (binaryDecoding == NULL_PTR)) != 0)
  break;
```

```
if ((status = B_DecodeUpdate
     (asciiDecoder, binaryDecoding, &binaryDecodingLenUpdate,
      asciiEncodingLenTotal, asciiEncoding,
      asciiEncodingLenTotal)) != 0)
  break;
```

Step 5: Final

Finalize the decoding process, writing out any bytes remaining:

```
unsigned int binaryDecodingLenFinal;

if ((status = B_DecodeFinal
     (asciiDecoder, binaryDecoding + binaryDecodingLenUpdate,
      &binaryDecodingLenFinal,
      asciiEncodingLenTotal - binaryDecodingLenUpdate)) != 0)
  break;
```

Step 6: Destroy

When you are done, remember to destroy all objects and free up any memory that has been allocated:

```
B_DestroyAlgorithmObject (&asciiDecoder);
T_free (binaryDecoding);
```

Chapter 5

Symmetric-Key Operations

Recall that the RC4 algorithm of the “Introductory Example” on page 9 is called symmetric-key encryption because the key used to encrypt is the same key that will be needed to decrypt. Crypto-C offers two types of symmetric-key encryption operations: stream ciphers and block ciphers. RC4, the only stream cipher in Crypto-C, was used in the “Introductory Example” on page 9. This chapter gives examples of the block ciphers DES, RC2, and RC5.

For an example of public-key encryption, see “Performing RSA Operations” on page 186.

Block Ciphers

DES with CBC

The example in this section corresponds to the file `descbc.c`.

Step 1: Creating an Algorithm Object

Declare a variable to be a `B_ALGORITHM_OBJ.` As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, it address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ encryptionObject = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&encryptionObject)) != 0)
  break;
```

Step 2: Setting the Algorithm Object

There are a number of DES AIs to choose from. See Table 3-6 on page 107 for a summary. For this example, choose `AI_FeedbackCipher`. `AI_FeedbackCipher` is a general-purpose AI that allows you to choose different block cipher methods, such as DES, RC2, and RC5. It also allows you to choose different feedback methods for your cipher. This makes updating your program to use a different block cipher or feedback method easy; you simply have to replace the arguments.

See "Block Ciphers" on page 36 of this manual for an overview of block cipher algorithms and feedback methods. We will implement DES in CBC mode using the padding scheme defined in PKCS #5.

The description of `AI_FeedbackCipher` in Chapter 2 of the *Library Reference Manual* says that the format of the ***`info`*** supplied to `B_SetAlgorithmInfo` is a pointer to a

`B_BLK_CIPHER_W_FEEDBACK_PARAMS` structure:

```
typedef struct {
  unsigned char *encryptionMethodName;    /* examples include “des”, “rc5” */
  POINTER        encryptionParams;                 /* e.g., RC5 parameters */
  unsigned char *feedbackMethodName;
  POINTER        feedbackParams;             /* Points at init vector ITEM */
                                        /* for all feedback modes except cfb */
  unsigned char *paddingMethodName;
  POINTER        paddingParams;        /* Ignored for now, but may be used */
                                                /* for new padding schemes */
} B_BLK_CIPHER_W_FEEDBACK_PARAMS;
```

encryptionMethodName is the block cipher that you will use; for this example, use “`des`”. The information in the *Library Reference Manual* indicates that you do not need to supply any parameters for the DES encryption algorithm, so set ***encryptionParams*** to `NULL_PTR`.

Use Cipher Block Chaining (CBC) for your feedback method. For this method, the *Library Reference Manual* says that ***feedbackParams*** is an `ITEM` structure containing the initialization vector:

```
  typedef struct {
    unsigned char *data;
    unsigned int   len;
  } ITEM;
```

See “Block Ciphers” on page 36 for an explanation of initialization vectors. Use a random number generator to produce an IV. Remember, the IV is not secret and will not assist anyone in breaking the encryption, but you should use a different IV for different messages. The size of the IV is eight bytes, because DES encrypts blocks of eight bytes. The size of the IV is always related to the size of the block, not the key:

```
unsigned char *ivBytes[BLOCK_SIZE];
B_BLK_CIPHER_W_FEEDBACK_PARAMS fbParams;

ITEM ivItem;
```

```
/* Complete steps 1 - 4 of Generating Random Numbers, then */
/* call B_GenerateRandomBytes.                             */

if ((status = B_GenerateRandomBytes
     (randomAlgorithm, ivBytes, 8,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

ivItem.data = ivBytes;
ivItem.len = 8;
```

You must also indicate that you want to use the standard CBC padding which is defined in PKCS #5; do this by setting ***fbParams.paddingMethodName*** to "pad". You do not need to pass in any padding parameters for this padding scheme. Again, “Block Ciphers” on page 36 explains padding.

Now set up the B_BLK_CIPHER_W_FEEDBACK_PARAMS structure:

```
fbParams.encryptionMethodName = (unsigned char *)"des";
fbParams.encryptionParams = NULL_PTR;
fbParams.feedbackMethodName = (unsigned char *)"cbc";
fbParams.feedbackParams = (POINTER)&ivItem;
fbParams.paddingMethodName = (unsigned char *)"pad";
fbParams.paddingParams = NULL_PTR;

if ((status = B_SetAlgorithmInfo
     (encryptionObject, AI_FeedbackCipher,(POINTER)&fbParams)) != 0)
  break;
```

Step 3: Init

You need a key before you can initialize the object for encryption.

Step 3a: Creating the Key Object

```
B_KEY_OBJ desKey = (B_KEY_OBJ)NULL_PTR;

if ((status = B_CreateKeyObject (&desKey)) != 0)
  break;
```

Step 3b: Setting the Key Object

You want to use a KI compatible with DES encryption, so return to the entry for `AI_FeedbackCipher` in Chapter 2 of the *Library Reference Manual*:

Key info types for `keyObject` in `B_EncryptInit` or `B_DecryptInit`:
Depends on cipher type, as follows:

Cipher	**KIs**
DES	`KI_Item`, `KI_DES8`, `KI_DES8Strong`, `KI_8Byte`

See “Summary of KIs” on page 115 of this manual for a discussion of the KIs. For this example, you will use `KI_DES8Strong`. Its entry in the *Library Reference Manual* states:

Format of *info* supplied to `B_SetKeyInfo`:
pointer to an unsigned char array which holds the 8-byte DES key.
The key is DES parity-adjusted when it is copied to the key object.

Use a random number generator to produce eight bytes for the key:

```
    unsigned char keyData[8];

   /* Complete steps 1 - 4 of Generating Random Numbers, */
   /* then call B_GenerateRandomBytes. */
   if ((status = B_GenerateRandomBytes
         (randomAlgorithm, keyData, 8,
          (A_SURRENDER_CTX *)NULL_PTR)) != 0)
      break;

if ((status = B_SetKeyInfo
     (desKey, KI_DES8Strong, (POINTER)keyData)) != 0)
  break;
```

Now that you have a key, you need an algorithm chooser and a surrender context.

This is a speedy function, so you can use a properly cast NULL_PTR for the surrender context; but you do want to build a chooser:

```
B_ALGORITHM_METHOD *DES_CBC_CHOOSER[] = {
  &AM_CBC_ENCRYPT,
  &AM_DES_ENCRYPT,
  &AM_SHA_RANDOM,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_EncryptInit
     (encryptionObject, desKey, DES_CBC_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Enter the data to encrypt with B_EncryptUpdate. The *Library Reference Manual* Chapter 2 entry for AI_FeedbackCipher states that you may pass (B_ALGORITHM_OBJ)NULL_PTR for all ***randomAlgorithm*** arguments. Once you have your input, call B_EncryptUpdate.

Remember that DES is a block cipher and requires input that is a multiple of eight bytes. Because you set ***fbParams.paddingMethodName*** to "pad" (see page 162), Crypto-C will pad to make the input a multiple of eight bytes. That means that the output buffer should be at least eight bytes longer than the input length. DES is a fast algorithm, so for small amounts of data it is reasonable to pass a properly cast NULL_PTR for the surrender context. If you want to pass a surrender context, you can:

```
static char *dataToEncrypt = "Encrypt this sentence.";
unsigned char *encryptedData = NULL_PTR;
unsigned int outputBufferSize;
unsigned int outputLenUpdate, outputLenFinal;
unsigned int encryptedDataLen;

encryptedDataLen = dataToEncryptLen + 8;
encryptedData = T_malloc (encryptedDataLen);
if ((status = (encryptedData == NULL_PTR)) != 0)
  break;
```

```
if ((status = B_EncryptUpdate
     (encryptionObject, encryptedData, &outputLenUpdate,
      encryptedDataLen, (unsigned char *)dataToEncrypt,
      dataToEncryptLen, (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

```
if ((status = B_EncryptFinal
     (encryptionObject, encryptedData + outputLenUpdate,
      &outputLenFinal, encryptedDataLen - outputLenUpdate,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects that you created and free up any memory that you allocated:

```
B_DestroyKeyObject (&desKey);
B_DestroyAlgorithmObject (&encryptionObject);
B_DestroyAlgorithmObject (&randomAlgorithm);
T_free (encryptedData);
```

Note: Using `T_free` means you can no longer access the data at that address. Do not free a buffer until you no longer need the data it contains. If you will need the data later, you might want to save it to a file first.

Decrypting

As in the “Introductory Example” on page 9, decrypting is similar to encrypting. Use the same AI, IV, and key data. Use the proper decryption AM and call `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`.

RC2

RC2 is a variable-key-size block cipher. Whereas a DES key requires eight bytes — no more, no less — an RC2 key can be anywhere between one and 128 bytes. The larger the key, the greater the security. RC2 is called a block cipher because it encrypts 8-byte blocks. Recall that DES also is a block cipher that encrypts 8-byte blocks. That means RC2 can serve as a drop-in replacement for DES. The steps for using `AI_FeedbackCipher` with RC2 are almost identical to those for DES.

The example in this section corresponds to the file `rc2.c`.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ rc2Encrypter = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&rc2Encrypter)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There are a number of RC2 AIs to choose from. See Table 3-6 on page 107 for a summary. Choose `AI_FeedbackCipher`; as in the previous example, the format of the ***info*** supplied to `B_SetAlgorithmInfo` is a pointer to a `B_BLK_CIPHER_W_FEEDBACK_PARAMS` structure:

```
typedef struct {
  unsigned char *encryptionMethodName;    /* examples include “des”, “rc5” */
  POINTER        encryptionParams;                 /* e.g., RC5 parameters */
  unsigned char *feedbackMethodName;
  POINTER        feedbackParams;              /* Points at init vector ITEM */
                                        /* for all feedback modes except cfb */
  unsigned char *paddingMethodName;
  POINTER        paddingParams;          /* Ignored for now, but may be used */
                                                 /* for new padding schemes */
} B_BLK_CIPHER_W_FEEDBACK_PARAMS;
```

Once again, ***encryptionMethodName*** is the block cipher that you will use; in this example, use “`rc2`”. All the other parameters are the same as for DES, except ***encryptionParams***. For RC2, the *Library Reference Manual* indicates that you need to supply an `A_RC2_PARAMS` structure for the RC2 encryption algorithm:

```
typedef struct {
  unsigned int effectiveKeyBits;            /* effective key size in bits */
} A_RC2_PARAMS;
```

There is a distinction between key size and effective key bits. The RC2 algorithm begins by building a 128-byte table based on the key. The total number of possible tables is limited by the number of effective key bits. Using 80 effective key bits is generally sufficient for most applications.

Use Cipher Block Chaining (CBC) for your feedback method. Once again, for this method, you need an initialization vector; use a random number generator to produce one. Remember, the IV is not secret and will not assist anyone in breaking the encryption. Its size will be eight bytes, because RC2 encrypts blocks of eight bytes. The *Library Reference Manual* says that ***feedbackParams*** is an `ITEM` structure containing the initialization vector:

```
  typedef struct {
    unsigned char *data;
    unsigned int   len;
  } ITEM;
```

now you can set your algorithm object as follows:

```
ITEM ivItem;
unsigned char initVector[BLOCK_SIZE];
A_RC2_PARAMS rc2Params;
B_BLK_CIPHER_W_FEEDBACK_PARAMS fbParams;

/* Complete steps 1 - 4 of Generating Random Numbers,
   then call B_GenerateRandomBytes. */
if ((status = B_GenerateRandomBytes
     (randomAlgorithm, (unsigned char *)initVector, 8,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

rc2Params.effectiveKeyBits = 80;
ivItem.data = initVector;
ivItem.len = BLOCK_SIZE;
```

```
fbParams.encryptionMethodName = (unsigned char *)"rc2";
fbParams.encryptionParams = NULL_PTR;
fbParams.feedbackMethodName = (unsigned char *)"cbc";
fbParams.feedbackParams = (POINTER)&ivItem;
fbParams.paddingMethodName = (unsigned char *)"pad";
fbParams.paddingParams = NULL_PTR;

if ((status = B_SetAlgorithmInfo
     (rc2Encrypter, AI_FeedbackCipher, (POINTER)&fbParams)) != 0)
  break;
```

Step 3: Init

You need a key before you can initialize the algorithm object for encryption.

Step 3a: Creating a Key Object

```
B_KEY_OBJ rc2Key = (B_KEY_OBJ)NULL_PTR;

if ((status = B_CreateKeyObject (&rc2Key)) != 0)
  break
```

Step 3b: Setting the Key Object

You are using 80 effective key bits. That does not mean you need exactly ten bytes of key data, although for security reasons, it is important to use at least ten bytes. You can generate 24 bytes (192 bits) of key data and the algorithm will still work at 80 effective bits. Thus, in the future, if you want to increase the effective key bits, you do not have to change the code that generates key data, only the effective key bit parameter.

Key generation is almost the same as with DES, but you will use a different KI. In the *Library Reference Manual* Chapter 2 entry for `AI_FeedbackCipher`, you see you have a choice of KIs. Because your key is going to be 24 bytes, you cannot use `KI_8Byte`, so choose `KI_Item`. Looking up `KI_Item` in Chapter 3 of the *Library Reference Manual*, you find that the ***info*** you supply to `B_SetKeyInfo` is a pointer to an `ITEM` struct, which is

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

Use a random number generator to come up with 24 bytes.

```
ITEM rc2KeyItem;

rc2KeyItem.len = 24;
rc2KeyItem.data = T_malloc (rc2KeyItem.len);
if ((status = (rc2KeyItem.data == NULL_PTR)) != 0)
  break;

/* Complete steps 1 - 4 of Generating Random Numbers, then
     call B_GenerateRandomBytes. */
if ((status = B_GenerateRandomBytes
     (randomAlgorithm, rc2KeyItem.data, rc2KeyItem.len,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

if ((status = B_SetKeyInfo
     (rc2Key, KI_Item, (POINTER)&rc2KeyItem)) != 0)
  break;
```

It is a good idea to zeroize any sensitive data after leaving the `do-while`. In fact, you may want to zeroize the memory and free it up immediately after setting the key. To do so, first free the memory using `T_free`, then reset ***rc2KeyItem.data*** to `NULL_PTR`, duplicating the following sequence after the `do-while`. If there is an error inside the `do-while`, you will still zeroize and free sensitive data; if there is no error, you have reset to `NULL_PTR`, and the code after the `do-while` will not create havoc.

```
if (rc2KeyItem.data != NULL_PTR) {
  T_memset (rc2KeyItem.data, 0, rc2KeyItem.len);
  T_free (rc2KeyItem.data);
  rc2KeyItem.data = NULL_PTR;
  rc2KeyItem.len = 0;
}
```

You need an algorithm chooser and a surrender context. This is a speedy function, so it is reasonable to use a properly cast `NULL_PTR` for the surrender context. However, you do want to build a chooser:

```
B_ALGORITHM_METHOD *RC2_CHOOSER[] = {
  &AM_CBC_ENCRYPT,
  &AM_RC2_ENCRYPT,
  &AM_SHA_RANDOM,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_EncryptInit
     (rc2Encrypter, rc2Key, RC2_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Enter the data to encrypt through `B_EncryptUpdate`. From the *Library Reference Manual* Chapter 2 entry on `AI_FeedbackCipher`, you see that you can pass `(B_ALGORITHM_OBJ)NULL_PTR` for all ***randomAlgorithm*** arguments. Once you have your input, call `B_EncryptUpdate`.

Remember that RC2 is a block cipher and requires that the input be a multiple of eight bytes. Because you set ***fbParams***`.paddingMethodName` to `"pad" (see` page 166), Crypto-C will pad to make the input a multiple of eight bytes. That means that the output buffer should be at least eight bytes larger than the input length.

RC2 is a fast algorithm, so for small amounts of data it is reasonable to pass a properly cast `NULL_PTR` for the surrender context. If you want to pass a surrender context, you can:

```
/* Assume dataToEncrypt points to already set data and
     dataToEncryptLen has been set to the number of bytes
     in dataToEncrypt. */

unsigned char *dataToEncrypt;
unsigned char *encryptedData = NULL_PTR;
unsigned int dataToEncryptLen;
unsigned int encryptedDataLen;
unsigned int outputLenUpdate;
encryptedDataLen = dataToEncryptLen + 8;
encryptedData = T_malloc (encryptedDataLen);
```

```
if ((status = (encryptedData == NULL_PTR)) != 0)
  break;

if ((status = B_EncryptUpdate
     (rc2Encrypter, encryptedData, &outputLenUpdate,
      encryptedDataLen, dataToEncrypt, dataToEncryptLen,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

```
unsigned int outputLenFinal;
if ((status = B_EncryptFinal
     (rc2Encrypter, encryptedData + outputLenUpdate,
      &outputLenFinal, encryptedDataLen - outputLenUpdate,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects created and free up any memory allocated:

```
B_DestroyKeyObject (&rc2Key);
B_DestroyAlgorithmObject (&rc2Encrypter);
B_DestroyAlgorithmObject (&randomAlgorithm);

if (encryptedData != NULL_PTR) {
  T_memset (encryptedData, 0, encryptedDataLen);
  T_free (encryptedData);
  encryptedData = NULL_PTR;
}

if (rc2KeyItem.data != NULL_PTR) {
  T_memset (rc2KeyItem.data, 0, rc2KeyItem.len);
  T_free (rc2KeyItem.data);
  rc2KeyItem.data = NULL_PTR;
  rc2KeyItem.len = 0;
}
```

Decrypting

As with the "Introductory Example" on page 9, decrypting is similar to encrypting. Use the same AI, IV, and key. Use the proper decrypting AM and call `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`.

RC5

RC5 is more properly known as RC5 *w*/*r*/*b*, where *w* stands for word size, *r* stands for rounds and *b* stands for key size in bytes.

The word size parameter is designed to take advantage of variable hardware word sizes. A hardware implementation can choose a 16-, 32-, or 64-bit word size, depending on how many bits make up a register, or word. Software implementations of RC5 can emulate any word size, regardless of the size of the machine's register size. Crypto-C implements word sizes of 32 or 64 bits; the 64-bit implementation has not been optimized.

The next feature of RC5 is the rounds parameter. Increasing the number of rounds increases security, but slows down the operation. This allows the application developer to establish a desired trade-off between security and speed. RC5 allows round counts from 0 to 255 rounds. RSA Data Security, Inc. recommends using at least 16 rounds for the 32-bit word implementation. Analysis indicates that, in theory, RC5 may be susceptible to various attacks for values less than 16.

The last feature is the variable key size. Whereas a DES key requires eight bytes, an RC5 key can be anywhere between zero and 255 bytes. The larger the key, the greater the security. Key size has no appreciable effect on speed.

RC5 is a block cipher; the size of the blocks is twice the word size. For RC5 32/*r*/*b*, the block size is 64 bits or 8 bytes; for RC5 64/*r*/*b,* the block size is 128 bits or 16 bytes.

The example in this section corresponds to the file `rc5.c`.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ` and as defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for

`B_CreateAlgorithmObject`.

```
B_ALGORITHM_OBJ rc5Encrypter = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&rc5Encrypter)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There are a number of RC5 AIs to choose from. See Table 3-6 on page 107 for descriptions. For this example, you will use a different cipher from `AI_FeedbackCipher`. Choose `AI_RC5_CBCPad`. The *Library Reference Manual* Chapter 2 entry for this AI indicates that the format of ***`info`*** supplied to `B_SetAlgorithmInfo` is:

```
typedef struct {
  unsigned int  version;                    /* currently 1.0 defined 0x10 */
  unsigned int  rounds;                     /* number of rounds (0 - 255) */
  unsigned int  wordSizeInBits;             /* AI_RC5_CBCPad requires 32 */
  unsigned char *iv;                            /* initialization vector */
} A_RC5_CBC_PARAMS;
```

As a provision for future revisions of the RC5 algorithm, Crypto-C includes a version number. So that the version number can be one byte, it is two hex digits. Version 1.0 is therefore `0x10`. Version 3.8, if there ever is one, will be `0x38`. The hex number `0x10` is the decimal number 16. Both are valid, but it is probably better to use `0x10` because it is easier to see as a version number.

For this example, you will use 12 rounds with a word size of 32.

Because you have chosen an AI that uses Cipher Block Chaining (CBC), you need an initialization vector. Use a random number generator to produce an IV. Because the word size is 32, the block size is 64 bits or eight bytes, and your IV must be eight bytes long. Remember, the IV is not secret and will not assist anyone in breaking the encryption. Its size will be eight bytes, because RC5 encrypts blocks of eight bytes.

Remember, the IV is related to the block, not the key:

```
unsigned char initVector[8];
A_RC5_CBC_PARAMS rc5Params;

/* Complete steps 1 - 4 of Generating Random Numbers,
   then call B_GenerateRandomBytes. */

if ((status = B_GenerateRandomBytes
     (randomAlgorithm, (unsigned char *)initVector, 8,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

rc5Params.version = 0x10;
rc5Params.rounds = 12;
rc5Params.wordSizeInBits = 32;
rc5Params.iv = (unsigned char *)initVector;

if ((status = B_SetAlgorithmInfo
     (rc5Encrypter, AI_RC5_CBCPad, (POINTER)&rc5Params)) != 0)
  break;
```

Step 3: Init

You need a key before you can initialize the algorithm object for encryption.

Step 3a: Creating A Key Object

```
B_KEY_OBJ rc5Key = (B_KEY_OBJ)NULL_PTR;

if ((status = B_CreateKeyObject (&rc5Key)) != 0)
  break;
```

Step 3b: Setting The Key Object

For this example, you will use 10 key bytes (80 bits). In the *Library Reference Manual* Chapter 2 entry for `AI_RC5_CBCPad`, you see you must use `KI_Item`. Looking up `KI_Item` in Chapter 3 of the *Library Reference Manual*, you find that the ***info*** you supply to `B_SetKeyInfo` is a pointer to an `ITEM` struct, defined in `algobal.h`:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

Use a random number generator to create 10 bytes:

```
ITEM rc5KeyItem;

rc5KeyItem.data = NULL_PTR;
rc5KeyItem.len = 10;
rc5KeyItem.data = T_malloc (rc5KeyItem.len);
if ((status = (rc5KeyItem.data == NULL_PTR)) != 0)
  break;

if ((status = B_GenerateRandomBytes
     (randomAlgorithm, rc5KeyItem.data, rc5KeyItem.len,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

if ((status = B_SetKeyInfo
     (rc5Key, KI_Item, (POINTER)&rc5KeyItem)) != 0)
  break;
```

It is a good idea to zeroize any sensitive data after leaving the `do-while`. In fact, you may want to zeroize the memory and free it up immediately after you set the key. To do so, first free the memory using `T_free`, then reset `rc5KeyItem.data` to `NULL_PTR` and duplicate the following sequence after the `do-while`. If there is an error inside the `do-while` before you zeroize and free, you will still perform this important task; if there is not an error, by resetting to `NULL_PTR`, you ensure that the code after the `do-while` will not create havoc:

```
if (rc5KeyItem.data != NULL_PTR) {
  T_memset (rc5KeyItem.data, 0, rc5KeyItem.len);
  T_free (rc5KeyItem.data);
  rc5KeyItem.data = NULL_PTR;
  rc5KeyItem.len = 0;
};
```

Now that you have a key, you need an algorithm chooser and a surrender context.

This is a speedy function, so you can use a properly cast NULL_PTR for the surrender context; but you do want to build a chooser:

```
B_ALGORITHM_METHOD *RC5_CHOOSER[] = {
  &AM_RC5_CBC_ENCRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_EncryptInit
     (rc5Encrypter, rc5Key, RC5_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Enter the data to encrypt through B_EncryptUpdate. From the *Library Reference Manual* Chapter 2 entry on AI_RC5_CBCPad you learn that you may pass (B_ALGORITHM_OBJ)NULL_PTR for all randomAlgorithm arguments. Assuming you have some input, call B_EncryptUpdate.

Remember that RC5 is a block cipher and requires input that is a multiple of eight bytes. Because you are using AI_RC5_CBCPad, Crypto-C will pad to make the input a multiple of eight bytes. That means that the output buffer should be at least eight bytes larger than the input length.

RC5 is a fast algorithm, so for small amounts of data it is reasonable to pass a properly cast NULL_PTR for the surrender context. If you want to pass a surrender context, you can:

```
/* Assume dataToEncrypt points to already set data and
     dataToEncryptLen has been set to the number of bytes
     in dataToEncrypt. */

unsigned char *dataToEncrypt;
unsigned char *encryptedData = NULL_PTR;
unsigned int dataToEncryptLen;
unsigned int encryptedDataLen;
unsigned int outputLenUpdate;
```

```
encryptedDataLen = dataToEncryptLen + 8;
encryptedData = T_malloc (encryptedDataLen);
if ((status = (encryptedData == NULL_PTR)) != 0)
  break;

if ((status = B_EncryptUpdate
     (rc5Encrypter, encryptedData, &outputLenUpdate,
      encryptedDataLen, dataToEncrypt, dataToEncryptLen,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

```
unsigned int outputLenFinal;

if ((status = B_EncryptFinal
     (rc5Encrypter, encryptedData + outputLenUpdate,
      &outputLenFinal, dataToEncryptLen + 8 - outputLenUpdate,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects that you created and free up any memory that you allocated.

```
B_DestroyKeyObject (&rc5Key);
B_DestroyAlgorithmObject (&rc5Encrypter);
B_DestroyAlgorithmObject (&randomAlgorithm);
if (rc5KeyItem.data != NULL_PTR) {
  T_memset (rc5KeyItem.data, 0, rc5KeyItem.len);
  T_free (rc5KeyItem.data);
  rc5KeyItem.data = NULL_PTR;
  rc5KeyItem.len = 0;
}
```

```
if (encryptedData != NULL_PTR) {
  T_memset (encryptedData, 0, encryptedDataLen);
  T_free (encryptedData);
  encryptedData = NULL_PTR;
}
```

Decrypting

As in the “Introductory Example” on page 9, decrypting is similar to encrypting. Use the same AI, IV, and key data. Use the proper decrypting AM and call `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`.

Password-Based Encryption

In previous encryption methods, you used a random number generator to produce a key. In password-based encryption (PBE), you will use a message digest algorithm to derive a key from a password. See “Message Digests” on page 46 for information on that topic.

For encryption, enter a password, append a salt to the password (see Step 2), and digest that quantity. Extract the required number of bytes from the digest and you have a key. Use that key to encrypt data using DES or RC2.

For decryption, enter a password, append the same salt, and then digest. Extract the required number of bytes from the digest and use them as a key to decrypt. If you entered the same password that you used to encrypt, you will obtain the same digest and hence the same key, and the encrypted data will decrypt to the original data.

Crypto-C will automatically append the salt, digest the data, and extract the key.

The example in this section corresponds to the file `pbe.c`.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ pbEncrypter = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&pbEncrypter)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There are a number of PBE AIs to choose from (see “Summary of AIs” on page 106 for a more detailed description). For now, choose `AI_MD5WithRC2_CBCPad`. In Chapter 2 of the *Library Reference Manual*, the description of this AI indicates the format of ***info*** supplied to `B_SetAlgorithmInfo` is:

```
typedef struct {
  unsigned int   effectiveKeyBits;          /* effective key size in bits */
  unsigned char *salt;                     /* pointer to 8 byte salt value */
  unsigned int   iterationCount;                       /* iteration count */
} B_RC2_PBE_PARAMS;
```

The section “RC2” on page 38 contains an explanation of effective key bits. The salt is a value that provides security against dictionary attacks or precomputation. An attacker could precompute the digests of thousands of possible passwords, creating a “dictionary” of likely keys. But recall that when you digest, changing input data even a little changes the resulting digest. By digesting the password with a salt, the attacker’s dictionary is rendered useless. The attacker would have to create a dictionary of the keys that were generated from each password; then each password would have to have a dictionary of each possible salt. The salt is not secret; knowing the salt will not help anyone without the password to decrypt the data.

To produce the salt, create an eight-byte buffer and then employ a random number generator to generate eight bytes. The iteration count is the number of times Crypto-C will digest. If that value is one, digest the password and salt once; if it is two, digest the password and salt, then digest the digest, and so on. Each iteration will increase an attacker’s task greatly. Five is generally sufficient for most applications:

```
#define SALT_LEN 8

B_RC2_PBE_PARAMS rc2PBEParams;
unsigned char saltData[SALT_LEN];

/* Complete steps 1 - 4 of Generating Random Numbers,
   then call B_GenerateRandomBytes.*/
if ((status = B_GenerateRandomBytes
     (randomAlgorithm, saltData, SALT_LEN,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

```
rc2PBEParams.effectiveKeyBits = 64;
rc2PBEParams.salt = saltData;
rc2PBEParams.iterationCount = 5;

if ((status = B_SetAlgorithmInfo
     (pbEncrypter, AI_MD5WithRC2_CBCPad,
      (POINTER)&rc2PBEParams)) != 0)
  break;
```

Step 3: Init

You need a key before you can initialize the algorithm object for encryption. In PBE, the password is the key. Simply enter the password data as the key data; Crypto-C will generate the symmetric key from the password and salt.

Step 3a: Creating A Key Object

```
#define MAX_PW_LEN 20

B_KEY_OBJ pbeKey = (B_KEY_OBJ)NULL_PTR;

if ((status = B_CreateKeyObject (&pbeKey)) != 0)
  break;
```

Step 3b: Setting The Key Object

In the *Library Reference Manual* Chapter 2 entry for `AI_MD5WithRC2_CBCPad`, you see you have only one choice for a KI: `KI_Item`. Looking up `KI_Item` in Chapter 3 of the *Library Reference Manual*, you find that the ***`info`*** you supply to `B_SetKeyInfo` is a pointer to an `ITEM` structure, which is:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

The data portion of the `struct` is the password. For this example, we will use the following method to enter the password. This method for entering a password is *not* secure; it is used for illustrative purposes only. It is not for duplication:

```
unsigned char enteredPassword[MAX_PW_LEN];
ITEM pbeKeyItem;

puts ("Enter the password, then press Return or Enter");
gets ((char *)enteredPassword);

pbeKeyItem.data = enteredPassword;
pbeKeyItem.len = strlen (enteredPassword);

if ((status = B_SetKeyInfo
      (pbeKey, KI_Item, (POINTER)&pbeKeyItem)) != 0)
  break;
```

You should zeroize any sensitive data after leaving the `do-while`. In fact, you might want to zeroize the memory immediately after you set the key:

```
T_memset (pbeKeyItem.data, 0, MAX_PW_LEN);
```

Now that you have a key, you need an algorithm chooser and a surrender context. This is a speedy function, so it is reasonable to use a properly cast `NULL_PTR` for the surrender context. You do want to build a chooser:

```
B_ALGORITHM_METHOD *PBE_CHOOSER[] = {
  &AM_MD5,
  &AM_RC2_CBC_ENCRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_EncryptInit
     (pbEncrypter, pbeKey, PBE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Enter the data to encrypt through `B_EncryptUpdate`. The *Library Reference Manual* Chapter 2 entry on `AI_MD5WithRC2_CBCPad` states that you can pass `(B_ALGORITHM_OBJ)NULL_PTR` for all `randomAlgorithm` arguments. Assuming you have some input data, call `B_EncryptUpdate`. Remember that RC2 is a block cipher and requires the input to be a multiple of eight bytes. But because you are using AI_MD5WithRC2_CBCPad, Crypto-C will pad to make the input a multiple of eight bytes. That means, though, that the output buffer should be at least eight bytes larger than the input length.

PBE with MD5 and RC2 is a fast algorithm, so for small amounts of data, you can pass a properly cast NULL_PTR for the surrender context. If you want to pass a surrender context, you can:

```
/* Assume dataToEncrypt points to already set data and
     dataToEncryptLen has been set to the number of bytes
     in dataToEncrypt. */

#define BLOCK_LEN 8
unsigned char *dataToEncrypt;
unsigned char *encryptedData = NULL_PTR;
unsigned int dataToEncryptLen;
unsigned int encryptedDataLen;
unsigned int outputLenUpdate;

encryptedDataLen = dataToEncryptLen + BLOCK_LEN;
encryptedData = T_malloc (encryptedDataLen);
if ((status = (encryptedData == NULL_PTR)) != 0)
  break;

if ((status = B_EncryptUpdate
     (pbEncrypter, encryptedData, &outputLenUpdate,
      encryptedDataLen, dataToEncrypt, dataToEncryptLen,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

```
unsigned int outputLenFinal;

if ((status = B_EncryptFinal
     (pbEncrypter, encryptedData + outputLenUpdate,
      &outputLenFinal, encryptedDataLen - outputLenUpdate,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects and free up any allocated memory:

```
B_DestroyKeyObject (&pbeKey);
B_DestroyAlgorithmObject (&pbEncrypter);
B_DestroyAlgorithmObject (&randomAlgorithm);

  if (pbeKeyItem.data, 0, MAX_PW_LEN) {
    T_memset (pbeKeyItem.data, 0, MAX_PW_LEN);
    T_free (pbekeyItem.data);
     pbeKeyItem.data = NULL_PTR;
  }

  if (encryptedData != NULL_PTR) {
    T_memset (encryptedData, 0, encryptedDataLen);
    T_free (encryptedData);
    encryptedData = NULL_PTR;
  }
```

Decrypting

As in the “Introductory Example” on page 9, decrypting is similar to encrypting. Use the same AI, password, and salt. Use the proper decrypting AM and call `B_DecryptInit`, `B_DecryptUpdate`, and `B_DecryptFinal`.

Chapter 6

Public-Key Operations

In public-key cryptography, two associated keys are necessary, one to encrypt and the other to decrypt. The sender encrypts a message using the recipient's public key. Once a message is encrypted, it can be decrypted with the recipient's private key. This is in contrast to algorithms like DES, RC2, RC4, and RC5, which are called symmetric-key encryption algorithms because the key used to encrypt is the same key needed to decrypt.

In public-key cryptography, it is also possible to encrypt using a private key. In this case, the sender takes the plaintext input and the private key and follows the same steps need to decrypt an encrypted file. This creates a ciphertext that can be read using the public key; to read it, the recipient follows the same steps needed to encrypt with the public key and restores it to the plaintext. Private-key encryption with public-key decryption is used for digital signatures and verification. See "RSA Digital Signatures" on page 198 and "DSA Signatures" on page 213 for more information.

Crypto-C supplies a number of public-key algorithms. These include:

- RSA encryption and decryption
- DSA signatures
- Diffie-Hellman key agreement
- Elliptic curve public-key operations

Performing RSA Operations

RSA is a public-key algorithm that relies on the difficulty of factoring a number that is the product of two large primes. If you are not familiar with the RSA algorithm and terminology, you may want to read “The RSA Algorithm” on page 50 before you continue.

The algorithm chooser used throughout the sections concerning executing the RSA algorithm can be found in “Algorithm Choosers” on page 118.

The example in this section corresponds to the file `rsapkcs.c`.

Generating a Key Pair

Before you can encrypt and decrypt, you need a key pair. The key pair consists of a private key and its associated public key. Generating a key pair is not trivial. The RSA algorithm relies on very large prime numbers, which are produced during key pair generation. This could be fairly time-consuming, so we recommend you use a surrender context. The surrender context used below is the one in “The Surrender Context” on page 120.

Most Crypto-C operations follow the six-step procedure outlined in the “Introductory Example” on page 9. Generating a key pair needs only five of the steps; there is no Update call.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ keypairGenerator = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&keypairGenerator)) != 0)
  break;
```

Step 2: Setting the Algorithm Object

For this example, use `AI_RSAKeyGen` to generate an RSA key pair. The *Library Reference Manual* Chapter 2 entry for `AI_RSAKeyGen` states that the ***`info`*** for `B_SetAlgorithmInfo` is a pointer to an A_RSA_KEY_GEN_PARAMS structure, defined as:

```
typedef struct {
  unsigned int modulusBits;                     /* size of modulus in bits */
  ITEM         publicExponent;                    /* fixed public exponent */
} A_RSA_KEY_GEN_PARAMS;
```

where ITEM is:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

The size of the modulus in bits can be any number from 256 to 2048, the larger the modulus, the greater the security. Unfortunately, the larger the modulus, the longer it takes to generate key pairs and to encrypt and decrypt. RSA Data Security, Inc., recommends 768 bits or more for applications. In testing and learning, though, it is safe to choose a smaller modulus to save time. For this exercise, choose 512.

The public exponent is usually one of two values: $F_0 = 3$ or $F_4 = 65537$. Recall that the algorithm requires a public exponent that has no common divisor with $(p-1)(q-1)$. With F_0 or F_4, it is easier to find primes p and q that meet that requirement. F_4 is also a good choice for a public exponent because it is large, prime, and of low weight. Weight here refers to the number of 1's in the binary representation: in hex, F_4 is 01 00 01. The F in F_0 and F_4 stands for Pierre de Fermat, the 17th-century mathematician who first described the special properties of these and other interesting numbers. For more information on F_4 (and other Fermat numbers), see ITU-T X.509, Annex D.

For this example, choose F_4:

```
A_RSA_KEY_GEN_PARAMS keygenParams;
static unsigned char f4Data[3] = {0x01, 0x00, 0x01};

keygenParams.modulusBits = 512;
keygenParams.publicExponent.data = f4Data;
keygenParams.publicExponent.len = 3;
if ((status = B_SetAlgorithmInfo
     (keypairGenerator, AI_RSAKeyGen,
      (POINTER)&keygenParams)) != 0)
  break;
```

Step 3: Init

Look up the description and prototype for `B_GenerateInit` in Chapter 4 of the *Library Reference Manual*. For this example, you can use the following:

```
if ((status = B_GenerateInit
     (keypairGenerator, RSA_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Here, you use `NULL_PTR` for the surrender context because `B_GenerateInit` is a speedy function. `B_GenerateKeypair` in Step 5 is the time-consuming function.

Step 4: Update

There is no Step 4 in generating a key pair.

Step 5: Generate

Find the description and prototype for `B_GenerateKeypair` in Chapter 4 of the *Library Reference Manual*. This function takes five arguments. The first is the algorithm object: for this example, it is ***`keypairGenerator`***. The second and third are key objects. For this call, all you have to do is create the key objects; they will be set by `B_GenerateKeypair`. The fourth argument is a random algorithm. For this, complete Steps 1 through 4 of "Generating Random Numbers" on page 147. You do not need random bytes, only an algorithm that can generate them. The algorithm chooser you are using (defined in "Algorithm Choosers" on page 118) contains the AM for SHA1 random number generation.

The last argument is the surrender context. This function call can take a while, although the amount of time is not uniform. On slower machines, it may take over two or three minutes to generate a 512-bit key pair, or it may take only 17 seconds.

Crypto-C needs to find two primes of the proper size. To find a prime, Crypto-C generates a candidate and tests to see if it is prime. If the candidate passes the test, Crypto-C has one of the primes; if not, Crypto-C builds a new number. If you are lucky, two early numbers Crypto-C creates will pass the test. Sometimes, though, Crypto-C has to try many numbers before it finds a pair.

Note: The numbers Crypto-C produces are not provably prime. They are numbers for which the probability is very low that they are not prime. This does not affect the accuracy of the algorithm and will not appreciably decrease security.

When you generate a key pair, it can look as if your program has stopped or as if the machine has frozen up. To help allay fears of disaster, use the surrender function outlined in “The Surrender Context” on page 120. It will print out a dot every second to let you know the program is running properly. If the dots do not appear, then you know something is wrong:

```
B_KEY_OBJ publicKey = (B_KEY_OBJ)NULL_PTR;
B_KEY_OBJ privateKey = (B_KEY_OBJ)NULL_PTR;

if ((status = B_CreateKeyObject (&publicKey)) != 0)
  break;

if ((status = B_CreateKeyObject (&privateKey)) != 0)
  break;

/* generalFlag is for the surrender function. */
generalFlag = 0;
if ((status = B_GenerateKeypair
     (keypairGenerator, publicKey, privateKey,
      randomAlgorithm, &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

When you are done with your objects, remember to destroy them:

```
B_DestroyAlgorithmObject (&randomAlgorithm);
B_DestroyAlgorithmObject (&keypairGenerator);
B_DestroyKeyObject (&publicKey);
B_DestroyKeyObject (&privateKey);
```

Distributing an RSA Public Key

After generating a key pair, you need to make the public key available to the public.

Crypto-C Format

publicKey is a key object that was set by the Crypto-C function `B_GenerateKeypair`. Its key info type (KI) is `KI_RSAPublic`. In the *Library Reference Manual* Chapter 3 entry on

KI_RSAPublic, the section titled "Format of ***info*** returned by B_GetKeyInfo:" tells you that the function returns a pointer to an A_RSA_KEY struct:

```
typedef struct {
  ITEM modulus;                                                     /* modulus */
  ITEM exponent;                                                   /* exponent */
} A_RSA_KEY;
```

So you need to declare a variable to be a pointer to such a struct and pass this variable's address as the argument.

Using the *Library Reference Manual* Chapter 4 prototype for B_GetKeyInfo as a guide, write the following:

```
A_RSA_KEY *getPublicKey = (A_RSA_KEY *)NULL_PTR;

if ((status = B_GetKeyInfo
     ((POINTER *)&getPublicKey, publicKey, KI_RSAPublic)) != 0)
  break;
```

If you looked at the elements of the struct:

```
getPublicKey->modulus.data
getPublicKey->modulus.len
getPublicKey->exponent.data
getPublicKey->exponent.len
```

you could see the public key that Crypto-C generated. This is the information you would make public.

Note: If you want to email the information, you will not be able to send the information over most email systems because the data is in binary form, not ASCII. Crypto-C offers encoding and decoding functions to convert between binary and ASCII. See "Converting Data Between Binary and ASCII" on page 154 for more information.

BER/DER Encoding

There is a problem with distributing the key in the above struct: it is not standard; it is unique to Crypto-C. If the recipient is not using Crypto-C, how do you give them the information? Suppose your application mails this key to a certification authority. What information do you send? There is a standard that defines what the public key consists of and how that information should be formatted: BER-encoding. It is defined in ASN.1, which defines the Basic Encoding Rules (BER) and Distinguished Encoding Rules (DER). See “BER/DER Encoding” on page 125 for more information.

You must put the key into DER format, encode it into ASCII, and email the encoding. The recipient will decode the DER string and convert the key information into the format of their choice.

This sounds difficult, but Crypto-C offers a means of doing it simply. Above, in order to obtain the key, you used `B_GetKeyInfo` with `KI_RSAPublic`. Chapter 3 of the *Library Reference Manual* also lists `KI_RSAPublicBER`, which states:

Format of *info* returned by `B_GetKeyInfo`**:**
pointer to an `ITEM` structure which gives the address and length of the DER-encoding. Note that `B_GetKeyInfo` returns an encoding which contains the object identifier for rsaEncryption (defined in PKCS #1) as opposed to rsa.

Crypto-C returns a pointer to where that information resides, not the information. Another call to Crypto-C might alter or destroy it. Therefore, once you get the pointer to the information, copy it into your own buffer:

```
ITEM *cryptocPublicKeyBER;
ITEM myPublicKeyBER;

myPublicKeyBER.data = NULL_PTR;

if ((status = B_GetKeyInfo
     ((POINTER *)&cryptocPublicKeyBER, publicKey,
      KI_RSAPublicBER)) != 0)
  break;

myPublicKeyBER.len = cryptocPublicKeyBER->len;
myPublicKeyBER.data = T_malloc (myPublicKeyBER.len);
if ((status = (myPublicKeyBER.data == NULL_PTR)) != 0)
  break;
T_memcpy (myPublicKeyBER.data, cryptocPublicKeyBER->data,
          myPublicKeyBER.len);
```

So, to distribute a key, you generate the key pair, get the key ***`info`*** in BER format with `B_GetKeyInfo` and `KI_RSAPublicBER`, encode the BER data into ASCII format, and send it off.

Remember to free any memory you allocated:

```
T_free (myPublicKeyBER.data);
```

Note: The conversion into BER or DER is known as BER-encoding or DER-encoding; the conversion between binary to ASCII is known as encoding and decoding. In general, the word "encoding" without "BER" in front of it means binary to ASCII. If the encoding is BER- or DER-encoding, the BER or DER should be explicitly stated.

RSA Public-Key Encryption

Follow Steps 1 through 6 to encrypt the following using an RSA public key:

```
static unsigned char dataToEncryptWithRSA[8] = {
  0x4a, 0x72, 0x55, 0x36, 0xda, 0x2f, 0xb9, 0x51
};
```

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ rsaEncryptor = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&rsaEncryptor)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There are a number of RSA AIs, described in Table 3-7 on page 110. For this example, use `AI_PKCS_RSAPublic`. This AI encrypts and decrypts data according to the Public-Key Cryptography Standard #1 (PKCS #1). See the PKCS document [1] for more information. According to the *Library Reference Manual* Chapter 2 entry for

AI_PKCS_RSAPublic, the ***info*** supplied to B_SetAlgorithmInfo is NULL_PTR:

```
if ((status = B_SetAlgorithmInfo
     (rsaEncryptor, AI_PKCS_RSAPublic, NULL_PTR)) != 0)
  break;
```

Step 3: Init

You will encrypt using the recipient's RSA public key. Normally, you would obtain the public key from the recipient or a certificate service. For this exercise, though, you will simply use the public key you generated in "Generating a Key Pair" on page 186. B_EncryptInit is quick, so you are safe in passing NULL_PTR as the surrender context:

```
if ((status = B_EncryptInit
     (rsaEncryptor, publicKey, RSA_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

The *Library Reference Manual* Chapter 2 entry on AI_PKCS_RSAPublic states:

Input constraints:
The total number of bytes to encrypt may not be more than *k* – 11, where *k* is the key's modulus size in bytes.

For this example, the key's size in bits is 512, which is 64 bytes. So you cannot pass more than 53 bytes. If you were encrypting more than 53 bytes, you could not use AI_PKCS_RSAPublic. If you had more than 53 bytes to encrypt and tried to break it up into smaller units, calling B_EncryptUpdate for each unit, it would not work. That is because PKCS RSA encryption adds padding, and the padding scheme needs at least 11 spare bytes to work. It is intended for digital envelopes and digital signatures, and in those situations, the number of bytes to encrypt is usually eight, 16, or (for BER-encoded digests) 34 or 35. If you want to encrypt larger amounts of data using the RSA algorithm, you must use AI_RSAPublic, also known as *raw RSA*. See "Raw RSA" on page 197 for more information.

You are encrypting eight bytes, so you do not need to worry about that constraint. However, the output of RSA encryption is the same size as the modulus, as described in "The RSA Algorithm" on page 50. That means you must set the output buffer, which will hold the encrypted data, to be the same size as the modulus. Your modulus is 512 bits, or 64 bytes.

Note: The input to the RSA algorithm must also be the same size as the modulus, but `AI_PKCS_RSAPublic` will automatically pad.

The description of `AI_PKCS_RSAPublic` notes that "`B_EncryptUpdate` and `B_EncryptFinal` require a random algorithm." The random number generator is for the padding. You do not need random bytes, only an algorithm that can generate them. Although RSA encryption is not as slow as key pair generation, you will not see an immediate response. Use a surrender context so that you know the program is running and has not frozen:

```
#define BLOCK_SIZE 64

unsigned char encryptedData[BLOCK_SIZE];
unsigned int outputLenUpdate;

/* generalFlag is for the surrender function.*/
generalFlag = 0;
if ((status = B_EncryptUpdate
     (rsaEncryptor, encryptedData, &outputLenUpdate,
      BLOCK_SIZE, (unsigned char *)dataToEncryptWithRSA, 8,
      randomAlgorithm, &generalSurrenderContext)) != 0)
  break;
```

Step 5: Final

```
unsigned int outputLenFinal;

/* generalFlag is for the surrender function.*/
generalFlag = 0;
if ((status = B_EncryptFinal
     (rsaEncryptor, encryptedData + outputLenUpdate,
      &outputLenFinal, BLOCK_SIZE - outputLenUpdate,
      randomAlgorithm, &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

When you are done with all your objects, remember to destroy them.

```
B_DestroyAlgorithmObject (&randomAlgorithm);
B_DestroyAlgorithmObject (&rsaEncryptor);
B_DestroyKeyObject (&publicKey);
```

RSA Private-Key Decryption

This example shows how to decrypt using an RSA private key. Remember that with Crypto-C, you have the choice of doing your private-key operations normally or utilizing the blinding technique (see "Timing Attacks and Blinding" on page 96). You make this choice in the algorithm chooser. For normal decryption operations, use `AM_RSA_CRT_DECRYPT`; to execute blinding, use `AM_RSA_CRT_DECRYPT_BLIND`.

Step 1: Creating an Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ rsaDecryptor = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&rsaDecryptor)) != 0)
  break;
```

Step 2: Setting the Algorithm Object

Because you used `AI_PKCS_RSAPublic` to encrypt, it is easiest to use `AI_PKCS_RSAPrivate` to decrypt. Crypto-C padded the data before encrypting; when you use the "matching" AI to decrypt, Crypto-C will automatically strip the padding. The *Library Reference Manual* Chapter 2 entry on this AI indicates the ***info*** supplied to `B_SetAlgorithmInfo` is `NULL_PTR`:

```
if ((status = B_SetAlgorithmInfo
     (rsaDecryptor, AI_PKCS_RSAPrivate, NULL_PTR)) != 0)
  break;
```

Step 3: Init

To decrypt, you must use the RSA private key that is associated with the public key that was used to encrypt, which would be the key you generated in "Generating a Key Pair" on page 186. B_DecryptInit is quick, so you are safe in passing NULL_PTR as the surrender context.

```
if ((status = B_DecryptInit
     (rsaDecryptor, privateKey, RSA_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

When you encrypted, there were certain constraints on the size of the input data to B_EncryptUpdate. The only constraint on the data passed to B_DecryptUpdate is that it be numerically less than the modulus. If the data you are decrypting was indeed encrypted using RSA, it will be.

You know the encryption process padded the original data, so, while the encrypted data is 64 bytes, the decrypted data will be less than 64 bytes. But you do not know how much less. For simplicity, make the decrypted data buffer 64 bytes large. Presumably, the encrypter added outputLenUpdate and outputLenFinal from the encryption to get the total number of bytes of encrypted data. The *Library Reference Manual* Chapter 2 entry on AI_PKCS_RSAPrivate indicates you may pass a properly cast NULL_PTR for ***randomAlgorithm*** arguments.

Although RSA decryption is not as slow as key pair generation, you will not see an immediate response. Use the surrender context given above so you know the program is running and has not frozen:

```
#define BLOCK_SIZE 64

unsigned char decryptedData[BLOCK_SIZE];
unsigned int outputLenTotal;
unsigned int outputLenUpdate;
  /* where outputLenTotal is the sum of the encryption's
       outputLenUpdate and outputLenFinal. The encrypter should
       send this information along with the encrypted data. */
```

```
/* generalFlag is for the surrender function.*/
generalFlag = 0;
if ((status = B_DecryptUpdate
     (rsaDecryptor, decryptedData, &outputLenUpdate, BLOCK_SIZE,
      encryptedData, outputLenTotal, NULL_PTR,
      &generalSurrenderContext)) != 0)
  break;
```

Step 5: Final

```
unsigned int outputLenFinal;

/* generalFlag is for the surrender function.*/
generalFlag = 0;
if ((status = B_DecryptFinal
     (rsaDecryptor, decryptedData + outputLenUpdate,
      &outputLenFinal, BLOCK_SIZE - outputLenUpdate, NULL_PTR,
      &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

When you are done with all objects, remember to destroy them:

```
B_DestroyAlgorithmObject (&rsaDecryptor);
B_DestroyKeyObject (&privateKey);
```

Raw RSA

When you used `AI_PKCS_RSAPublic`, you could not encrypt more than *k* – 11 bytes, where *k* was the size of the modulus in bytes. That is because PKCS RSA encryption pads, and the padding scheme needs 11 spare bytes to work. It is intended for digital envelopes and digital signatures; in those situations, the number of bytes to encrypt is usually eight, 16, or (for BER-encoded digests) 34 or 35. If you want to encrypt and decrypt more than *k* – 11 bytes, use raw RSA.

Note: In general, there should be no need for raw RSA encryption or decryption. For most applications, if you have a longer message, it is faster and simpler to encrypt the message with a symmetric algorithm and then use the RSA algorithm to encrypt the key. (See "Digital Envelopes" on page 54.) If you do use raw RSA encryption and decryption, your application must be responsible for adding and removing the necessary padding. We do not recommend using raw RSA unless you are familiar with the issues involved.

To encrypt more bytes than the PKCS AIs allow, use `AI_RSAPublic` for encryption and and `AI_RSAPrivate` for decryption. Note that this is different from the recommended use for these AIs, as described in the *Library Reference Manual*. There are two important constraints to consider when using these AIs:

- The total length of the data must be a multiple of the modulus size.

  If your data's length is not a multiple of the modulus size, your application must do the padding. When decrypting with raw RSA, Crypto-C will not strip the padding; the application must do that.
- The data must be numerically less than the modulus.

  To do this, divide your data into blocks that are one byte smaller than the modulus. Prepend one byte of 0 to each block. If the leading byte of the data is 0, your data will meet this second constraint.

  For example, suppose you wanted to encrypt 100 bytes with RSA using a 512-bit modulus. You would break the data into two blocks, the first one 63 bytes, the second 37. Prepend a 0 byte to the first block and it is now 64 bytes (512 bits). Prepend a 0 byte and append 26 pad bytes to the second block and it, too is now 64 bytes. Call `B_EncryptUpdate` for each of the two blocks, then `B_EncryptFinal`. This will produce 128 bytes of encrypted data.

  When decrypting, call `B_DecryptUpdate` once for all 128 bytes, then `B_DecryptFinal`. The application will have to then strip the prepended zeroes and the padding. You could also break the encrypted data into 64-byte blocks and call `B_DecryptUpdate` for each block and strip the padding then.

Some padding procedures are recommended; others are discouraged. For a description of one particular trusted padding system, see PKCS #1 v2 [1].

RSA Digital Signatures

The section "Authentication and Digital Signatures" on page 55 discusses what a digital signature is. This section describes how to write Crypto-C code that computes or verifies digital signatures. For signing, Crypto-C offers `B_SignInit`, `B_SignUpdate`, and `B_SignFinal`, which will digest the data and encrypt the digest using RSA encryption with a private key. For verification, Crypto-C offers `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal`, which will digest the data again, decrypt the signature with the RSA public key, and compare the digest to the decrypted signature.

Note that you cannot use the Sign and Verify functions if you do not want to digest the data. Some applications may not call for a digest; they may demand that the signature be the actual data encrypted with a private key. This is the case with some forms of authentication, for instance. In other cases, the data passed to the application has already been digested. In such an application, encrypt using AI_PKCS_RSAPrivate or AI_RSAPrivate; do not follow the model outlined here.

A digital signature is actually not the private-key encrypted digest of the data, but the private-key encrypted BER-encoding of the digest. (Remember that when you "encrypt" using the private key, you are actually following the same steps you use for decryption, even though you apply them to a plaintext file.) When you are using SHA1, this means the input data will be 35 bytes, not 20. The "encryption" follows the PKCS standards, so the data must be at least 11 bytes shorter than the modulus. Hence, the modulus must be at least 46 bytes (368 bits) for computing digital signatures using SHA1 as the digesting algorithm.

The example in this section corresponds to the file rsasign.c.

Computing a Digital Signature

Remember that with Crypto-C, you have the choice of doing your private-key operations normally or of using the blinding technique (see "Timing Attacks and Blinding" on page 96). You make this choice in the algorithm chooser. For normal signature operations, use AM_RSA_CRT_ENCRYPT. To use blinding, use AM_RSA_CRT_ENCRYPT_BLIND.

Step 1: Creating An Algorithm Object

Declare a variable to be B_ALGORITHM_OBJ. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for B_CreateAlgorithmObject:

```
B_ALGORITHM_OBJ digitalSigner = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&digitalSigner)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

Crypto-C provides three methods for computing RSA digital signatures: MD2 with

RSA encryption, MD5 with RSA encryption, and SHA1 with RSA encryption.

Note: Recent cryptanalytic work has discovered a collision in MD2's internal compression function, and there is some chance that the attack on MD2 may be extended to the full hash function. The same attack applies to MD. Another attack has been applied to the compression function on MD5, though this has yet to be extended to the full MD5. RSA Data Security, Inc., recommends that before you use MD, MD2, or MD5, you should consult the RSA Laboratories web site to be sure that their use is consistent with the latest information.

For this example, choose `AI_SHA1WithRSAEncryption`. The *Library Reference Manual* Chapter 2 entry on this AI states that the format of ***`info`*** supplied to `B_SetAlgorithmInfo` is `NULL_PTR`:

```
if ((status = B_SetAlgorithmInfo
     (digitalSigner, AI_SHA1WithRSAEncryption, NULL_PTR)) != 0)
  break;
```

Step 3: Init

Associate a key and algorithm method with the algorithm object through `B_SignInit`. The *Library Reference Manual* Chapter 4 entry for this function shows that it takes four arguments: the algorithm object, a key object, an algorithm chooser, and a surrender context. The algorithm object in this example is ***`digitalSigner`***. Remember, if the algorithm object was not set to `AI_MD5WithRSAEncryption`, `AI_MD2WithRSAEncryption`, `AI_SHA1WithRSAEncryption`, or their BER counterparts, you cannot use `B_SignInit`. For a key object, use an RSA private key. Follow Steps 1 through 5 of "Generating a Key Pair" on page 186 to produce a key pair. Remember, the modulus must be at least 368 bits.

Build an algorithm chooser with the AMs listed in the *Library Reference Manual* Chapter 2 entry for the AI in use:

```
B_ALGORITHM_METHOD *SIGN_SAMPLE_CHOOSER[] = {
  &AM_SHA,
  &AM_RSA_CRT_ENCRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};
```

Note: If you want to sign using the blinding technique to thwart timing attacks (see "Timing Attacks and Blinding" on page 96), use `AM_RSA_CRT_ENCRYPT_BLIND` in the algorithm chooser.

B_SignInit is fast, so it is reasonable to pass a properly cast NULL_PTR for the surrender context:

```
if ((status = B_SignInit
     (digitalSigner, privateKey, SIGN_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Digest the data to sign with B_SignUpdate, which is described in Chapter 4 of the *Library Reference Manual*. Unless there is an extraordinarily large amount of data (for example, a megabyte), this function is quick and a NULL_PTR for the surrender context should be no problem. Assuming you have your input data and you know its length, your call would be the following:

```
if ((status = B_SignUpdate
     (digitalSigner, inputData, inputDataLen,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

B_SignUpdate digested the data. Encrypt the digest and output the result to a signature buffer with B_SignFinal. The signature will be the same size as the public modulus, so make sure the output buffer is big enough. The *Library Reference Manual* Chapter 2 entry on AI_SHAWithRSAEncryption states that "You may pass (B_ALGORITHM_OBJ)NULL_PTR for all ***randomAlgorithm*** arguments." This function does not return immediately, so a surrender context can be helpful; for this example use the surrender context outlined in "The Surrender Context" on page 120:

```
#define BLOCK_SIZE 64;

/* Assuming we are using a 512-bit key */
unsigned char signature[BLOCK_SIZE];
unsigned int signatureLen;
```

```
/* generalFlag is for the surrender function. */
generalFlag = 0;
if ((status = B_SignFinal
     (digitalSigner, signature, &signatureLen, 64,
      (B_ALGORITHM_OBJ)NULL_PTR,
      &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

When you are done with all objects, remember to destroy them.

```
B_DestroyAlgorithmObject (&digitalSigner);
B_DestroyKeyObject (&privateKey);
```

Verifying a Digital Signature

The Crypto-C sequence `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal` will digest the original data, decrypt the signature with the provided RSA public key, and compare the digest to the decrypted signature. If the values are the same, `B_VerifyFinal` returns a 0; if they are different, it returns an error code.

Note: If a signing application did not digest the data before encrypting to produce a signature, you cannot use the `Verify` functions. Instead, decrypt the signature using `AI_PKCS_RSAPublic` or `AI_RSAPublic`.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ digitalVerifier = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&digitalVerifier)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

The signer should tell you which message digest and decryption algorithms you need to use to verify the signature. To verify the signature created above, you would use the same AI:

```
if ((status = B_SetAlgorithmInfo
     (digitalVerifier, AI_SHA1WithRSAEncryption, NULL_PTR)) != 0)
  break;
```

Step 3: Init

Associate a key and algorithm method with the algorithm object through `B_VerifyInit`. The Chapter 4 *Library Reference Manual* entry for this function shows that it takes four arguments: the algorithm object, a key object, an algorithm chooser, and a surrender context. The algorithm object in this example is ***`digitalVerifier`***. For a key object, use an RSA public key, presumably the partner to the RSA private key that was used for the signature. Build an algorithm chooser which incorporates the AMs listed in the *Library Reference Manual* Chapter 2 entry for the AI in use. `B_VerifyInit` is fast, so it is reasonable to pass a properly cast `NULL_PTR` for the surrender context:

```
B_ALGORITHM_METHOD *VERIFY_SAMPLE_CHOOSER[] = {
  &AM_SHA,
  &AM_RSA_DECRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_VerifyInit
     (digitalVerifier, publicKey, VERIFY_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Note: If the algorithm object was not set to `AI_MD5WithRSAEncryption`, `AI_MD2WithRSAEncryption`, `AI_SHA1WithRSAEncryption`, or their BER counterparts, you cannot use `B_VerifyInit`.

Step 4: Update

Use `B_VerifyUpdate` to digest the data that was signed. Its prototype is in Chapter 4 of the *Library Reference Manual*. Unless there is an extraordinarily large amount of data (for example, a megabyte), `B_VerifyUpdate` is quick and a `NULL_PTR` for the surrender context should be no problem. Assuming that you have the same input data and you know its length, your call is the following:

```
if ((status = B_VerifyUpdate
     (digitalVerifier, inputData, inputDataLen,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

`B_VerifyUpdate` digested the data. Decrypt the signature and compare the result to the digest with `B_VerifyFinal`. The *Library Reference Manual* Chapter 2 entry on `AI_SHA1WithRSAEncryption` states that “You may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all ***`randomAlgorithm`*** arguments.” This function does not return immediately, so use a surrender context:

```
/* generalFlag is for the surrender function. */
generalFlag = 0;
if ((status = B_VerifyFinal
     (digitalVerifier, signature, signatureLen,
      (B_ALGORITHM_OBJ)NULL_PTR,
      &generalSurrenderContext)) != 0)
  break;
```

The return value will be 0 if the signature verifies, nonzero if it does not. Of course, a nonzero return value may indicate some other error, so check any error return against the *Crypto-C Error Types*, in Appendix A of the *Library Reference Manual*.

Step 6: Destroy

When you are done with all objects, remember to destroy them:

```
B_DestroyAlgorithmObject (&digitalVerifier);
B_DestroyKeyObject (&publicKey);
```

ANSI X9.31-Compliant RSA Digital Signatures

Crypto-C supplies a special AI, `AI_SignVerify`, for ANSI X9.31-compliant digital signing and verification. The procedure to sign and verify using `AI_SignVerify` is similar to the steps outlined in the previous section “RSA Digital Signatures” on page 198. The steps that differ are shown below.

The example in this section corresponds to the file signver.c.

Computing A Digital Signature

Step 1: Creating an Algorithm Object

Create your algorithm object as in “Computing a Digital Signature” on page 199.

Step 2: Setting the Algorithm Object

Assume that RSA_MODULUS_BITS gives the modulus size of the RSA key pair. The proper AI to use for following the ANSI X9.31 standard for digital signatures is AI_SignVerify. The *Library Reference Manual* Chapter 2 entry for this AI states that you have to pass a pointer to a B_SIGN_VERIFY_PARAMS structure to B_SetAlgorithmInfo:

```
typedef struct {                                          /* Current Choices */
  unsigned char *encryptionMethodName;   /* "rsaSignX931", "rsaVerifyX931" */
  POINTER        encryptionParams;  /* Null for what is currently available*/
  unsigned char *digestMethodName;                                /* "sha1" */
  POINTER        digestParams;                             /* Null for sha1 */
  unsigned char *formatMethodName;                          /* "formatX931" */
  POINTER        formatParams; /* structure of type A_X931_PARAMS for sha1 */
} B_SIGN_VERIFY_PARAMS;
```

Currently, the only signing method supported is "rsaSignX931", the only digest available is "sha1", and the only format method is "formatX931". You can pass in a NULL_PTR for the encryption and digest parameters, but the ***formatParams*** field requires a pointer to a A_X931_PARAMS structure:

```
typedef struct {
  unsigned int blockLen;
  unsigned int oidNum;
} A_X931_PARAMS;
```

You need to determine ***blockLen*** for your modulus. AI_SignVerify encodes the input data in blocks. Because of the requirements of the underlying RSA algorithm, the number of bits of data must be the same as the number of bits of the RSA modulus. However, the input block size is measured in bytes. Because the modulus size, which is stored in RSA_MODULUS_BITS, may not be an even number of bytes, you need to calculate the smallest number of bytes you can use for your block. This number is the integer part of `(RSA_MODULUS_BITS + 7) / 8`. For example, if your modulus is 514 bits long, the smallest block size you can use is the integer part of (514 + 7)/8 bytes, or 65 bytes.

```
   A_X931_PARAMS x931params;
   B_SIGN_VERIFY_PARAMS signVerifyParams;
   x931params.blockLen = ((RSA_MODULUS_BITS + 7) / 8);
   x931params.oidNum = 3;

   signVerifyParams.encryptionMethodName = (unsigned char *)"rsaSignX931";
   signVerifyParams.encryptionParams = NULL_PTR;
   signVerifyParams.digestMethodName = (unsigned char *)"sha1";
   signVerifyParams.digestParams = NULL_PTR;
   signVerifyParams.formatMethodName = (unsigned char *)"formatX931";
   signVerifyParams.formatParams = (POINTER)&x931params;

   if ((status = B_SetAlgorithmInfo (digitalSigner, AI_SignVerify,
                                     (POINTER)&signVerifyParams)) != 0)
     break;
```

Note: For verifying, use `"rsaVerifyX931"` in place of `"rsaSignX931"`.

Step 3: Init

Associating a key and algorithm method is the same as in the previous example, but you need to include different algorithm methods in the chooser. The *Library Reference Manual* Chapter 2 entry for `AI_SignVerify` lists the appropriate ones to add:

```
B_ALGORITHM_METHOD *SIGNVERIFY_SAMPLE_CHOOSER[] = {
  &AM_SHA,
  &AM_SHA_RANDOM,
  &AM_RSA_STRONG_KEY_GEN,
  &AM_FORMAT_X931,
  &AM_RSA_CRT_X931_ENCRYPT,
  &AM_EXTRACT_X931,                     /*  We will use these two AMs */
  &AM_RSA_X931_DECRYPT,              /*  for verifying the signature  */
  (B_ALGORITHM_METHOD *)NULL_PTR
};
```

```
if ((status = B_SignInit
     (digitalSigner, privateKey, SIGNVERIFY_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Steps 4, 5, 6

All other steps remain the same as in the example “Computing a Digital Signature” on page 199.

Verifying A Digital Signature

Step 1: Creating An Algorithm Object

Create your algorithm object as in “Verifying a Digital Signature” on page 202.

Step 2: Setting the Algorithm Object

To verify the signature created above, you need to use the same AI you used for signing. Again, you must set up the appropriate structures containing the information for the algorithm you wish to use. The ***x931params*** structure is the same as the one used for signing, but you need to use `"rsaVerifyX931"` for the ***encryptionMethodName***.

```
signVerifyParams.encryptionMethodName = (unsigned char *)"rsaVerifyX931";
signVerifyParams.encryptionParams = NULL_PTR;
signVerifyParams.digestMethodName = (unsigned char *)"sha1";
signVerifyParams.digestParams = NULL_PTR;
signVerifyParams.formatMethodName = (unsigned char *)"formatX931";
signVerifyParams.formatParams = (POINTER)&x931params;

if ((status = B_SetAlgorithmInfo (digitalVerifier, AI_SignVerify,
                                  (POINTER)&signVerifyParams)) != 0)
  break;
```

Step 3: Init

Again, the only change required in the Init step is to include the appropriate algorithm methods in the chooser. These are the same methods included in the

`SIGNVERIFY_SAMPLE_CHOOSER` above. Then, call `B_VerifyInit` with the chooser:

```
    if ((status = B_VerifyInit
        (digitalVerifier, publicKey, SIGNVERIFY_SAMPLE_CHOOSER,
         (A_SURRENDER_CTX *)NULL_PTR)) != 0)
      break;
```

Steps 4, 5, 6

All other steps remain the same as in the example “Verifying a Digital Signature” on page 202.

Performing DSA Operations

The Digital Signature Algorithm (DSA) is part of the Digital Signature Standard (DSS), published by the National Institute of Standards and Technology (NIST, a division of the US Department of Commerce), and is the digital authentication standard of the US government. The section "Digital Signature Algorithm (DSA)" on page 58 gives a more detailed description of the actual algorithm.

Generating a DSA key pair is a two-step process. First, you must generate the DSA parameters, then you can generate the actual key pair.

The example in this section corresponds to the file `dsasign.c`.

Generating DSA Parameters

In this section, you generate the DSA parameters: a prime, a subprime, and a base. There is no Step 4, Update, in generating DSA parameters.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ dsaParamGenerator = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&dsaParamGenerator)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There is only one AI that will generate DSA parameters, `AI_DSAParamGen`. The format of ***info*** supplied to `B_SetAlgorithmInfo` is a pointer to the following:

```
typedef struct {
  unsigned int primeBits;                          /* size of prime in bits */
} B_DSA_PARAM_GEN_PARAMS;
```

Crypto-C will generate the prime, but you must decide how big that prime will be. The number of prime bits can be anywhere from 512 to 2048. Larger numbers provide greater security, but are also much slower. As with the RSA algorithm, RSA Data Security recommends using 768 bits. To save time, because this is for illustrative purposes only, this example will use 512. The subprime is always 160 bits long:

```
B_DSA_PARAM_GEN_PARAMS dsaParams;

dsaParams.primeBits = 512;
if ((status = B_SetAlgorithmInfo
     (dsaParamGenerator, AI_DSAParamGen,
      (POINTER)&dsaParams)) != 0)
  break;
```

Step 3: Init

Initialize the generation process with `B_GenerateInit`. Build an algorithm chooser. Because this function is quick, it is reasonable to pass `NULL_PTR` as the surrender context. Generating the parameters in Step 5 is time-consuming, though, so you will use a surrender context there:

```
B_ALGORITHM_METHOD *DSA_PARAM_GEN_CHOOSER[] = {
  &AM_SHA_RANDOM,
  &AM_DSA_PARAM_GEN,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_GenerateInit
     (dsaParamGenerator, DSA_PARAM_GEN_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

There is no Step 4 in generating DSA parameters.

Step 5: Generate

To generate DSA parameters, call the Crypto-C function `B_GenerateParameters`. The *Library Reference Manual* Chapter 4 entry for this call indicates there are four arguments. The first is the algorithm object that generates the parameters; in this example, that is ***dsaParamGenerator***.

The second is a result algorithm object. Crypto-C will generate some values and will need to place them somewhere. This information will be used in later Crypto-C calls, so you might as well place these values in an algorithm object now. Create an algorithm object, but do not set it; `B_GenerateParameters` will do that. (This is similar to generating an RSA key pair, where the results were placed into key objects.)

The third argument is a random algorithm. Complete Steps 1 through 4 of "Generating Random Numbers" on page 147. You do not need random bytes, only an algorithm that can generate them. The algorithm chooser you are using contains the AM for SHA1 random number generation.

The last argument is a surrender context. Generating DSA parameters can be time-consuming, sometimes taking two or three minutes. On slower machines, generating parameters over 800 bits can take more than an hour. Use the surrender context described previously. It will print out a dot every second to let you know that Crypto-C is computing and the machine has not crashed:

```
B_ALGORITHM_OBJ dsaKeyGenObj = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&dsaKeyGenObj)) != 0)
  break;

/* generalFlag is for this tutorial's surrender function. */
generalFlag = 0;
if ((status = B_GenerateParameters
     (dsaParamGenerator, dsaKeyGenObj, randomAlgorithm,
      &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy your objects. Do not destroy the ***`dsaKeyGenObj`*** object until you have used it to generate the actual key pair:

```
B_DestroyAlgorithmObject (&randomAlgorithm);
B_DestroyAlgorithmObject (&dsaParamGenerator);
```

Generating a DSA Key Pair

The previous code generated the DSA parameters and set an algorithm object. With that algorithm object, you can generate the key pair. Remember, the algorithm object has already been created and set, so you can jump directly to Step 3.

Step 3: Init

When it generated the parameters, Crypto-C set the algorithm object ***dsaKeyGenObj*** to AI_DSAKeyGen. That means that when you build an algorithm chooser for the Init call, you need to include AM_DSA_KEY_GEN. Look up the description and prototype for B_GenerateInit in Chapter 4 of the *Library Reference Manual*. For this example, you can use the following:

```
B_ALGORITHM_METHOD *DSA_KEY_GEN_CHOOSER[] = {
  &AM_SHA_RANDOM,
  &AM_DSA_KEY_GEN,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_GenerateInit
     (dsaKeyGenObj, DSA_KEY_GEN_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

This example uses NULL_PTR for the surrender context because B_GenerateInit is a speedy function. B_GenerateKeypair in Step 5 is the time-consuming function.

Step 4: Update

There is no Step 4 in generating a key pair.

Step 5: Generate

The description and prototype for B_GenerateKeypair in Chapter 4 of the *Library Reference Manual* show that this function takes five arguments. The first is the algorithm object; for this example, it is ***dsaKeyGenObj***. The second and third are key objects. For this call, all you have to do is create the key objects; they will be set by B_GenerateKeypair. The fourth argument is a random algorithm. For this, complete Steps 1 through 4 of "Generating Random Numbers" on page 147. You do not need random bytes, only an algorithm that can generate them. The algorithm chooser you are using (from Step 3) contains the AM for SHA1 random number generation. The last argument is the surrender context. This function call is quick; the lengthy portion was generating the parameters:

```
B_KEY_OBJ dsaPublicKey = (B_KEY_OBJ)NULL_PTR;
B_KEY_OBJ dsaPrivateKey = (B_KEY_OBJ)NULL_PTR;
```

```
if ((status = B_CreateKeyObject (&dsaPublicKey)) != 0)
  break;

if ((status = B_CreateKeyObject (&dsaPrivateKey)) != 0)
  break;

if ((status = B_GenerateKeypair
     (dsaKeyGenObj, dsaPublicKey, dsaPrivateKey,
      randomAlgorithm, (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

When you are done with all objects, remember to destroy them:

```
B_DestroyAlgorithmObject (&randomAlgorithm);
B_DestroyAlgorithmObject (&dsaKeyGenObj);
B_DestroyKeyObject (&dsaPublicKey);
B_DestroyKeyObject (&dsaPrivateKey);
```

DSA Signatures

In this section, we describe how to write Crypto-C code that computes or verifies DSA digital signatures. See “Authentication and Digital Signatures” on page 55 for information on what a digital signature is. For signing, Crypto-C offers `B_SignInit`, `B_SignUpdate`, and `B_SignFinal`, which will digest the data and create a signature using DSA with a private key. For verification, Crypto-C offers `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal` to digest the data again and check the signature using the DSA public key.

Computing a Digital Signature

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for

B_CreateAlgorithmObject:

```
B_ALGORITHM_OBJ dsaSigner = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&dsaSigner)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There is only one Crypto-C choice for computing DSA digital signatures, `AI_DSAWithSHA1` (or its BER counterpart). The *Library Reference Manual* Chapter 2 entry for this AI states that the format of ***info*** supplied to `B_SetAlgorithmInfo` is `NULL_PTR`.

```
if ((status = B_SetAlgorithmInfo
     (dsaSigner, AI_DSAWithSHA1, NULL_PTR)) != 0)
  break;
```

Step 3: Init

Associate a key and algorithm method with the algorithm object through `B_SignInit`. The Chapter 4 *Library Reference Manual* entry on this function shows that it takes four arguments: the algorithm object, a key object, an algorithm chooser and a surrender context. The algorithm object in this example is ***dsaSigner***. For a key object you want to use a DSA private key. See the previous section on generating a DSA key pair.

Build an algorithm chooser, the elements being the AMs listed in the *Library Reference Manual* Chapter 2 entry for the AI in use. `B_SignInit` is fast, so it is reasonable to pass a properly cast `NULL_PTR` for the surrender context:

```
B_ALGORITHM_METHOD *DSA_SIGN_CHOOSER[] = {
  &AM_SHA,
  &AM_DSA_SIGN,
  (B_ALGORITHM_METHOD *)NULL_PTR
};
if ((status = B_SignInit
     (dsaSigner, dsaPrivateKey, DSA_SIGN_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Digest the data to sign with B_SignUpdate, the prototype of which is in Chapter 4 of the *Library Reference Manual*. Unless there is an extraordinarily large amount of data (for example, a megabyte or more), this function is quick and a NULL_PTR for the surrender context should be no problem. Assuming you have some input data and you know its length, your call is the following:

```
if ((status = B_SignUpdate
     (dsaSigner, inputData, inputDataLen,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

B_SignUpdate digested the data. Create the signature and send the result to a signature buffer with B_SignFinal. The signature will be as many as 48 bytes long, so make sure the output buffer is big enough. The *Library Reference Manual* Chapter 2 entry on AI_DSAWithSHA1 states:

> You must pass a random algorithm in B_SignFinal, but may pass (B_ALGORITHM_OBJ)NULL_PTR for all other ***randomAlgorithm*** arguments.

This function does not return immediately, so a surrender context can be helpful. For this example, use the surrender context described in “The Surrender Context” on page 120:

```
#define MAX_SIG_LEN 48

unsigned char signature[MAX_SIG_LEN];
unsigned int signatureLen;

/* generalFlag is for the surrender function. */
generalFlag = 0;
if ((status = B_SignFinal
     (dsaSigner, signature, &signatureLen, MAX_SIG_LEN,
      randomAlgorithm,
      &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

When you are done with all objects, remember to destroy them:

```
B_DestroyAlgorithmObject (&dsaSigner);
B_DestroyKeyObject (&dsaPrivateKey);
```

Verifying a Digital Signature

The Crypto-C sequence `B_VerifyInit`, `B_VerifyUpdate`, and `B_VerifyFinal` digests the original data and checks the signature. If the signature is valid, `B_VerifyFinal` returns a zero; if the signature is not valid, it returns an error code.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ dsaVerifier = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&dsaVerifier)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

To verify the signature created above, use the same AI:

```
if ((status = B_SetAlgorithmInfo
     (dsaVerifier, AI_DSAWithSHA1, NULL_PTR)) != 0)
  break;
```

Step 3: Init

Associate a key and algorithm method with the algorithm object through `B_VerifyInit`. The Chapter 4 *Library Reference Manual* entry on this function shows that it takes four arguments: the algorithm object, a key object, an algorithm chooser, and a surrender context. The algorithm object in this example is ***`dsaVerifier`***. For a key object, you want to use a DSA public key, presumably the partner to the DSA private key used to sign. Build an algorithm chooser, the elements being the AMs listed in the

Library Reference Manual Chapter 2 entry for the AI in use. `B_VerifyInit` is fast, so it is reasonable to pass a properly cast `NULL_PTR` for the surrender context:

```
B_ALGORITHM_METHOD *DSA_VERIFY_CHOOSER[] = {
  &AM_SHA1,
  &AM_DSA_VERIFY,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_VerifyInit
     (dsaVerifier, dsaPublicKey, DSA_VERIFY_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Digest the data that was signed with `B_VerifyUpdate`, the prototype of which is in Chapter 4 of the *Library Reference Manual*. Unless there is an extraordinarily large amount of data (for example, a megabyte or more), this function is quick and a `NULL_PTR` for the surrender context will probably be no problem. Assuming you have the same input data and you know its length, your call is the following:

```
if ((status = B_VerifyUpdate
     (dsaVerifier, inputData, inputDataLen,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

`B_VerifyUpdate` digested the data. Check the signature with `B_VerifyFinal`. The *Library Reference Manual* Chapter 2 entry on `AI_DSAWithSHA1` states:

> You must pass a random algorithm in `B_SignFinal`, but may pass `(B_ALGORITHM_OBJ)NULL_PTR` for all other ***`randomAlgorithm`*** arguments.

This function does not return immediately, so use a surrender context:

```
/* generalFlag is for the surrender function. */
generalFlag = 0;
if ((status = B_VerifyFinal
     (dsaVerifier, signature, signatureLen,
      (B_ALGORITHM_OBJ)NULL_PTR,
      &generalSurrenderContext)) != 0)
  break;
```

The return value will be zero if the signature verifies, nonzero if it does not. Of course, a nonzero return value may indicate some other error, so check any error return against the *Crypto-C Error Types*, Appendix A of the *Library Reference Manual*.

Step 6: Destroy

When you are done with all objects, remember to destroy them:

```
B_DestroyAlgorithmObject (&dsaVerifier);
B_DestroyKeyObject (&dsaPublicKey);
```

Performing Diffie-Hellman Key Agreement

The Diffie-Hellman Key Agreement is a method for two parties to obtain the same symmetric key. In this procedure, a central authority generates parameters and gives them to the two individuals seeking to generate a secret key. In Phase 1, each individual uses these parameters to produce a public value and a private value. In Phase 2, they trade public values and each uses the other's public value with their own private value to generate the same secret value.

Note: One of the individuals could act as the central authority and generate the parameters. Security does not depend on a third party's independently producing the parameters.

The section "Diffie-Hellman Public Key Agreement" on page 61 gives a detailed description of the Diffie-Hellman algorithm.

Generating Diffie-Hellman Parameters

The parameters are a prime, a base, and, optionally, the length in bits of the private value. The parties will generate their own private values in Phase 1, although the central authority has the option of declaring how long these values will be.

Note: You may have noticed that the Diffie-Hellman algorithm is very similar to the RSA algorithm. The Diffie-Hellman prime is analogous to the RSA modulus, and the Diffie-Hellman base is analogous to the RSA data to encrypt. The Diffie-Hellman private value is analogous to the RSA private exponent (private key) in private-key encryption.

The example in this section corresponds to the file `dhparam.c`. There is no Step 4, Update, in generating Diffie-Hellman parameters.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ dhParamGenerator = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&dhParamGenerator)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There is only one AI for generating Diffie-Hellman parameters: `AI_DHParamGen`. The format of ***info*** supplied to `B_SetAlgorithmInfo` is a pointer to the following struct:

```
typedef struct {
  unsigned int primeBits;                /* size of prime modulus in bits */
  unsigned int exponentBits;            /* size of random exponent in bits */
} A_DH_PARAM_GEN_PARAMS;
```

Crypto-C will generate the prime, but you must decide how big that prime will be. As with the RSA modulus, the number of prime bits can be anywhere from 256 to 2048. Larger numbers provide greater security, but operations with larger numbers are much slower. RSA Data Security recommends 768. To save time, because this is for illustrative purposes only, this example will use 512.

The *exponent* is the private value, generated randomly by each party during Phase 1. The value ***exponentBits*** is the length of that private value. The Diffie-Hellman algorithm allows the parameter generator (the central authority) to optionally determine the length of the private value. Crypto-C exercises that option and requires the length.

The exponent length should be at least twice the general security level of the system. For instance, if 80-bit security against brute-force attack is desired, the exponent should be 160 bits long. (This is how DSS does it.) The prime length should be chosen to have a comparable level of difficulty against the best discrete logarithm algorithms. The relationship between the sizes changes from time to time; a 1024-bit prime would not be too far off from the 80-bit level.

The closer the exponent length is to the prime length, the longer it takes to generate the Diffie-Hellman parameters, because Crypto-C generates a prime p and a prime q where $p-1$ is a multiple of q, and the length of q is the same as the desired length of the exponent. If the lengths are very close it will take a long time to find an appropriately related pair of primes, because for a given q there won't be all that many possible p's. For example: for a one-bit difference between the prime and exponent lengths, p must equal $2q+1$, and it's unlikely that q and $2q+1$ are simultaneously prime.

The Chapter 2 entry for `AI_DHParamGen` notes that the "***exponentBits*** must be less than

primeBits." For this example, choose 512 prime bits and 504 exponent bits:

```
A_DH_PARAM_GEN_PARAMS dhParams;

dhParams.primeBits = 512;
dhParams.exponentBits = 504;
if ((status = B_SetAlgorithmInfo
     (dhParamGenerator, AI_DHParamGen,
      (POINTER)&dhParams)) != 0)
  break;
```

Step 3: Init

Initialize the generation process with `B_GenerateInit`. Build an algorithm chooser. Because this function is quick, it is reasonable to pass `NULL_PTR` as the surrender context. Generating the parameters in Step 5 is time-consuming, though, so you will use a surrender context there:

```
B_ALGORITHM_METHOD *DH_SAMPLE_CHOOSER[] = {
  &AM_SHA_RANDOM,
  &AM_DH_PARAM_GEN,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_GenerateInit
     (dhParamGenerator, DH_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

There is no Step 4 in generating Diffie-Hellman parameters.

Step 5: Generate

To generate Diffie-Hellman parameters, call the Crypto-C function `B_GenerateParameters`. The *Library Reference Manual* Chapter 4 entry for this call indicates there are four arguments. The first is the algorithm object that generates the parameters; in this example, that is ***dhParamGenerator***. The second is a result algorithm object. Crypto-C will generate some values and will need to place them somewhere. So you might as well place them into an algorithm object now. (This is similar to generating an RSA key pair, where the results were placed into key objects.) Create an algorithm object, but do not set it; `B_GenerateParameters` will do that.

The third argument is a random algorithm. Complete Steps 1 through 4 of "Generating Random Numbers" on page 147. You do not need random bytes, only an algorithm that can generate them. The algorithm chooser you are using contains the AM for SHA random number generation.

The last argument is a surrender context. Generating Diffie-Hellman parameters is time-consuming; it can take up to two minutes. On slower machines, generating parameters over 800-bits can take more than an hour. Use the surrender context mentioned above. It will print out a dot every second to let you know that Crypto-C is computing and the machine has not crashed:

```
B_ALGORITHM_OBJ dhParametersObj = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&dhParametersObj)) != 0)
  break;

/* generalFlag is for this tutorial's surrender function. */
generalFlag = 0;
if ((status = B_GenerateParameters
     (dhParamGenerator, dhParametersObj, randomAlgorithm,
      &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy your objects. Do not destroy the ***`dhParametersObj`*** object until you have passed it on to the parties executing the agreement. The next section discusses that point:

```
B_DestroyAlgorithmObject (&randomAlgorithm);
B_DestroyAlgorithmObject (&dhParamGenerator);
```

Distributing Diffie-Hellman Parameters

The central authority, after computing the parameters, must send this information to the parties seeking agree on a secret key. This can be done using Crypto-C format or BER-encoded format.

Note: It is not necessary to generate parameters each time two parties wish to agree on a secret key. Any number of key agreements can use the same parameters. Of course, for greater security, it is a good idea to generate new parameters every so often.

Crypto-C Format

To send the information in Crypto-C format, you can send a copy of the algorithm object to the participants. Actually, you do not send the object itself, but rather the "***info*** supplied to B_SetAlgorithmInfo."

Recall that you did not set the algorithm object ***dhParametersObj***; the Crypto-C function B_GenerateParameters did. It is set to the AI AI_DHKeyAgree. In the *Library Reference Manual* Chapter 2 entry on AI_DHKeyAgree, the topic "Format of ***info*** returned by B_GetAlgorithmInfo" states that it returns a pointer to an A_DH_KEY_AGREE_PARAMS structure:

```
typedef struct {
  ITEM        prime;                                  /* prime modulus */
  ITEM        base;                                 /* base generator */
  unsigned int exponentBits;          /* size of random exponent in bits */
} A_DH_KEY_AGREE_PARAMS;
```

where ITEM is:

```
typedef struct {
  unsigned char *data;
  unsigned int   len;
} ITEM;
```

Declare a variable to be a pointer to such a structure and pass its address as the argument.

Using the *Library Reference Manual* Chapter 4 prototype for B_GetAlgorithmInfo as a guide, you can write the following:

```
A_DH_KEY_AGREE_PARAMS *dhKeyAgreeParams =
     (A_DH_KEY_AGREE_PARAMS *)NULL_PTR;
```

```
if ((status = B_GetAlgorithmInfo
     ((POINTER *)&dhKeyAgreeParams, dhParametersObj,
      AI_DHKeyAgree)) != 0)
  break;
```

If you look at the elements of the `struct`:

> ***dhKeyAgreeParams***`->prime.data`
> ***dhKeyAgreeParams***`->prime.len`
> ***dhKeyAgreeParams***`->base.data`
> ***dhKeyAgreeParams***`->base.len`
> ***dhKeyAgreeParams***`->exponentBits`

you will see the parameters Crypto-C generated. This is the information the central authority sends to the participants in the key agreement. Copy this information to a file or diskette, for instance, and pass it on.

If you want to email the information, you will not be able to send the information over most email systems because the data is in binary form, not ASCII. Crypto-C offers encoding and decoding functions to convert between binary and ASCII. See "Converting Data Between Binary and ASCII" on page 154 for more information.

BER Format

There is a problem with distributing the parameters in the above structure. The `struct` is not standard; it is unique to Crypto-C. If one or both of the parties are not using Crypto-C, how do you give them the information? The standard is ASN.1, which defines Basic Encoding Rules (BER) and Distinguished Encoding Rules (DER). See "BER/DER Encoding" on page 125 for a description of this topic.

The central authority puts the parameters into DER format, encodes them, and emails the encoding. The parties decode the DER string and convert that information into the parameters in the format of their choice.

This sounds difficult, but Crypto-C offers a means of doing it simply. Above, in order to obtain the parameters, you used `B_GetAlgorithmInfo` with `AI_DHKeyAgree`. Chapter 2 of the *Library Reference Manual* lists `AI_DHKeyAgreeBER`, which states:

> **Format of *info* returned by** `B_GetAlgorithmInfo`**:**
> pointer to an `ITEM` structure which gives the address and length of the DER-encoded algorithm identifier.

Crypto-C returns a pointer to where that information resides, not the information. As soon as the object that contains that information is destroyed, the information will no longer be accessible. Therefore, once you get the pointer to that information, copy it into your own buffer:

```
ITEM *cryptocDHParametersBER;
ITEM myDHParametersBER;

myDHParametersBER.data = NULL_PTR;

if ((status = B_GetAlgorithmInfo
     ((POINTER *)&cryptocDHParametersBER, myDHParametersObj,
      AI_DHKeyAgreeBER)) != 0)
  break;

myDHParametersBER.len = cryptocDHParametersBER->len;
myDHParametersBER.data = T_malloc (myDHParametersBER.len);
if ((status = (myDHParametersBER.data == NULL_PTR)) != 0)
  break;
T_memcpy (myDHParametersBER.data, cryptocDHParametersBER->data,
          myDHParametersBER.len);
```

In summary, generate the parameters, get the algorithm ***info*** in BER format with `B_GetAlgorithmInfo` and `AI_DHKeyAgreeBER`, encode the BER data into ASCII format and send it to the Diffie-Hellman key agreement participants.

Note: The conversion into BER or DER is known as BER-encoding or DER-encoding, and the conversion between binary to ASCII is known as encoding and decoding. This may get confusing, but the word encoding without a BER in front of it generally means binary to ASCII. If the encoding is BER- or DER-encoding, the BER or DER should be explicitly stated.

Diffie-Hellman Key Agreement

If you are one of the parties involved in the key agreement, perform the following steps. Note that instead of Update and Final, you use `B_KeyAgreePhase1` and `B_KeyAgreePhase2`. Also, if you are writing an application that executes the Diffie-Hellman key agreement, the application must be interactive.

This process will produce an agreed-upon secret value. That value may be larger than necessary. For instance, the agreement may produce a 64-byte agreed upon secret value, yet the parties may need only 8 bytes. The application must determine which bytes from the agreed upon secret value to use.

The example in this section corresponds to the file `dhagree.c`.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ dhKeyAgreeAlg = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&dhKeyAgreeAlg)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There are two possible AIs to use in setting a Diffie-Hellman key agreement algorithm object: `AI_DHKeyAgree` and `AI_DHKeyAgreeBER`. Recall that in generating the Diffie-Hellman parameters, the central authority set an algorithm object and then retrieved its ***info*** using `B_GetAlgorithmInfo`. The central authority then distributed that ***info*** to you, telling you which AI to use. For this example, use `AI_DHKeyAgreeBER` to match the usage in "Distributing Diffie-Hellman Parameters" on page 222:

```
/* Assume you received the BER-encoded DH parameters from the
     central authority in the ITEM dhParametersBER. */
ITEM dhParametersBER;

if ((status = B_SetAlgorithmInfo
     (dhKeyAgreeAlg, AI_DHKeyAgreeBER,
      (POINTER)&dhParametersBER)) != 0)
  break;
```

Step 3: Init

Initialize the algorithm object with `B_KeyAgreeInit`. The *Library Reference Manual* Chapter 4 entry on this function indicates it takes four arguments. The first is the algorithm object, ***dhKeyAgreeAlg***. The second is a key object. The Diffie-Hellman key agreement algorithm does not require a key, so use a properly cast `NULL_PTR` for this argument. The third argument is an algorithm chooser, and the last is a surrender context. This function is fast, so it is reasonable to pass a properly cast `NULL_PTR` for the surrender context.

```
B_ALGORITHM_METHOD *DH_AGREE_SAMPLE_CHOOSER[] = {
  &AM_DH_KEY_AGREE,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_KeyAgreeInit
     (dhKeyAgreeAlg, (B_KEY_OBJ)NULL_PTR, DH_AGREE_SAMPLE_CHOOSER,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Phase 1

In Phase 1, you generate a random private value and compute a public value from that private value and the parameters. The *Library Reference Manual* Chapter 4 entry on `B_KeyAgreePhase1` describes the format of its six arguments.

The first is the algorithm object. The second is output. This output is the public value, which will be the same size as the prime. You are responsible for allocating the memory for the buffer to contain the public value. In this example, you do not know how big the prime is; just set the algorithm with the BER-encoded ***info***. That ***info*** does contain the size of the prime, but you would have to know exactly where to look. An easier way to find the prime size is by getting the algorithm ***info*** as `AI_DHKeyAgree`.

The third argument for the Phase 1 call is the address of an `unsigned int`. Crypto-C will place the length in bytes of the public value at that address. The fourth is the size of the buffer you allocated; if the buffer is not big enough to hold the output, Crypto-C will generate an error. The fifth argument is a random algorithm object. For this, complete Steps 1 through 4 of "Generating Random Numbers" on page 147. You do not need random bytes, only an algorithm that can generate them. The last argument is a surrender context. This function does not return immediately, so a surrender context is helpful. Use the one outlined in "The Surrender Context" on page 120:

```
unsigned char *myPublicValue = NULL_PTR;
unsigned int myPublicValueLen;
A_DH_KEY_AGREE_PARAMS *getParams;
```

```
/* Find out how big the prime is so we know how many bytes to
   allocate for the public value buffer.  */

if ((status = B_GetAlgorithmInfo
     ((POINTER *)&getParams, dhKeyAgreeAlg, AI_DHKeyAgree)) != 0)
  break;

myPublicValue = T_malloc (getParams->prime.len);
if ((status = (myPublicValue == NULL_PTR)) != 0)
  break;

/* generalFlag is for the surrender function.*/
generalFlag = 0;
if ((status = B_KeyAgreePhase1
     (dhKeyAgreeAlg, myPublicValue, &myPublicValueLen,
      getParams->prime.len, randomAlgorithm,
      &generalSurrenderContext)) != 0)
  break;
```

Step 5: Phase 2

After you have computed your public value, you must send it off to the other party and receive their public value. You need the same algorithm object from Phase 1 to complete Phase 2. This is why the process must be interactive. You cannot save your private value and stop the program after sending off your public value while you wait for the other party's public value.

The input of `B_KeyAgreePhase2` is the other party's public value; the output is the agreed-upon secret value. The output will be the same size as the prime; you must allocate the space to hold this output. Although the output will be at least 32 bytes, the parties might only need eight bytes for a session key. If that is the case, it is the application's responsibility to specify which bytes of the agreed-upon secret value will be used. This function does not return immediately, so a surrender context is useful:

```
/* The other party should send their public value and its length.  */

unsigned char *otherPublicValue;
unsigned int otherPublicValueLen;
unsigned char *agreedUponSecretValue = NULL_PTR;
unsigned int agreedUponSecretValueLen;

agreedUponSecretValue = T_malloc (getParams->prime.len);
if ((status = (agreedUponSecretValue == NULL_PTR)) != 0)
  break;

/* generalFlag is for the surrender function.*/
generalFlag = 0;
if ((status = B_KeyAgreePhase2
     (dhKeyAgreeAlg, agreedUponSecretValue,
      &agreedUponSecretValueLen, getParams->prime.len,
      otherPublicValue, otherPublicValueLen,
      &generalSurrenderContext)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all objects and free up any allocated memory:

```
B_DestroyAlgorithmObject (&dhKeyAgreeAlg);
B_DestroyAlgorithmObject (&randomAlgorithm);
T_free (myPublicValue);
T_free (agreedUponSecretValue);
```

Performing Elliptic Curve Operations

Elliptic curve cryptosystems can be used for a number of public-key operations. Crypto-C supports the following elliptic curve features:

- Generation of elliptic curve parameters
- Elliptic curve key pair generation
- Elliptic Curve Signature Schemes (ECDSA)
- Elliptic Curve Authenticated Encryption Scheme (ECAES)
- Elliptic Curve Diffie-Hellman key agreement (ECDH)

Crypto-C also allows you to generate precomputed acceleration tables to speed up certain elliptic curve operations.

For a description of elliptic curve parameters and algorithms, see “Elliptic Curve Cryptography” on page 64.

Generating Elliptic Curve Parameters

Before you can perform any elliptic curve operations, you must create the parameters for the curve that you will be using. Once you have generated elliptic curve parameters, you can use the parameters to: generate a key pair, to create an acceleration table, or to perform Elliptic Curve Diffie-Hellman (ECDH) key agreement. The same elliptic curve parameters can be used for multiple operations. See “Elliptic Curve Parameters” on page 65 for more information.

You need to make some choices about the kind of elliptic curve you want to use. You need to choose what to use for a base field: an odd prime finite field or a field of even characteristic. If you choose a field of even characteristic, you also have to choose what type of basis you want to use. You also have to choose the number of bits that you want for the length of an element in the field.

For this example, you will use an odd prime field for the base field. The example in this section corresponds to the file `ecparam.c`.

Step 1: Creating an Algorithm Object

You need to create two algorithm objects. The first, ***paramGenObj***, is initialized by the programmer prior to the parameter generation operation; it is used to hold information necessary to generate parameters. The second, ***ecParamsObj***, is set and initialized by `B_GenerateParameters`; it will hold the newly-generated elliptic curve parameters.

```
B_ALGORITHM_OBJ paramGenObj = (B_ALGORITHM_OBJ)NULL_PTR;
B_ALGORITHM_OBJ ecParamsObj = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject(&paramGenObj)) != 0)
  break;
if ((status = B_CreateAlgorithmObject(&ecParamsObj)) != 0)
  break;
```

Step 2: Setting the Algorithm Object

You need to set the algorithm object that will be used to generate the elliptic curve parameters. The only AI that can be used to generate elliptic curve parameters is `AI_ECParamGen`. Chapter 2 in the *Library Reference Manual* gives the following:

Format of *info* supplied to `B_SetAlgorithmInfo`**:**
pointer to a `B_EC_PARAM_GEN_PARAMS` structure.

To supply the necessary information, pass a pointer to a `B_EC_PARAM_GEN_PARAMS` structure as the third argument to `B_SetAlgorithmInfo`. The `B_EC_PARAM_GEN_PARAMS` structure is defined in the Chapter 2 entry in the *Library Reference Manual* for `AI_ECParamGen`:

```
typedef struct {
  unsigned int version;                          /* implementation version */
  unsigned int fieldType;              /* base field for the elliptic curve */
  unsigned int fieldElementBits;         /* length of field element in bits */
  unsigned int compressIndicator; /* controls field element representation */
  unsigned int minOrderBits;     /* minimum size of group generated by base */
                             /* input of 0 defaults to fieldElementBits - 7 */
  unsigned int trialDivBound;       /* maximum size of second largest prime */
                                   /*  subgroup of group generated by base  */
                                             /*  input of 0 defaults to 255 */
  unsigned int tableLookup;            /* characteristic 2 only. Set if the */
                                   /*  use of precomputed params is desired */
} B_EC_PARAM_GEN_PARAMS;
```

You must choose the field type and the length of the field element. The field type can be either: a prime field of odd characteristic, that is, F_p; or a field of even characteristic, F_{2^m}.

For this example, set the arguments as shown below. The first argument specifies the version number; in Crypto-C, the only version available is 0. The second argument specifies that you want your base field to be of the form F_p (p is an odd prime).

The third argument sets the length of a field element in bits; in this example, set it to be 160. For the prime field case, the size of a field element can be anywhere from 64 to 384 bits. The length of a field element, along with ***minOrderBits***, strongly affects the security of the system; the greater the length, the greater the security. However, the greater the length, the longer it takes to generate key pairs and encrypt and decrypt. Currently, RSA Data Security recommends a size of 160 to 170 bits for ***minOrderBits*** for prototyping and evaluation; because ***minOrderBits*** defaults to 7 bits smaller than ***fieldElementBits***, ***fieldElementBits*** should be set to 167–177 bits.

For the legal values for ***fieldElementBits*** in the even characteristic case, see the entry for `AI_ECParamGen` in Chapter 2 of the *Library Reference Manual*.

Note: Generating an elliptic curve for even characteristic without table lookup (***fieldtype*** = `FT_F2_ONB` or `FT_F2_POLYNOMIAL` and ***tableLookup*** = 0) can be extremely time-consuming, taking several hours in some cases. In general, larger values for ***minOrderBits*** means longer times for curve generation. Therefore, if you wish to generate curves for even characteristic, but do not want to use table lookup, you can speed curve generation by setting a smaller value for ***minOrderBits***. Remember, however, that the size of ***minOrderBits*** is directly tied to the security of your elliptic curve cryptosystem. Setting ***minOrderBits*** allows you to make a trade-off between the time it takes to generate curves and the security of your system.

The fourth argument specifies whether you will express the base and public key in uncompressed or hybrid form; pass `CI_NO_COMPRESS` to indicate that your application will not use compression. For the fifth and six arguments, pass 0; this tells Crypto-C to use its internal algorithms to generate its own values:

```
  B_EC_PARAM_GEN_PARAMS paramGenInfo;
  paramGenInfo.version = 0;
  paramGenInfo.fieldType = FT_FP;
  paramGenInfo.fieldElementBits = 160;
  paramGenInfo.compressIndicator = CI_NO_COMPRESS;
  paramGenInfo.minOrderBits = 0;
  paramGenInfo.trialDivBound = 0;

if ((status = B_SetAlgorithmInfo(paramGenObj, AI_ECParamGen,
                                  (POINTER)&paramGenInfo)) != 0)
    break;
```

Step 3: Init

You can pass a NULL_PTR for the surrender context, because B_GenerateInit is a speedy function. For AI_ECParamGen, Chapter 2 of the *Library Reference Manual* indicates which algorithm methods you need to include in your chooser, ***paramGenChooser***:

Algorithm methods to include in application's algorithm chooser:
AM_ECFP_PARAM_GEN for odd prime fields and AM_ECF2POLY_PARAM_GEN for even characteristic.

Because you are using an odd prime, use AM_ECFP_PARAM_GEN:

```
B_ALGORITHM_METHOD *paramGenChooser[] = {
    &AM_ECFP_PARAM_GEN,
    &AM_ECF2POLY_PARAM_GEN,
    (B_ALGORITHM_METHOD *)NULL_PTR
  };

  if ((status = B_GenerateInit(paramGenObj, paramGenChooser,
                               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
```

Step 4: Update

No Update step is necessary for parameter generation.

Step 5: Generate

This function may take a while, so you should use a surrender function. See "The Surrender Context" on page 120. B_GenerateParameters places the newly-generated elliptic curve parameters in ***ecParamsObj***:

```
  generalSurrenderContext.Surrender = GeneralSurrenderFunction;
  generalSurrenderContext.handle = (POINTER)&generalFlag;
  generalSurrenderContext.reserved = NULL_PTR;
  generalFlag = 0;

if ((status = B_GenerateParameters(paramGenObj, ecParamsObj,
                                    randomAlgorithm,
                                    &generalSurrenderContext)) != 0)
    break;
```

Step 6: Destroy

Destroy all algorithm objects that are no longer necessary. However, do not destroy ***ecParamsObj*** until you have retrieved and stored the parameters. See “Retrieving Elliptic Curve Parameters” on page 234 for more information. Do destroy ***ecParamsObj*** when it is no longer needed:

```
B_DestroyAlgorithmObject (&paramGenObj);
B_DestroyAlgorithmObject (&randomAlgorithm);
```

Retrieving Elliptic Curve Parameters

Once you have your elliptic curve parameters in an algorithm object, you need to be able to retrieve those parameters in an accessible form. Once you have retrieved your parameters, you can store the information or print it out. You also need to retrieve the elliptic curve parameters from the algorithm object when you generate acceleration tables.

This section outlines two application-specific procedures, ***AllocAndCopyECParamInfo*** and ***FreeECParamInfo***, that are used to retrieve and store information. These procedures are referred to in subsequent sections.

To retrieve information from an algorithm object, it is necessary to call `B_GetAlgorithmInfo` with an appropriate AI. The only AI listed in the *Library Reference Manua*l that allows you to set or retrieve the parameters is `AI_ECParameters`:

Type of information this allows you to use:
the parameters generated by executing `AI_ECParamGen` for either generating keys or executing key agreements.

The *Library Reference Manual* Chapter 2 entry for `AI_ECParameters` also states that the format of the information returned by `B_GetAlgorithmInfo` is a pointer to an

A_EC_PARAMS structure:

```
typedef struct {
  unsigned int version;                           /* implementation version */
  unsigned int fieldType;                   /* indicates type of base field */
  ITEM         fieldInfo;                          /*  It is the prime number */
                                          /* in case that fieldType = FT_FP; */
                 /* the basis polynomial if fieldType = FT_F2_POLYNOMIAL; */
                  /* and the degree of the field if fieldType = FT_F2_ONB */
  ITEM         coeffA;                         /* elliptic curve coefficient */
  ITEM         coeffB;                         /* elliptic curve coefficient */
  ITEM         base;                      /* elliptic curve group generator */
  ITEM         order;            /* order of subgroup's generating element */
  ITEM         cofactor;                    /* the cofactor of the subgroup */
  unsigned int compressIndicator; /* controls field element representation */
  unsigned int fieldElementBits;              /* field element size in bits */
} A_EC_PARAMS;
```

Assume that the elliptic curve parameters are placed in the algorithm object ***ecParamsObj*** (see "Generating Elliptic Curve Parameters" on page 230). Make the appropriate call to B_GetAlgorithmInfo:

```
  A_EC_PARAMS *cryptocECParamInfo;

  if ((status = B_GetAlgorithmInfo((POINTER *)&cryptocECParamInfo,
                                   ecParamsObj, AI_ECParameters)) != 0)
    break;
```

Note that ***cryptocECParamInfo*** is a pointer to the information, not the information itself. The memory that ***cryptocECParamInfo*** points to belongs to Crypto-C; another call to Crypto-C may alter or destroy it. Therefore, once you get the pointer to the information, you must copy it to your own buffer.

The following procedure, ***AllocAndCopyECParamInfo***, is an example of an application-specific procedure that allocates space to store the parameters. You can also write your own procedure to satisfy the needs of your application:

```
int AllocAndCopyECParamInfo(output, input)
A_EC_PARAMS *output;
A_EC_PARAMS *input;
{
  int status;
```

```
do {
  output->version = input->version;

  output->fieldType = input->fieldType;

  output->fieldInfo.len = input->fieldInfo.len;
  output->fieldInfo.data = T_malloc(output->fieldInfo.len);
  if ((status = (output->fieldInfo.data == NULL_PTR)) != 0)
    break;
  T_memcpy(output->fieldInfo.data, input->fieldInfo.data,
           output->fieldInfo.len);

  output->coeffA.len = input->coeffA.len;
  output->coeffA.data = T_malloc(output->coeffA.len);
  if ((status = (output->coeffA.data == NULL_PTR)) != 0)
    break;
  T_memcpy(output->coeffA.data, input->coeffA.data,
           output->coeffA.len);

  output->coeffB.len = input->coeffB.len;
  output->coeffB.data = T_malloc(output->coeffB.len);
  if ((status = (output->coeffB.data == NULL_PTR)) != 0)
    break;
  T_memcpy(output->coeffB.data, input->coeffB.data,
           output->coeffB.len);

  output->base.len = input->base.len;
  output->base.data = T_malloc(output->base.len);
  if ((status = (output->base.data == NULL_PTR)) != 0)
    break;
  T_memcpy(output->base.data, input->base.data,
           output->base.len);

  output->order.len = input->order.len;
  output->order.data = T_malloc(output->order.len);
  if ((status = (output->order.data == NULL_PTR)) != 0)
    break;
  T_memcpy(output->order.data, input->order.data,
           output->order.len);
    output->cofactor.len = input->cofactor.len;
    output->cofactor.data = T_malloc(output->cofactor.len);
    if ((status = (output->cofactor.data == NULL_PTR)) != 0)
      break;
    T_memcpy(output->cofactor.data, input->cofactor.data,
             output->cofactor.len);

    output->compressIndicator = input->compressIndicator;

    output->fieldElementBits = input->fieldElementBits;
  } while(0);

  if (status != 0)
    printf("AllocAndCopyECParamInfo failed with status %i\n", status);

  return status;
}
```

For this example application, use ***AllocAndCopyECParamInfo()*** to make a copy of the information that ***cryptocECParamInfo*** points to and place that in your own buffer, ***ecParamInfo***:

```
  A_EC_PARAMS ecParamInfo;

  if ((status = AllocAndCopyECParamInfo(&ecParamInfo,
                                        cryptocECParamInfo)) != 0)
    break;
```

When the information in ***ecParamInfo*** is no longer needed, you must remember to free any memory that you allocated:

```
  FreeECParamInfo(&ecParamInfo);
```

where ***FreeECParamInfo*** is a procedure that performs this operation. In the sample code, ***FreeECParamInfo*** is implemented as follows:

```
void FreeECParamInfo(ecParams)
A_EC_PARAMS *ecParams;
{
  T_free(ecParams->fieldInfo.data);
  T_free(ecParams->coeffA.data);
  T_free(ecParams->coeffB.data);
  T_free(ecParams->base.data);
  T_free(ecParams->order.data);
  T_free(ecParams->cofactor.data);
}
```

Generating an Elliptic Curve Key Pair

In this section, you will generate a key pair suitable for use with Elliptic Curve DSA (ECDSA) and the Elliptic Curve Authenticated Encryption Scheme (ECAES).

You can optionally use an acceleration table to speed up the key generation operation. This is useful if you will be doing key generation with the same elliptic curve several times. If you will be using an acceleration table with this example, assume that you have gone through the steps of generating an acceleration table and that you have the table in the `ITEM` structure ***accelTableItem***.

Step 1: Create

Create the algorithm object that you will use to generate the key pair:

```
  B_ALGORITHM_OBJ ecKeyGen = (B_ALGORITHM_OBJ)NULL_PTR;

  if ((status = B_CreateAlgorithmObject (&ecKeyGen)) != 0)
    break;
```

Also create the key objects to hold the keys after they have been generated:

```
  B_KEY_OBJ publicKey = (B_KEY_OBJ)NULL_PTR;
  B_KEY_OBJ privateKey = (B_KEY_OBJ)NULL_PTR;
  if ((status = B_CreateKeyObject (&publicKey)) != 0)
    break;
  if ((status = B_CreateKeyObject (&privateKey)) != 0)
    break;
```

Step 2: Set

The *Library Reference Manual* indicates that the appropriate AI to use for generating an elliptic curve key pair is `AI_ECKeyGen`. You must set the algorithm object with the parameter information for the elliptic curve that you are using to generate the key. You do this by providing `B_SetAlgorithmInfo` with a pointer to a `B_EC_PARAMS` structure.

```
typedef struct {
  B_INFO_TYPE parameterInfoType;
  POINTER parameterInfoValue;
} B_EC_PARAMS;
```

Place the elliptic curve parameters in the `A_EC_PARAMS` structure ***ecParamInfo***. You can do this either by setting ***ecParamInfo*** with the appropriate values, or by following the steps outlined in "Retrieving Elliptic Curve Parameters" on page 234 to retrieve the parameters from an algorithm object and place them into an `A_EC_PARAMS` structure.

The AI that describes data in this format is `AI_ECParameters`:

```
  B_EC_PARAMS paramInfo;

  paramInfo.parameterInfoType = AI_ECParameters;
  paramInfo.parameterInfoValue = (POINTER)&ecParamInfo;

  if ((status = B_SetAlgorithmInfo (ecKeyGen, AI_ECKeyGen,
                                    (POINTER)&paramInfo)) != 0)
    break;
```

You can also optionally use the acceleration table to speed up key generation. See "Generating Acceleration Tables" on page 243 for more information. Assume that you have the acceleration table corresponding to your elliptic curve in the `ITEM` structure ***accelTableItem***. The appropriate AI to use with `B_SetAlgorithmInfo` in this case is `AI_ECAcceleratorTable`. Pass in a pointer to the `ITEM` structure holding the acceleration table as the third argument to `B_SetAlgorithmInfo`. Now set your key-generation algorithm object with the acceleration table information:

```
if ((status = B_SetAlgorithmInfo (ecKeyGen, AI_ECAcceleratorTable,
                                  (POINTER)&accelTableItem)) != 0)
  break;
```

Step 3: Initialize

Here you can pass a `NULL_PTR` for the surrender context, because `B_GenerateInit` is a speedy function. The *Library Reference Manual* entry on `AI_ECKeyGen` indicates which algorithm methods you need to include in your chooser, ***keyGenChooser***:

```
B_ALGORITHM_METHOD *keyGenChooser[] = {
  &AM_ECFP_KEY_GEN,
  &AM_ECF2POLY_KEY_GEN,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_GenerateInit (ecKeyGen, keyGenChooser,
                              (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

There is no Update step for key generation.

Step 5: Generate

Now you can complete the key-generation operation. Note that you must pass in a properly-initialized random algorithm as the fourth argument:

```
if ((status = B_GenerateKeypair
              (ecKeyGen, publicKey, privateKey, randomAlgorithm,
               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Remember to destroy all key objects and algorithm objects once they are no longer needed:

```
B_DestroyAlgorithmObject(&ecKeyGen);
B_DestroyAlgorithmObject(&randomAlgorithm);
B_DestroyKeyObject(&publicKey);
B_DestroyKeyObject(&privateKey);
```

Retrieving an Elliptic Curve Key

If you need to store or transport information about your elliptic curve keys, you need to be able to retrieve the key information from an algorithm object. This section outlines the steps needed to retrieve information for a public key. The steps for retrieving a private key are similar.

You need to call `B_GetKeyInfo` with the appropriate KI. The *Library Reference Manual* describes two KIs for use with elliptic curve public keys: `KI_ECPublic` and `KI_ECPublicComponent`. However, `KI_ECPublicComponent` does not supply the elliptic curve parameters, which must be associated with any elliptic curve key. Therefore, you can only use `KI_ECPublicComponent` if you only need the public component, for example, if you have already retrieved the appropriate EC parameters. Therefore, for this example, you'll use `KI_ECPublic`.

`KI_ECPublic` gives a pointer to an `A_EC_PUBLIC_KEY` structure:

```
typedef struct {
  ITEM       publicKey;                                /* public component */
  A_EC_PARAMS curveParams;    /* the underlying elliptic curve parameters */
} A_EC_PUBLIC_KEY;
```

After you have your public key information in the key object ***publicKey***, make a call to `B_GetKeyInfo`. See “Generating an Elliptic Curve Key Pair” on page 238 for more information:

```
A_EC_PUBLIC_KEY *cryptocPublicKeyInfo;

if ((status = B_GetKeyInfo((POINTER *)&cryptocPublicKeyInfo,
                           *publicKey, KI_ECPublic)) != 0)
  break;
```

`B_GetKeyInfo` gives a pointer to memory, but this memory is owned by Crypto-C. If you want to store this information, you need to make your own copy of the information because another call to Crypto-C may modify the memory owned by Crypto-C. The routines ***AllocAndCopyECPubKeyInfo*** and ***FreeECPubKeyInfo*** given below retrieve and store the key information. These routines are used in the sample code for building public-key acceleration tables.

AllocAndCopyECPubKeyInfo takes as input a pointer to an A_EC_PUBLIC_KEY structure containing memory belonging to Crypto-C. It copies the information from the structure owned by Crypto-C to an A_EC_PUBLIC_KEY structure created by the application and outputs a pointer to the structure just created. The memory allocated with ***AllocAndCopyECPubKeyInfo*** should be freed using ***FreeECPubKeyInfo*** when appropriate:

```
int AllocAndCopyECPubKeyInfo(output, input)
A_EC_PUBLIC_KEY *output;
A_EC_PUBLIC_KEY *input;

{
  int status;

  do {
    output->publicKey.len = input->publicKey.len;
    output->publicKey.data = T_malloc(output->publicKey.len);
    if ((status = (output->publicKey.data == NULL_PTR)) != 0)
      break;
    T_memcpy(output->publicKey.data, input->publicKey.data,
             output->publicKey.len);

    if ((status = AllocAndCopyECParamInfo(&(output->curveParams),
                                          &(input->curveParams))) != 0)
      break;
  } while(0);

  if (status != 0)
    printf("AllocAndCopyECPubKeyInfo failed with status %i\n", status);

  return status;
}    /*  end AllocAndCopyECPubKeyInfo  */
```

FreeECPubKeyInfo takes a pointer to an A_EC_PUBLIC_KEY structure that contains space that was allocated by ***AllocAndCopyECPubKeyInfo*** and calls T_malloc to free all allocated data:

```
/*  This procedure takes a pointer to an A_EC_PUBLIC_KEY structure containing
 *  space allocated by AllocAndCopyECPubKeyInfo and frees all data allocated
 *  with T_malloc.  */

void FreeECPubKeyInfo(pubKey)
A_EC_PUBLIC_KEY *pubKey;
{
  T_free(pubKey->publicKey.data);
  FreeECParamInfo(&(pubKey->curveParams));
}    /*  end FreeECPubKeyInfo  */
```

Generating Acceleration Tables

An acceleration table stores precomputed versions of certain values that are frequently used during some elliptic curve operations. Acceleration tables can speed up certain elliptic curve operations. However, this increase in speed comes at the cost of space, as these tables tend to be very large.

There are two types of acceleration tables in Crypto-C:

- *Generic acceleration table:* stores values that are commonly used in many elliptic-curve operations, including key-pair generation, Elliptic Curve Diffie-Hellman key agreement, and ECDSA signing and verifying.
- *Public-key acceleration table:* stores all the values stored by the generic acceleration table, as well as additional values commonly used only in ECDSA verification.

The examples in this section are in the file `eparam.c`.

Generating a Generic Acceleration Table

This acceleration table can be used to speed up key-pair generation, public-key encryption, Elliptic Curve Diffie-Hellman key agreement, and ECDSA signing and verifying. This table is most useful if these operations are performed repeatedly with the same elliptic curve. The function `BuildAccelTable`, used in the sample code and defined in the file `ecparam.c`, demonstrates the following steps in creating the acceleration table.

Step 1: Create

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in

Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
    B_ALGORITHM_OBJ buildTable = (B_ALGORITHM_OBJ)NULL_PTR;

    if ((status = B_CreateAlgorithmObject(&buildTable)) != 0)
      break;
```

Step 2: Set

Step 2a: Retrieve the elliptic curve parameters

Because you are generating an acceleration table corresponding to a particular elliptic curve, you need to retrieve the elliptic curve parameters and place them in the algorithm object. Assume that you have gone through the steps to generate an elliptic curve and you have stored the parameters in the algorithm object ***ecParamsObj***. See "Retrieving Elliptic Curve Parameters" on page 234 for more details:

```
  A_EC_PARAMS *cryptocECParamInfo;
  A_EC_PARAMS ecParamInfo;

  if ((status = B_GetAlgorithmInfo((POINTER *)&cryptocECParamInfo,
                                   ecParamsObj, AI_ECParameters)) != 0)
    break;

   if ((status = AllocAndCopyECParamInfo(&ecParamInfo,
                                        cryptocECParamInfo)) != 0)
     break;
```

Step 2b: Format the information

You must put the information you retrieved into the proper format. The *Library Reference Manual* Chapter 2 entry for `AI_ECBuildAcceleratorTable` says that you must supply a pointer to a `B_EC_PARAMS` structure to `B_SetAlgorithmInfo`:

```
typedef struct {
  B_INFO_TYPE parameterInfoType;
  POINTER     parameterInfoValue;
} B_EC_PARAMS;
```

The first field in this structure, ***parameterInfoType***, is used to interpret the elliptic curve parameter information you supply in the second field, ***parameterInfoValue***. The

EC parameter information you have is an A_EC_PARAMS structure containing the data that describes the EC parameters. The B_INFO_TYPE that is used to properly interpret that information is AI_ECParameters.

Set the ***parameterInfoType*** field to AI_ECParameters and give the ***parameterInfoValue*** field a pointer to the location of the A_EC_PARAMS structure:

```
B_EC_PARAMS paramInfo;
paramInfo.parameterInfoType = AI_ECParameters;
paramInfo.parameterInfoValue = (POINTER)&ecParamInfo;

 if ((status = B_SetAlgorithmInfo
   (buildTable, AI_ECBuildAcceleratorTable,(POINTER)&paramInfo)) != 0)
  break;
```

Step 3: Init

In this step, you must supply the appropriate algorithm methods through the algorithm chooser. The *Library Reference Manual* Chapter 2 entry for AI_ECBuildAcceleratorTable indicates which AMs you must include in your chooser. This step doesn't take much time to complete, so you can pass in a NULL_PTR for your surrender context:

```
B_ALGORITHM_METHOD *ecAccelChooser[] = {
  &AM_ECFP_BLD_ACCEL_TABLE,                    /* for odd prime field */
  &AM_ECF2POLY_BLD_ACCEL_TABLE,         /* for characteristic 2 field */
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_BuildTableInit(buildTable, ecAccelChooser,
                               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

There is no Update step for building acceleration tables.

Step 5: Final

Step 5a: Allocate memory

You must allocate sufficient memory to hold the acceleration table. According to the

Library Reference Manual, you can use `B_BuildTableGetBufSize` to tell how much space will be required to store the acceleration table:

```
    ITEM accelTableItem;
    unsigned int maxTableLen;

    if ((status = B_BuildTableGetBufSize(buildTable, &maxTableLen)) != 0)
      break;

    accelTableItem.data = T_malloc(maxTableLen);

    if ((status = (accelTableItem.data == NULL_PTR)) != 0)
      break;
```

Step 5b: Build the acceleration table

Finally, build the acceleration table and store it in an `ITEM` structure. You store it this way for convenience—when you actually use the acceleration table, you will have to provide it in an `ITEM` structure to `B_SetAlgorithmInfo`. Building an acceleration table can take a lot of time, so use a surrender context. See “The Surrender Context” on page 120 for more information:

```
    ITEM accelTableItem;

    generalSurrenderContext.Surrender = GeneralSurrenderFunction;
    generalSurrenderContext.handle = (POINTER)&generalFlag;
    generalSurrenderContext.reserved = NULL_PTR;
    generalFlag = 0;

    if ((status = B_BuildTableFinal(buildTable, accelTableItem.data,
                                    &(accelTableItem.len), maxTableLen,
                                    &generalSurrenderContext)) != 0)
      break;
```

Step 6: Destroy

You must free all allocated memory and destroy all objects when they are no longer needed so that all sensitive information is zeroized and freed:

```
  T_memset(accelTableItem.data, 0, accelTableItem.len);
  T_free(accelTableItem.data);
  B_DestroyAlgorithmObject(&buildTable);
```

Public-Key Acceleration Table

This special-purpose acceleration table can be used to speed up ECDSA verification. Again, the cost in time to generate the table and space to store it must be weighed against the speedup in verification that it will provide. This table is most useful if ECDSA verification will be performed repeatedly with the same public key. The function `BuildPubKeyAccelTable`, used in the sample code and defined in the file `ecparam.c`, demonstrates the steps in creating the public-key acceleration table.

Step 1: Create

Create the algorithm object that will be used in building the public-key acceleration table. Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ buildTable = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject(&buildTable)) != 0)
  break;
```

Step 2: Set

Retrieve the public-key information and place it in the algorithm object used to build the acceleration table for that public key.

Step 2a: Retrieve the public key information

Because `B_GetKeyInfo` returns a pointer to memory that belongs to Crypto-C, you must make a copy of this information. See “Retrieving an Elliptic Curve Key” on page 241 for the definitions of ***`AllocAndCopyECPubKeyInfo`*** and ***`FreeECPubKeyInfo`***. Of course, you can write your own versions of these procedures to satisfy the needs of your application:

```
A_EC_PUBLIC_KEY *cryptocPublicKeyInfo;
A_EC_PUBLIC_KEY publicKeyInfo;

if ((status = B_GetKeyInfo((POINTER *)&cryptocPublicKeyInfo,
                           *publicKey, KI_ECPublic)) != 0)
  break;
```

```
    if ((status = AllocAndCopyECPubKeyInfo(&publicKeyInfo,
                                           cryptocPublicKeyInfo)) != 0)
      break;
```

When the information is no longer needed, don't forget to free the allocated memory:

```
FreeECPubKeyInfo(&publicKeyInfo);
```

Step 2b: Put the information retrieved in the proper format

To build the public-key acceleration table, use `AI_ECBuildPubKeyAccelTable`. The *Library Reference* Chapter 2 entry for `AI_ECBuildPubKeyAccelTable` states that you must supply a pointer to a `B_EC_PARAMS` structure. The procedure you use to fill this structure in is the same as the one you used to build the generic acceleration table. However, because you are building an acceleration table based on the public key, you must also pass in information about the public key.

You have an `A_EC_PUBLIC_KEY` struct containing the public key information, so the appropriate `B_INFO_TYPE` to use is `AI_ECPubKey`. According to the *Library Reference Manual* entry on `AI_ECPubKey`, you should pass `B_SetAlgorithmInfo` a pointer to `A_EC_PUBLIC_KEY` structure. Set the ***parameterInfoType*** to `AI_ECPubKey` and give ***parameterInfoValue*** the pointer to your `A_EC_PUBLIC_KEY` structure ***publicKeyInfo***.

```
    B_EC_PARAMS paramInfo;

    paramInfo.parameterInfoType = AI_ECPubKey;
    paramInfo.parameterInfoValue = (POINTER)&publicKeyInfo;

    if ((status = B_SetAlgorithmInfo(buildTable, AI_ECBuildPubKeyAccelTable,
                                     (POINTER)&paramInfo)) != 0)
      break;
```

Step 3: Init

In order to initialize the proper algorithms, you must supply an algorithm chooser with the appropriate algorithm methods. See the *Library Reference Manual* Chapter 2 entry for `AI_BuildPubKeyAccelTable` for a list of the appropriate AMs to include in the chooser:

```
    B_ALGORITHM_METHOD *ecAccelChooser[] = {
      &AM_ECFP_BLD_PUB_KEY_ACC_TAB,
      &AM_ECF2POLY_BLD_PUB_KEY_ACC_TAB,
      (B_ALGORITHM_METHOD *)NULL_PTR
    };

    if ((status = B_BuildTableInit(buildTable, ecAccelChooser,
                                   (A_SURRENDER_CTX *)NULL_PTR)) != 0)
      break;
```

Step 4: Update

There is no Update step for building acceleration tables.

Step 5: Final

Step 5a: Allocate memory

You must allocate sufficient memory to hold the acceleration table. Use `B_BuildTableGetBufSize` to obtain the maximum size of the public key acceleration table. Then allocate enough space to hold the table:

```
    ITEM pubKeyAccelTableItem;
    unsigned int maxTableLen;

    if ((status = B_BuildTableGetBufSize(buildTable, &maxTableLen)) != 0)
      break;

    pubKeyAccelTableItem.data = T_malloc(maxTableLen);

    if ((status = (pubKeyAccelTableItem.data == NULL_PTR)) != 0)
      break;
```

Step 5b: Build the public-key acceleration table

It can take a while to generate the table, so use a surrender function. See “The

Surrender Context" on page 120 for more information:

```
    ITEM pubKeyAccelTableItem;

    generalSurrenderContext.Surrender = GeneralSurrenderFunction;
    generalSurrenderContext.handle = (POINTER)&generalFlag;
    generalSurrenderContext.reserved = NULL_PTR;
    generalFlag = 0;

    if ((status = B_BuildTableFinal
                  (buildTable, pubKeyAccelTableItem.data,
                  &(pubKeyAccelTableItem.len), maxTableLen,
                  &generalSurrenderContext)) != 0)
      break;
```

Step 6: Destroy

Zeroize and free all sensitive information when it is no longer needed:

```
  T_memset(pubKeyAccelTableItem.data, 0, pubKeyAccelTableItem.len);
  T_free(pubKeyAccelTableItem.data);
  B_DestroyAlgorithmObject(&buildTable);
```

Performing EC Diffie-Hellman Key Agreement

Performing elliptic curve key agreement is similar to the ordinary Diffie-Hellman key agreement scheme, which allows two parties to obtain the same symmetric key. First, the two parties seeking to generate a secret key need to agree on the elliptic curve parameters. The parameters can be generated by a central authority or by the parties themselves.

The example in this section corresponds to the file `ecdh.c`. In this example, the two parties who wish to derive the same secret key are Alice and Bob. Both parties need to be provided with the same parameters:

```
  B_ALGORITHM_OBJ ecParamsObj = (B_ALGORITHM_OBJ)NULL_PTR;
```

In order to initialize ***ecParamsObj*** with a set of parameters describing an elliptic curve, follow the steps in the section "Generating Elliptic Curve Parameters" on page 230. Assume that these steps have been successfully completed and ***ecParamsObj*** contains the common parameters for Alice and Bob. Put the elliptic curve parameters in the A_EC_PARAMS structure, ***ecParams***. For an implementation of an application-specific procedure, ***AllocAndCopyECParamInfo***, which retrieves and stores the parameters, see "Retrieving Elliptic Curve Parameters" on page 234:

```
A_EC_PARAMS ecParams;
A_EC_PARAMS *cryptocECParams;

if ((status = B_GetAlgorithmInfo((POINTER *)&cryptocECParams, alice,
                                  AI_ECParameters)) != 0)
  break;

if ((status = AllocAndCopyECParamInfo(&ecParams, cryptocECParams)) != 0)
  break;
```

You will walk through the steps that Alice goes through, keeping in mind that Bob, perhaps in another application, is performing the same steps.

Note: If this key agreement operation is performed several times with the same parameters, you may wish to use the acceleration table. See "Generating Acceleration Tables" on page 243 for more information.

Step 1: Create

Create the algorithm object which you will use to perform the key agreement:

```
B_ALGORITHM_OBJ alice = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject(&alice)) != 0)
  break;
```

Step 2: Set

Set the algorithm object with the information necessary to perform the operation. AI_EC_DHKeyAgree, when used as the second argument to B_SetAlgorithmInfo, takes as the third argument a pointer to a B_EC_PARAMS structure:

```
typedef struct {
  B_INFO_TYPE parameterInfoType;
  POINTER     parameterInfoValue;
} B_EC_PARAMS;
```

Because you have the EC parameters in the A_EC_PARAMS structure ecParams, the appropriate AI that describes the data is AI_ECParameters:

```
B_EC_PARAMS commonECParams;
commonECParams.parameterInfoType = AI_ECParameters;
commonECParams.parameterInfoValue = (POINTER)&ecParams;

if ((status = B_SetAlgorithmInfo(alice, AI_EC_DHKeyAgree,
                                 (POINTER)&commonECParams)) != 0)
  break;
```

Step 2b (optional): Set Acceleration Table Info

If you are using an acceleration table, you need to set the algorithm object with the appropriate acceleration table. Once you have gone through the steps in "Generating Acceleration Tables" on page 243 and have an ITEM structure containing the acceleration table, you can pass a pointer to the ITEM structure as the third argument to B_SetAlgorithmInfo:

```
if ((status = B_SetAlgorithmInfo (alice, AI_ECAcceleratorTable,
                                  (POINTER)&aTableItem)) != 0)
  break;
```

Step 3: Initialize

Initialize the algorithm object to perform the key agreement protocol. The *Library Reference Manual* Chapter 2 entry for AI_EC_DHKeyAgree states which algorithm methods to include in your chooser:

```
B_ALGORITHM_METHOD *EC_DH_CHOOSER[] = {
  &AM_ECFP_DH_KEY_AGREE,
  &AM_ECF2POLY_DH_KEY_AGREE,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_KeyAgreeInit(alice, (B_KEY_OBJ)NULL_PTR, EC_DH_CHOOSER,
                             (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

You must allocate space to hold the results of Phase 1 and Phase 2. The largest size of Phase 1 output you can get is one byte larger than twice the field element size. For

Phase 2, the size of the output should be the same as the field element size. (See the *Library Reference Manual* Chapter 2 entry for `AI_EC_DHKeyAgree` for details.)

You can get the field element size using Alice's elliptic curve parameters. Since you have the parameters in the `A_EC_PARAMS` structure ***ecParams***, look at the ***fieldElementBits*** field, which gives you the required information. A simple manipulation gives you the field element length in bytes:

```
unsigned int fieldElementLen, maxPhase1Len, maxPhase2Len;

fieldElementLen = (ecParams->fieldElementBits + 7) / 8;
maxPhase1Len = (fieldElementLen * 2);
maxPhase2Len = fieldElementLen;
```

Step 4: Phase 1

During this phase, each party computes a private value and a public value. The private value is secret and currently cannot be accessed though the Crypto-C API. The public value should be transported to the other party. Note that you will have to supply a properly initialized random algorithm as the fifth argument to `B_KeyAgreePhase1`:

```
unsigned char *alicePublicValue = NULL_PTR;
unsigned int alicePublicValueLen;
alicePublicValue = T_malloc(maxPhase1Len);

if ((status = (alicePublicValue == NULL_PTR)) != 0)
  break;

if ((status = B_KeyAgreePhase1(alice, alicePublicValue,
                               &alicePublicValueLen, maxPhase1Len,
                               randomAlgorithm,
                               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Phase 2

By the time you have reached this step, Alice and Bob have exchanged public values. Assume that the pointer ***bobPublicValue*** points to Bob's public value and

bobPublicValueLen gives the length of Bob's public value:

```
unsigned char *bobPublicValue;
unsigned int bobPublicValueLen;
```

Using Bob's public value, Alice can compute the secret key that she and Bob will use to communicate with each other:

```
unsigned char *aliceSecretValue = NULL_PTR;
unsigned int aliceSecretValueLen;
aliceSecretValue = T_malloc(maxPhase2Len);

if ((status = (aliceSecretValue == NULL_PTR)) != 0)
  break;

if ((status = B_KeyAgreePhase2(alice, aliceSecretValue,
                               &aliceSecretValueLen, maxPhase2Len,
                               bobPublicValue, bobPublicValueLen,
                               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Always destroy key objects and algorithm objects once they are no longer needed:

```
T_free (alicePublicValue);
T_free (aliceSecretValue);
B_DestroyAlgorithmObject(&randomAlgorithm);
B_DestroyAlgorithmObject(&alice);
```

Performing ECDSA

The Elliptic Curve Digital Signature Agreement (ECDSA) is an elliptic curve analogue of DSA. To sign an arbitrarily long message with the elliptic curve version of DSA, you can use `AI_EC_DSAWithDigest`. First, you need to generate parameters for an elliptic curve and a key pair from that curve. Then, you will specify a digest algorithm for use with ECDSA in signing the message. Currently, the only digest algorithm supported for this operation is SHA1.

The example in this section corresponds to the file `ecdsadig.c`.

Generating EC Parameters

See the section "Generating Elliptic Curve Parameters" on page 230 for the steps you must complete to generate a new curve. You will need a properly initialized pseudo-random number generator. Assume that the function ***InitializeRandomAlgorithm*** goes through Steps 1-4 in "Generating Random Numbers" on page 147. Also, assume that the function ***InitializeECParamsObj*** goes through the steps in "Generating Elliptic Curve Parameters" on page 230 to generate new parameters and place them in ***ecParamsObj***:

```
B_ALGORITHM_OBJ randomAlgorithm = (B_ALGORITHM_OBJ)NULL_PTR;
B_ALGORITHM_OBJ ecParamsObj = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = InitializeRandomAlgorithm (&randomAlgorithm)) != 0)
  break;

if ((status = InitializeECParamsObj (&ecParamsObj,
                                     &randomAlgorithm)) != 0)
  break;
```

Now you have a properly initialized random algorithm object, ***randomAlgorithm***, and an algorithm object, ***ecParamsObj***, containing the parameters that describe the elliptic curve that you are going to use.

Generating an EC Key Pair

You also need to generate a public and private key. See "Generating an Elliptic Curve Key Pair" on page 238 for the required steps. To complete those steps, you will need a properly initialized random algorithm, the parameters describing an elliptic curve, and optionally the acceleration table corresponding to that curve:

```
B_KEY_OBJ publicKey = (B_KEY_OBJ)NULL_PTR;
B_KEY_OBJ privateKey = (B_KEY_OBJ)NULL_PTR;

if ((status = GenerateECKeys (&publicKey, &privateKey,
                              &ecParamsObj, &randomAlgorithm) != 0)
```

Assume that the steps in "Generating an Elliptic Curve Key Pair" on page 238 have been completed and that ***publicKey*** and ***privateKey*** are ready to be used.

Computing a Digital Signature

Step 1: Create

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
  B_ALGORITHM_OBJ ecDSASign = (B_ALGORITHM_OBJ)NULL_PTR;

  if ((status = B_CreateAlgorithmObject (&ecDSASign)) != 0)
    break;
```

Step 2: Set

The appropriate AI to use is `AI_EC_DSAWithDigest`. According to the entry in the *Library Reference Manual*, you have to provide a pointer to a `B_DIGEST_SPECIFIER` structure to `B_SetAlgorithmInfo`:

```
typedef struct {
  B_INFO_TYPE digestInfoType;
  POINTER     digestInfoParams;
} B_DIGEST_SPECIFIER;
```

Currently, the only digest algorithm supported is SHA1. This does not require any parameters, so specify `NULL_PTR` for ***digestInfoParams***:

```
  B_DIGEST_SPECIFIER digestInfo;
  digestInfo.digestInfoType = AI_SHA1;
  digestInfo.digestInfoParams = NULL_PTR;

  if ((status = B_SetAlgorithmInfo (ecDSASign, AI_EC_DSAWithDigest,
                                    (POINTER)&digestInfo)) != 0)
    break;
```

Step 2b (optional): Set Acceleration Table Info

```
  ITEM aTableItem;
```

Go through the steps in the section "Generating Acceleration Tables" on page 243 to create an acceleration table, placing the table information in ***aTableItem***:

```
  if ((status = B_SetAlgorithmInfo (ecDSASign, AI_ECAcceleratorTable,
                                    (POINTER)&aTableItem)) != 0)
    break;
```

Step 3: Init

Build an algorithm chooser with the appropriate AMs:

```
  B_ALGORITHM_METHOD *EC_DSA_CHOOSER[] = {
    &AM_SHA,
    &AM_ECFP_DSA_SIGN,
    &AM_ECF2POLY_DSA_SIGN,
    &AM_ECFP_DSA_VERIFY,
    &AM_ECF2POLY_DSA_VERIFY,
    (B_ALGORITHM_METHOD *)NULL_PTR
  };
```

Now you can associate your private key and your algorithm chooser with the algorithm object:

```
  if ((status = B_SignInit (ecDSASign, privateKey, EC_DSA_CHOOSER,
                            (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
```

Step 4: Update

Now, using `B_SignUpdate`, pass in the data to be signed:

```
  unsigned char *dataToSign = "Some arbitrarily long piece of data to
sign...";
  unsigned int dataToSignLen = strlen(dataToSign) + 1;
  if ((status = B_SignUpdate (ecDSASign, dataToSign, dataToSignLen,
                              (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
```

Step 5: Final

First you must allocate space to store the signature. The output of the ECDSA signature is the BER encoding of a sequence of two integers, (*r*,*s*). At most, the size of the output will be six bytes more than twice the length of the order. Retrieve the field element length from ***ecParamsObj*** and do a simple manipulation to find the field element length in bytes.

```
A_EC_PARAMS *ecParamInfo;
unsigned int order, maxSignatureLen;
unsigned char *signature;

if ((status = B_GetAlgorithmInfo ((POINTER *)&ecParamInfo, ecParamsObj,
                                  AI_ECParameters)) != 0)
  break;

order = (ecParamInfo->order.len + 7) / 8;
maxSignatureLen = (2 * order) + 6;
signature = T_malloc(maxSignatureLen);
 if ((status = (signature == NULL_PTR)) != 0)
  break;
```

Now, finalize the process and retrieve the signature. Note that the *Library Reference Manual* entry for `AI_EC_DSAWithDigest` indicates that you will have to pass in a properly initialized random algorithm in `B_SignFinal`:

```
unsigned int signatureLen;

if ((status = B_SignFinal (ecDSASign, signature, &signatureLen,
                           maxSignatureLen, randomAlgorithm,
                           (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Destroy all objects that are no longer needed:

```
B_DestroyAlgorithmObject(&ecDSASign);
B_DestroyKeyObject(&privateKey);
```

Verifying a Digital Signature

To verify the signature, you must go through a similar procedure. At the end, if the signature is valid, `B_VerifyFinal` returns 0. If it is not valid, `B_VerifyFinal` will return an error.

Step 1: Create

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ ecDSAVerify = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&ecDSAVerify)) != 0)
  break;
```

Step 2: Set

Use the same AI and ***digestInfo*** as you did for signing:

```
if ((status = B_SetAlgorithmInfo (ecDSAVerify, AI_EC_DSAWithDigest,
                                  (POINTER)&digestInfo)) != 0)
  break;
```

Step 2b (Optional): Set Public Key Acceleration Table Info

You can use either the public key acceleration table or the generic acceleration table to accelerate ECDSA verification. Verification using the public key acceleration table is faster than verification using only the generic acceleration table.

```
ITEM pubKeyAccelTableItem;
```

Go through the steps in the section “Generating Acceleration Tables” to create a generic acceleration table, placing the table information in ***aTableItem***:

```
if ((status = B_SetAlgorithmInfo (ecDSAVerify, AI_ECAcceleratorTable,
                                  (POINTER)&pubKeyAccelTableItem)) != 0)
  break;
```

Step 3: Init

Associate a key with the algorithm object and provide a chooser that contains the necessary algorithm methods:

```
if ((status = B_VerifyInit (ecDSAVerify, publicKey, EC_DSA_CHOOSER,
                            (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Pass in the original message. It will be internally digested to make a new signature that can be compared with the signature received by `B_VerifyFinal`:

```
if ((status = B_VerifyUpdate (ecDSAVerify, dataToSign, dataToSignLen,
                              (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

Pass in the signature that was received with the message. `B_VerifyFinal` returns 0 if the signature verifies, or an error if it is an invalid signature:

```
if ((status = B_VerifyFinal (ecDSAVerify, signature, signatureLen,
                             (B_ALGORITHM_OBJ)NULL_PTR,
                             (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 6: Destroy

Destroy all objects that are no longer needed:

```
T_free(signature);
B_DestroyAlgorithmObject(&ecParamsObj);
B_DestroyAlgorithmObject(&ecDSAVerify);
B_DestroyKeyObject(&publicKey);
```

Using ECAES

You can use the Elliptic Curve Authenticated Encryption System (ECAES) to perform public-key encryption. The example in this section corresponds to the file `eces.c`.

You will encrypt the following:

```
  unsigned char *dataToEncrypt = “Encrypt this arbitrarily long sentence
using ECAES!”;

  unsigned int dataToEncryptLen = sizeof(dataToEncrypt) + 1;
```

Using Elliptic Curve Parameters

See the section “Generating Elliptic Curve Parameters” on page 230 for the steps you must complete to generate a new curve. You need a properly initialized pseudo-random number generator. Assume that the function ***InitializeRandomAlgorithm*** goes through Steps 1 through 4 in the section “Generating Random Numbers” on page 147. Also assume that the function ***InitializeECParamsObj*** generates new parameters and places them in ***ecParamsObj***, following the steps in “Using Elliptic Curve Parameters” on page 261:

```
  B_ALGORITHM_OBJ randomAlgorithm = (B_ALGORITHM_OBJ)NULL_PTR;
  B_ALGORITHM_OBJ ecParamsObj = (B_ALGORITHM_OBJ)NULL_PTR;

  if ((status = InitializeRandomAlgorithm (&randomAlgorithm)) != 0)
    break;
  if ((status = InitializeECParamsObj (&ecParamsObj,
                                       &randomAlgorithm)) != 0)
    break;
```

You now have a properly initialized random algorithm object, ***randomAlgorithm***, and an algorithm object, ***ecParamsObj***, containing the parameters that describe the elliptic curve that you will use.

Using an EC Key Pair

Before you can encrypt, you need to generate a public/private key pair. As described in “Using an EC Key Pair” on page 261, key generation requires a properly initialized random algorithm and the parameters describing an elliptic curve, both of which you have created in the previous step:

```
  B_KEY_OBJ publicKey = (B_KEY_OBJ)NULL_PTR;
  B_KEY_OBJ privateKey = (B_KEY_OBJ)NULL_PTR;
```

Assume that the steps in "Using an EC Key Pair" have been completed and that ***publicKey*** and ***privateKey*** are ready to be used.

ECAES Public-Key Encryption

Once you have gone through the preliminary steps of generating your elliptic curve parameters and creating your public/private key pair, you are ready to encrypt your message.

Step 1: Create

First, create the algorithm object that will hold the information necessary to perform the encryption operation:

```
B_ALGORITHM_OBJ ecESEncrypt = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&ecESEncrypt)) != 0)
  break;
```

Step 2: Set

Associate the elliptic curve encryption AI, `AI_EC_ES`, with the algorithm object. According to the *Library Reference Manual* Chapter 2 entry for `AI_EC_ES`, you should pass `NULL_PTR` as the third argument to `B_SetAlgorithmInfo`:

```
if ((status = B_SetAlgorithmInfo
              (ecESEncrypt, AI_EC_ES, NULL_PTR)) != 0)
  break;
```

Step 2b (optional) Acceleration Table

You can use an acceleration table containing precomputed values to speed up encryption. Because users frequently perform encryption, it is worth while to use the acceleration table whenever the required memory is available.

To use the acceleration table, assume you have gone through the steps in "Generating a Generic Acceleration Table" on page 243 and placed the information in ***accelerationTableItem***:

```
ITEM accelerationTableItem;
```

Now, pass this information into your algorithm object:

```
if ((status = B_SetAlgorithmInfo
             (ecESEncrypt, AI_ECAcceleratorTable,
              (POINTER)&accelerationTableItem)) != 0)
  break;
```

Step 3: Init

You must initialize the algorithm object to perform encryption. You also need to provide the key that will be used for encryption. The algorithm chooser should contain the encryption algorithm methods listed in the *Library Reference Manual* for `AI_EC_ES`:

```
B_ALGORITHM_METHOD *EC_CHOOSER[] = {
  &AM_ECFP_ENCRYPT,
  &AM_ECF2POLY_ENCRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_EncryptInit (ecESEncrypt, publicKey, EC_CHOOSER,
                             (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

To update, first find the field element length in bytes. Remember that earlier, in “Using Elliptic Curve Parameters” on page 261, you placed the elliptic curve parameters in your algorithm object, ***ecParamsObj***. You can use this object to retrieve the field element length:

```
A_EC_PARAMS *ecParamInfo;
unsigned int fieldElementLen;

if ((status = B_GetAlgorithmInfo ((POINTER *)&ecParamInfo, ecParamsObj,
                                  AI_ECParameters)) != 0)
  break;

fieldElementLen = (ecParamInfo->fieldElementBits + 7) / 8;
```

Next, you must allocate space to hold the encrypted data. According to the *Library*

Reference Manual Chapter 2 entry for AI_EC_ES, the length of the encrypted data will be as much as (21 + 2 · (the size of a field element in bytes) + (length of input in bytes)) bytes.

```
unsigned int maxEncryptedDataLen;
unsigned int outputLenUpdate;

maxEncryptedDataLen = 21 + (2 * fieldElementLen) = dataToEncryptLen;
encryptedData = T_malloc(maxEncryptedDataLen);
if ((status = (encryptedData == NULL_PTR)) != 0)
  break;

if ((status = B_EncryptUpdate
              (ecESEncrypt, encryptedData, &outputLenUpdate,
               maxEncryptedDataLen, dataToEncrypt, dataToEncryptLen,
               (B_ALGORITHM_OBJ)NULL_PTR,
               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

```
unsigned int outputLenFinal, outputLenTotal;

if ((status = B_EncryptFinal
              (ecESEncrypt, encryptedData + outputLenUpdate,
               &outputLenFinal, maxEncryptedDataLen - outputLenUpdate,
               randomAlgorithm, (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

outputLenTotal = outputLenUpdate + outputLenFinal;
```

Step 6: Destroy

Destroy all objects that are no longer needed. Also, be sure to zeroize and free any allocated memory when it is no longer needed.

```
B_DestroyAlgorithmObject (&ecESEncrypt);
B_DestroyKeyObject (&publicKey);
T_free (encryptedData);
```

ECAES Private-Key Decryption

The steps for decryption are similar to those for encryption.

Step 1: Create

Create an algorithm object:

```
B_ALGORITHM_OBJ ecESDecrypt = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&ecESDecrypt)) != 0)
  break;
```

Step 2: Set

Associate the algorithm object with AI_EC_ES and pass NULL_PTR as the third argument:

```
if ((status = B_SetAlgorithmInfo
              (ecESDecrypt, AI_EC_ES, NULL_PTR)) != 0)
  break;
```

Step 3: Init

At this point, commit your algorithm object to perform decryption with a particular private key. Be sure that EC_CHOOSER contains the appropriate algorithm methods:

```
B_ALGORITHM_METHOD *EC_CHOOSER[] = {
  &AM_ECFP_DECRYPT,
  &AM_ECF2POLY_DECRYPT,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

 if ((status = B_DecryptInit (ecESDecrypt, privateKey, EC_CHOOSER,
                              (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Since you know that the length of the plaintext can't be larger than the length of the ciphertext, you'll use this approximation to allocate space for the decrypted data:

```
unsigned char *decryptedData;
unsigned int maxDecryptedDataLen;
unsigned int outputLenUpdate;

maxDecryptedDataLen = outputLenTotal;     /* Use the outputLenTotal from */
                                            /* Step 5 of ECAES encryption */
decryptedData = T_malloc(maxDecryptedDataLen);
if ((status = (decryptedData == NULL_PTR)) != 0)
  break;

if ((status = B_DecryptUpdate
              (ecESDecrypt, decryptedData, &outputLenUpdate,
               maxDecryptedDataLen, encryptedData, outputLenTotal,
               (B_ALGORITHM_OBJ)NULL_PTR,
               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 5: Final

```
unsigned int outputLenFinal, outputLenTotal;

if ((status = B_DecryptFinal
              (ecESDecrypt, decryptedData + outputLenUpdate,
               &outputLenFinal, maxDecryptedDataLen - outputLenUpdate,
               (B_ALGORITHM_OBJ)NULL_PTR,
               (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;

outputLenTotal = outputLenUpdate + outputLenFinal;
```

Step 6: Destroy

Destroy any objects that are no longer needed. Also, be sure to zeroize and free any allocated memory when it is no longer needed.

```
B_DestroyAlgorithmObject (&ecESDecrypt);
B_DestroyKeyObject (&privateKey);
T_free (decryptedData);
```

Chapter 7

Secret Sharing Operations

Secret Sharing

Secret sharing allows a system to require a certain number of “shares” to retrieve a secret. The process encrypts information and then creates a number of shares of the encrypted information. The information can be recovered by collecting a declared number (called the *threshold*) of shares. Note that the threshold must be less than or equal to the total number of shares.

Typically, the secret is a key used for encrypting sensitive data. For example, you might protect an RC2 key with a secret-sharing algorithm, creating four shares, and set the threshold to two. Then any two of the four shares can reconstruct the RC2 key.

Generating Shares

Crypto-C offers the Bloom-Shamir secret sharing method. For this implementation, the minimum total number of shares is two and the maximum is 255; the threshold must be less than or equal to the total number of shares. The 255 limit is not part of the Bloom-Shamir algorithm, but a constraint of the Crypto-C implementation. See Step 4 for details.

The following example will encrypt 16 bytes (for example, an RC2 key), splitting the secret into four shares, and set the threshold to two.

The example in this section corresponds to the file `scrtshar.c`.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ secretSplitter = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject (&secretSplitter)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

There is only one AI that implements the Bloom-Shamir secret sharing algorithm: `AI_BSSecretSharing`. The *Library Reference Manual* Chapter 2 entry on this AI reports that the format of ***info*** supplied to `B_SetAlgorithmInfo` is the following `struct`:

```
typedef struct {
  unsigned int threshold;                               /* share threshold */
} B_SECRET_SHARING_PARAMS;
```

Because you want to set the threshold to two, set your algorithm object as follows:

```
B_SECRET_SHARING_PARAMS secretSharingParams;

secretSharingParams.threshold = 2;

if ((status = B_SetAlgorithmInfo
     (secretSplitter, AI_BSSecretSharing,
      (POINTER)&secretSharingParams)) != 0)
  break;
```

Step 3: Init

Initialize the algorithm with `B_EncryptInit`. No key is necessary, so pass a properly cast `NULL_PTR` for the key object. This algorithm object does not need an algorithm chooser, so pass a properly cast `NULL_PTR` for that argument as well. This function is very quick, so it is reasonable to pass a NULL_PTR for the surrender context:

```
if ((status = B_EncryptInit
     (secretSplitter, (B_KEY_OBJ)NULL_PTR,
      (B_ALGORITHM_CHOOSER)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Call B_EncryptUpdate once for each of the total number of shares. Each call to B_EncryptUpdate produces a share. For each share, you must allocate a space that is one byte larger than the secret. A share is actually the same size as the secret, but Crypto-C also appends one byte containing the number of the share. (This is why Crypto-C limits the shares to 255; it is the largest integer one byte can represent.) Make sure you do not overwrite a previous share.

The input for each call to B_EncryptUpdate is the secret itself. You also need a random algorithm for the first call to B_EncryptUpdate. You can pass a random algorithm each time, however; Crypto-C simply ignores it on each successive call. Complete Steps 1 through 4 of “Generating Random Numbers” on page 147. You do not need random bytes, only an algorithm that can generate them. This function is not too time-consuming, so it is reasonable to pass a properly cast NULL_PTR for the surrender context.

To create four shares, you could use the following:

```
#define SECRET_SIZE   16
#define TOTAL_SHARES   4

static unsigned char secretKey[SECRET_SIZE] = {
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
  0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10
};
unsigned char *secretShare[TOTAL_SHARES];
unsigned int secretShareLen[TOTAL_SHARES];
int count;

for (count = 0; count < TOTAL_SHARES; ++count)
  secretShare[count] = NULL_PTR;
```

```
for (count = 0; count < TOTAL_SHARES; ++count) {
  secretShare[count] = T_malloc (SECRET_SIZE + 1);
  if ((status = (secretShare[count] == NULL_PTR)) != 0)
    break;

  if ((status = B_EncryptUpdate
       (secretSplitter, secretShare[count],
        &(secretShareLen[count]), SECRET_SIZE + 1,
        secretKey, SECRET_SIZE, randomAlgorithm,
        (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
}
if (status != 0)
  break;
```

Step 5: Final

Finalize the process with `B_EncryptFinal`. This function does not need a random algorithm, so pass a `NULL_PTR`. It is a quick call, so it is reasonable to pass a `NULL_PTR` for the surrender context:

```
unsigned int outputLenFinal;

if ((status = B_EncryptFinal
     (secretSplitter, NULL_PTR, &outputLenFinal, 0,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
```

Step 6: Destroy

Remember to destroy all objects and free up any allocated memory when you are done. Save the shares to files or disks before freeing the memory:

```
B_DestroyAlgorithmObject (&secretSplitter);
B_DestroyAlgorithmObject (&randomAlgorithm);
for (count = 0; count < TOTAL_SHARES; ++count)
  T_free (secretShare[count]);
```

Reconstructing The Secret

To reconstruct the secret, call `B_DecryptUpdate` for each share you are entering. You need at least ***threshold*** number of shares; if you enter fewer, `B_DecryptFinal` will return an error. Any combination of threshold shares will work.

Step 1: Creating An Algorithm Object

Declare a variable to be `B_ALGORITHM_OBJ`. As defined in the function prototype in Chapter 4 of the *Library Reference Manual*, its address is the argument for `B_CreateAlgorithmObject`:

```
B_ALGORITHM_OBJ secretReconstructer = (B_ALGORITHM_OBJ)NULL_PTR;

if ((status = B_CreateAlgorithmObject
     (&secretReconstructer)) != 0)
  break;
```

Step 2: Setting The Algorithm Object

Use the same AI, `AI_BSSecretSharing`:

```
B_SECRET_SHARING_PARAMS secretSharingParams;

secretSharingParams.threshold = 2;

if ((status = B_SetAlgorithmInfo
     (secretReconstructer, AI_BSSecretSharing,
      (POINTER)&secretSharingParams)) != 0)
  break;
```

Step 3: Init

Initialize the algorithm with `B_DecryptInit`. Once again no key or algorithm chooser is necessary. This function is very quick, so it is reasonable to pass a `NULL_PTR` for the surrender context:

```
if ((status = B_DecryptInit
     (secretReconstructer, (B_KEY_OBJ)NULL_PTR,
      (B_ALGORITHM_CHOOSER)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
  break;
```

Step 4: Update

Call B_DecryptUpdate once for each of the shares you are using to reconstruct the secret. You can use any number of shares from the threshold number to the total number of shares.

Each call to B_DecryptUpdate produces no output, so pass NULL_PTRs. The input is a share. This call does not need a random algorithm, so pass a NULL_PTR. It is also quick, so it is reasonable to pass a properly cast NULL_PTR for the surrender context:

```
unsigned int outputLenUpdate;

for (count = 0; count < (int)secretSharingParams.threshold; ++count) {
  if ((status = B_DecryptUpdate
       (secretReconstructer, NULL_PTR, &outputLenUpdate,
        0, secretShare[count], secretShareLen[count],
        (B_ALGORITHM_OBJ)NULL_PTR,
        (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
}
if (status != 0)
  break;
```

Step 5: Final

Finalize the process with B_DecryptFinal. There will be output now. This function does not need a random algorithm, so pass a NULL_PTR there. It is a quick call, so it is reasonable to pass a NULL_PTR for the surrender context:

```
unsigned char getSecret[SECRET_SIZE]
unsigned int getSecretLen;

if ((status = B_DecryptFinal
     (secretReconstructer, getSecret, &getSecretLen, SECRET_SIZE,
      (B_ALGORITHM_OBJ)NULL_PTR,
      (A_SURRENDER_CTX *)NULL_PTR)) != 0)
```

Step 6: Destroy

Remember to destroy all objects and free up any allocated memory when you are done:

```
B_DestroyAlgorithmObject (&secretReconstructer);
```

Chapter 8

Cryptographic Hardware

Crypto-C is designed to interface with cryptographic hardware devices. If you are using such a device and the manufacturer has built an interface to Crypto-C using BSAFE Hardware Application Programming Interface (BHAPI), you can write an application that will use the hardware.

Using Hardware Registration

This section describes a typical scenario for modifying an application to use hardware registration.

An application can define this algorithm method in `chooser.c`:

```
B_ALGORITHM_METHOD CHOOSER[] = {&AM_CBC_DES_ENCRYPT, ...,NULL_PTR};
```

To modify the application to use hardware registration, execute the following steps:

1. Modify `chooser.c` and rename ***CHOOSER*** to ***FIXED_CHOOSER***.
2. Add two declarations in `main`:

```
B_Chooser CHOOSER = (B_Chooser)NULL_PTR;
unsigned char **listOfOEMTags = (unsigned  char**)NULL_PTR;
```

3. Add a call in `main` to `B_CreateSessionChooser` that precedes all calls to the Crypto-C initialization calls:

```
B_CreateSessionChooser
 (
 FIXED_CHOOSER,
 &CHOOSER,
 FIXED_HARDWARE_LIST,                              /* defined in hrdwrsmp.c */
                                                         /* in btest/source */
 NULL_PTR,          /* reserved for use with dynamic hardware libraries */
 NULL_PTR,          /* reserved for use with dynamic hardware libraries */
 &listOfOEMTags                /* Used to identify supplier of given AM */
                              /* for purposes of selecting a given OEM. */
 );
```

4. Add a call in `main` to `B_FreeSessionChooser` after all the Crypto-C calls.

```
 B_FreeSessionChooser
   (
    &CHOOSER,
    &listOfOEMTags
   );
```

Retrieving Random Numbers

This demonstration program shows how you can use the `AM_HW_RANDOM` algorithm method to glean true random numbers from a cryptographically secure random number generator on a hardware co-processor. The use of `AM_HW_RANDOM` is almost totally transparent; it may be used in place of `AM_MD5_RANDOM`, `AM_MD2_RANDOM`, or `AM_SHA_RANDOM`.

Using a hardware device that implements the BHAPI interface is similar to using Crypto-C software function calls. The differences are mentioned below.

Step 0: Include Files

As with any Crypto-C program, begin by including the appropriate files:

```
#include "aglobal.h"
#include "bsafe.h"

#include "bhapi.h"
```

Step 1: Creating an Algorithm Object

Just as in any Crypto-C program, you create an algorithm object by declaring a variable to be an algorithm object and calling `B_CreateAlgorithmObject`:

```
  B_ALGORITHM_OBJ randomAlgorithm = (B_ALGORITHM_OBJ)NULL_PTR;

  if ((status = B_CreateAlgorithmObject (&randomAlgorithm)) != 0)
    break;
```

Step 1a: Create the session chooser

When accessing a hardware device, Crypto-C uses what is called a *session chooser*. This chooser combines the hardware-based algorithm methods indicated in the hardware chooser list and a standard algorithm chooser to create a hardware-based chooser. For this example, assume you already have a hardware chooser list containing a hardware implementation of `AM_HW_Random` and a software chooser. Let `FIXED_HARDWARE_LIST` be the hardware list constructed from the set of available hardware-based methods and `SOFTWARE_CHOOSER` be a standard software algorithm chooser.

Note: Consult the documentation for your hardware to the list of available hardware-based methods. See "Algorithm Choosers" on page 118 for more information on software choosers.

To create the session chooser for use with your hardware, call `B_CreateSessionChooser`. As defined in the *Library Reference Manual*, Chapter 4, this function takes 6 arguments:

```
int B_CreateSessionChooser (
B_Chooser         fixedChooser,    /* Chooser consisting of software-based */
                                                  /* algorithm methods. */
B_Chooser        *sessionChooser,  /* Runtime chooser dynamically bound to */
                                    /* available hardware based methods. */
HW_TABLE_ENTRY   *staticHardwareList[ ],     /* List of statically defined */
                                     /* hardware methods terminated by a */
                                             /*  properly cast NULL_PTR. */
ITEM             *passPhrase,                     /* hardware passphrase */
POINTER          *amTagList,               /* For now pass (*)NULL_PTR */
unsigned char  ***listOfOEMTags            /* Returns list of OEM tags */
                                        /* for methods in sessionChooser */
);
```

For the first argument, pass in your software chooser, ***SOFTWARE_CHOOSER***. The second argument is a pointer to a location in memory where the session chooser will be placed. The third argument is the fixed hardware list, ***FIXED_HARDWARE_LIST***. The fourth argument is the ***passPhrase***, which is used to control access to the hardware. At registration time, the hardware-method interface can check whether the ***passphrase*** contains the access code to enable access to a hardware instantiation of a particular method. In this example, pass `NULL_PTR`, since `AM_HW_Random` does not require a ***passPhrase***. The fifth argument is ***amTagList***, which supports the dynamic linking of DLL version of hardware libraries into the session chooser. According to the *Library Reference Manual* Chapter 4 entry for `B_CreateSessionChooser`, you pass a properly-cast `NULL_PTR` for ***amTagList***. The final argument is a `char *`. A list of OEM tags will be placed here by Crypto-C.

Now you can create the call to `B_CreateSessionChooser`:

```
B_ALGORITHM_CHOOSER SESSION_CHOOSER = (B_ALGORITHM_CHOOSER)NULL_PTR;
HW_TABLE_ENTRY *FIXED_HARDWARE_LIST[] = {
  & HW_XYZ_RANDOM, (HW_TABLE_ENTRY *)NULL_PTR
};
B_ALGORITHM_METHOD *SOFTWARE_CHOOSER[] = {
  &AM_HW_RANDOM,
  (B_ALGORITHM_METHOD *)NULL_PTR
};

if ((status = B_CreateSessionChooser
     (SOFTWARE_CHOOSER, &SESSION_CHOOSER,
     (POINTER *)FIXED_HARDWARE_LIST, (ITEM*)NULL_PTR,
     (POINTER*)NULL_PTR, &oemTagList)) != 0)
  break;
```

Step 2: Setting the Algorithm Object

Setting the algorithm object for a hardware implementation is the same as for a software implementation. Just pass in the correct AI; in this case, it is `AI_HW_Random`:

```
if ((status = B_SetAlgorithmInfo
    (randomAlgorithm, AI_HW_Random, NULL_PTR)) !=0 )
  break;
```

Step 3: Init

Now you need to call `B_RandomInit`. This call is the same as any other call to `B_RandomInit`, except that you must pass in the chooser, ***SESSION_CHOOSER***, created by the call to `B_CreateSessionChooser`:

```
  if ((status = B_RandomInit
            (randomAlgorithm, SESSION_CHOOSER,
            (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
```

Step 4: Update

No Update step is needed for `AI_HW_Random`. In a software implementation, you would call `B_RandomUpdate` during the Update step to seed the pseudo-random number generator. In this case, since you're dealing with a hardware source of randomness, you don't need to worry about setting the seed, because the hardware should take care of seeding the generator, if necessary.

Step 5: Generate

After you have initialized the random number generator, generate your random bytes:

```
    randomData = T_malloc (randomDataLen);
    if ((status = (randomData == NULL_PTR)) != 0)
      break;

     if ((B_GenerateRandomBytes
                (randomAlgorithm, randomData, randomDataLen,
                (A_SURRENDER_CTX *)NULL_PTR)) != 0)
      break;
```

Step 6: Destroy

Be sure to destroy the session chooser, the random algorithm object, and any memory that you allocated:

```
  B_FreeSessionChooser (&SESSION_CHOOSER, &oemTagList);
  B_DestroyAlgorithmObject (&randomAlgorithm);
  T_free(randomData);
```

Appendix A

Command-Line Demos

Overview of the Demos

In addition to the sample programs included on the CD, there are three Crypto-C command-line demo applications: BDEMO, BDEMODSA, and BDEMOEC. These are actual applications that demonstrate some of the aspects of building cryptographic applications using Crypto-C. They use the Crypto-C library routines and are provided to all Crypto-C customers in source form.

The BDEMO application is found in `bdemo.c` with supporting files `fileio.c`, `filebsl.c`, `tstdlib.c`, a chooser, `choosc.c`, and include files `fileio.h`, `filebsl.h` and `demochos.h`. Because BDEMO utilizes BSLite, `bslite.c` must be linked in and the `bslite.h` file must be included. See “BSLite” on page 292 for more information about BSLite.

The command-line demos provide the following functionality:

- BDEMO can create and verify an RSA digital signature for a DES-encrypted file. It can also seal and open an RSA digital envelope, placing the encrypted output in another file. The signature and envelope methods used by Crypto-C are compatible with the Public-Key Cryptography Standards (PKCS).
- BDEMODSA demonstrates the use of DSA to digitally sign and verify the integrity of data files.

- BDEMOEC can use ECDSA to create and verify digital signatures for a file, and it can use the Elliptic Curve Authenticated Encryption Scheme (ECAES) to seal and open a digital envelope, placing the output in another file. These demo programs support input files of arbitrary length. As with BDEMO, the file to be sealed with the digital envelope is encrypted using the DES algorithm; however, in BDEMOEC, the DES key is encrypted using ECAES instead of RSA.

This appendix has three sections. “Command-Line Demo User’s Guide” on page 283 shows how to use the BDEMO, BDEMODSA, and BDEMOEC Command-Line Demos. “File Reference” on page 290 explains the files used in these applications. “BSLite” on page 292 describes the BSLite routines.

Command-Line Demo User's Guide

The three command-line demos are menu-driven application that demonstrates basic cryptographic operations. Each demo prompts you for commands; you type the responses. The various commands and expected responses are explained in the sections for the individual demos.

BDEMO

Starting BDEMO

Command Line mode

To start BDEMO, enter the following after the system prompt:

```
> bdemo
```

Input Redirection mode

You may also run BDEMO in input redirection mode where your responses to the menu prompts are read from a file. For example, to read commands from a file named `testin`, enter the following after the system prompt:

```
> bdemo -s < testin
```

Notice that this uses '<' to redirect `testin` as the input to BDEMO. The **`-s`** option to BDEMO eliminates the menu prompts when BDEMO is taking input from a file.

Any line that is blank or begins with '#' is ignored. This means that the file used in response file mode may contain blank lines and comment lines that begin with '#'.

Specifying User Keys

BDEMO comes pre-loaded with RSA key pairs for two test users: User 1 and User 2. You can also use BDEMO to generate a new RSA key pair; if you do so, this becomes the key pair for User 3. See "Generate a Key Pair" on page 285 for key pair generation.

Note: Key pair generation in BDEMO is for demonstration purposes only and is *not* cryptographically secure.

When you sign, verify, seal, or open a file, BDEMO asks which user's key to use. You can specify either 1 or 2. If you have generated a new RSA key, you can specify 3.

Using BDEMO

When you type "**bdemo**" at the system prompt, the following top-level menu is displayed:

```
S - Sign a file
E - Envelope a file
V - Verify a signed file
O - Open an enveloped file
G - Generate a keypair (may take a long time)
Q - Quit
  Enter choice:
```

Commands may be entered in either upper or lower case, and all but the initial letter of a command is ignored. So, for example, to sign a file you may either type "**s**" or "**sign**".

Each of the commands on this top-level menu is described below.

Sign a File

To sign a file:

1. Enter "**s**" at the top-level menu.
2. You will be prompted in succession for:
   - the name and location of the file to be signed
   - the name of the file you want to create to hold the signature
   - the private key used for signing
3. Once this information is supplied, BDEMO uses the private key to create a signature.

Envelope a File

To create an envelope for a file:

1. Enter "**e**" at the top-level menu.
2. You will be prompted in succession for:
   - the name and location of the file to be signed and enveloped
   - the names of the files for storing the encrypted DES key, the initialization vector (IV), and the encrypted data
   - a seed for generating the random DES key and the IV

3. Once this information is supplied, BDEMO encrypts the DES key using the recipient's public key, saving the IV, encrypted DES key, and the encrypted content in the previously specified files.

Verify a Signed File

To verify the signature for a file:

1. Enter "`v`" at the top-level menu.
2. You will be prompted in succession for:
   - the name and location of the file to be verified
   - the digital signature file
   - the signer's user number (1 or 2; you may also choose 3 if you have generated a key pair)
3. BDEMO uses the signer's public key to verify the signature. If the signature is valid, BDEMO prints "**`Signature verified.`**"; otherwise, BDEMO prints "**`ERROR: Invalid signature while verifying file.`**"

Open an Enveloped File

To open an enveloped file:

1. Enter "`o`" at the top-level menu.
2. You will be prompted in succession for:
   - the name and location of the file that contains the encrypted data
   - the name and location of the of the file that contains the encrypted DES key
   - the name and location of the of the file that contains the IV
   - the name of the file where the decrypted content should be stored. To print the content to the screen instead, use a hyphen (`-`) as the file name
   - the recipient's user number
3. BDEMO uses the recipient's private key to recover the DES key. It then uses the DES key to decrypt the data and saves it to the specified file. If a hyphen was entered as the output file name, it prints the decrypted data to the screen instead of saving it to a file.

Generate a Key Pair

You can use BDEMO to generate a new RSA key pair. However, *this is only for demonstration purposes, and does not generate cryptographically secure RSA keys*. BDEMO will generate an RSA public/private key pair, but the keys are lost when you exit

BDEMO.

To generate a key pair:

1. Enter "**`g`**" at the top-level menu.
2. You will be prompted in succession for:
   - the key size in bits
   - some seed information
3. BDEMO generates the key pair and keeps it as the key pair for User 3. Once a keypair has been generated, you may not generate another during the same BDEMO session.

Depending on the key size and the speed of the computer, key pair generation may take from a few seconds to several minutes.

BDEMODSA

Running BDEMODSA

Command Line mode

To start BDEMODSA, enter the following after the system prompt:

```
> bdemodsa
```

Input Redirection mode

You may also run BDEMODSA in input redirection mode where your responses to the menu prompts are read from a file. For example, to read commands from a file named `testsgn`, enter the following after the system prompt:

```
> bdemodsa -s < testsgn
```

Notice that this uses '<' to redirect `testsgn` as the input to BDEMODSA. BDEMODSA's **`-s`** option is used to omit the menu prompts when input is taken from a file.

Any line that is blank or begins with '#' is ignored. This means that the file used in response file mode may contain blank lines and comment lines that begin with '#'.

Using BDEMODSA

When you use BDEMODSA in command-line mode, you will be prompted to generate a DSA key pair for your BDEMODSA session. To do this:

1. Start BDEMODSA by typing "**`bdemodsa`**" at the system prompt

   The request "**`Enter seed to generate DSA keypair (blank to cancel):`**" is displayed.
2. Enter any arbitrary string of printable characters.

   The message "**`Generating DSA Keypair,  please wait...`**" is displayed. Depending on the computer and level of code optimization, key generation will take from several seconds to several minutes.

   When the key pair has been generated, the message "**`DSA public key and private key are now ready to use`**" is displayed.

Once a key pair has been generated, the following top-level menu is displayed:

```
S - Sign a file using DSA/SHA
V - Verify a DSA signed file
Q - Quit
  Enter choice:
```

Commands may be entered in either upper or lower case, and all but the initial letter of a command is ignored. So, for example, to sign a file you may either type "**`s`**" or "**`sign`**".

The commands on this top-level menu are described below.

Sign a File

To sign a file:

1. Enter "**`s`**"
2. You will be prompted in succession for:
   - the name and location of the file to be signed
   - the name of the file that will hold the signature
3. BDEMODSA uses the private key generated at the beginning of the session to create a signature and places the result in the specified file.

Verify a Signed File

To verify the signature for a file:

1. Enter "**`v`**"

2. You will be prompted in succession for:
   - the name and location of the file that was signed
   - the name and location of the file containing the digital signature
3. BDEMODSA uses the public key generated at the beginning of the session to verify the signature. If the signature is valid, BDEMODSA prints "**`Signature verified.`**"; otherwise, BDEMODSA prints "**`ERROR: Invalid signature while verifying file`**".

Note: If the signature was generated during a previous execution of BDEMODSA, it is necessary to re-use the seed from signature signing, otherwise verification will fail.

BDEMOEC

BDEMOEC provides the same functionality as BDEMO, but uses elliptic curve for its algorithms. The algorithm used for sealing and opening digital envelopes is ECAES to encrypt the DES symmetric key. Digital signatures are created and verified using ECDSA with SHA1.

A set of elliptic curve parameters are hard-coded in the demo along with two key pairs generated with that curve. A new key pair can be generated, but since the size of the key pair is dependent on the elliptic curve parameters used, the user cannot specify the desired key size.

Running BDEMOEC

Command Line mode

To start BDEMOEC, enter the following after the system prompt:

```
> bdemoec
```

Input Redirection mode

You may also run BDEMOEC in input redirection mode where your responses to the menu prompts are read from a file. For example, to read commands from a file named `testin`, enter the following after the system prompt:

```
> bdemoec -s < testec
```

Notice that this uses '<' to redirect `testin` as the input to BDEMOEC. The **`-s`** option to BDEMOEC eliminates the menu prompts when BDEMOEC is taking input from a file.

Any line that is blank or begins with '#' is ignored. This means that the file used in response file mode may contain blank lines and comment lines that begin with '#'.

Using BDEMOEC

The menu options and procedures for BDEMOEC are identical for those for BDEMO. See "Using BDEMO" on page 284 for a description of the menu commands.

File Reference

The C source code files for the demo programs provide a convenient means to learn Crypto-C by example and are a good starting point for your own Crypto-C applications.

The source files for the demo programs are described in Table A-1:

Table A-1 **Demo Program Source Files**

File(s)	Description
`bdemo.c`	This file contains BDEMO's `main` function, menu interpreter, and drivers for each of the menu commands. This file uses the standard C library functions such as `printf`, `fopen`, etc.
`bdemodss.c`	This file contains BDEMODSA's `main` function. It is entirely analogous to `bdemo.c`.
`bdemoec.c`	This file contains BDEMOEC's main function. It is entirely analogous to `bdemo.c`. The elliptic curve parameters used for this demonstration, along with two key pairs, are hard-coded in the beginning of this file.
`bslite.c` and `bslite.h`	`bslite.c` contains a collection of routines that enable BDEMO to interface to the Crypto-C cryptographic library. The routines are written in straightforward, easy-to-read portable C code. These routines also illustrate the coding of interfaces to a number of common Crypto-C library functions. A developer may wish use this module as a starting point for developing an application. Refer to "`blreadme`" (in the `demosrc` directory) for extended descriptions of routines contained in `bslite.c`.
`bsliteds.c` and `bsliteds.h`	`bsliteds.c` contains routines used by BDEMODSA to interface to the Crypto-C library. These routines illustrate how to code portable interfaces to Crypto-C's implementation of the Digital Signature Algorithm.
`bslec.c` and `bslec.h`	`bslec.c` contains routines used by BDEMOEC to interface to the Crypto-C library. These routines are analogous to `bslite.c` and `bslite.h`. However, not all functions in `bslite.c` have a counterpart in `bslec.c`.
`choosc.c` and `demochos.h`	These files define the `DEMO_ALGORITHM_CHOOSER` which may be used as a default for the *`algorithmChooser`* argument to Crypto-C routines. `DEMO_ALGORITHM_CHOOSER` is externally declared in `demochos.h` for inclusion by applications that need access to the `DEMO_ALGORITHM_CHOOSER`.
`filebsl.c`, `filebsl.h`, `fileio.c` and `fileio.h`	These files call on the BSLite routines in `bslite.c` and handle the file I/O for each operation. These files use the standard C library functions such as `printf`, `fopen`, etc.

Table A-1 **Demo Program Source Files**

File(s)	Description
`fbslec.c`, `fbslec.h`, `fileio.c` and `fileio.h`	These files are used by BDEMOEC. These files call on the routines in `bslec.c` and handle the file I/O for each operation. These files use the standard C library functions such as `printf`, `fopen`, etc. The files `fbslec.c` and `fbslec.h` are analogous to filebsl.c and filebsl.h used by BDEMO.
`tstdlib.c`	This file contains memory, I/O, and buffer manipulation routines needed by Crypto-C, such as `T_malloc` and `T_memcmp`. This file illustrates how these routines can be implemented on most platforms. However, some of these routines may need alteration for different platforms. For example, Crypto-C requires that `T_free` perform no function if it is passed `NULL_PTR`, but some library implementations of `free` may not satisfy this convention. Therefore, an explicit check for `NULL_PTR` may be needed in `T_free`. `tstdlib.c` uses the constant `MEMMOVE_PRESENT`. If the platform's C library provides `memmove`, `MEMMOVE_PRESENT` should be defined as 1; otherwise, it should be defined as 0. In `tstdlib.c`, default values are given for these constants, but they may be overridden by a compiler flag. For example: `-DMEMMOVE_PRESENT=0`

BSLite

BSLite is a collection of routines that interface with the Crypto-C library. BSLite demonstrates how to call Crypto-C to execute various cryptographic procedures. The routines are written in straightforward, easy-to-read portable C and is provided to all Crypto-C customers in source form. BSLite includes a number of the most popular functions the Crypto-C library supports:

- symmetric key generation
- symmetric block and stream encryption
- Diffie-Hellman parameter generation
- Diffie-Hellman key agreement
- message digest computation
- RSA key generation
- RSA digital signature creation and verification
- RSA digital envelope sealing and opening
- password-based private key protection/encryption

A single C source file, `bslite.c`, with a single header file, `bslite.h`, contains the entire BSLite Code. For more information on BSLite, see the file `blreadme`.

Appendix B

References and Reading Material

1. *The Public-Key Cryptography Standards (PKCS)*, RSA Laboratories. (`http://www.rsa.com/rsalabs/pubs/PKCS/`)
2. *Frequently Asked Questions (FAQ) About Today's Cryptography*, available from RSA Data Security, Inc. See RSA's web site at `http://www.rsa.com`.
3. The following Internet Standard documents:
    - RFCs 1421, 1422, 1423, 1424 on Privacy Enhancement for Internet Electronic Mail
    - RFCs 1319 (MD2), 1321 (MD5).
4. The following CCITT Recommendation documents:
    - X.690: Specifications for the Basic Encoding Rules (BER) for Abstract Notation One (ASN.1).
    - X.509: The Directory — Authentication Framework.
5. Rivest, Shamir, and Adleman, A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, February 1978.
6. A. Shamir, How to share a secret. *Communications of the ACM*, 22(11): 612-613, November 1979.
7. W. Diffie and M. E. Hellman, New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644-654, 1976.

8. *Data Encryption Standard*, FIPS Pub 46-2, National Institute of Standards and Technology. Available from `http://www.nist.gov.itl/div897/pubs/index.htm`.

9. *DES Modes of Operations*, FIPS Pub 81, National Institute of Standards and Technology, 1980.

10. Digital Signature Standard and Secure Hashing Algorithm (DSS and SHA)
    - FIPS Pub 180-1
    - X9.30 Part III

11. The following reports from RSA Laboratories (`http://www.rsa.com/rsalabs`):
    - Stream Ciphers
    - MD2, MD4, MD5, SHA and Other Hash Functions
    - On Pseudo-collisions in MD5
    - Results from the RSA Factoring Challenge
    - Recommendations on Elliptic Curve Cryptosystems
    - Recent Results for MD2, MD4, and MD5

12. The following OAEP specifications:
    - *SET Secure Electronic Transaction Specification. Book 3: Formal Protocol Definition, version 1.0.* SETCo, 1997. (`http://www.setco.org/`)
    - *PKCS #1: RSA Cryptography Specifications. Version 2.0*. RSA Data Security, Inc., 1998. (`http://www.rsa.com/rsalabs/pubs/PKCS/`)

13. The following ANSI Financial Services Industry documents:
    - X9.31 (RSA signatures, reversible DSA)
    - X9.52 Draft (Triple DES)
    - X9.62 Draft and X9.63 Draft (Elliptic Curves)

14. IEEE *Standard Specifications for Public-Key Cryptography* on `http://stdsbbs.ieee.org/groups/1363/index.html.`

15. B. Schneier, *Applied Cryptography,* John Wiley & Sons, Inc., New York, 1994.

16. G. Simmons, *Contemporary Cryptography,* IEEE Press.

17. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996. Chapter 2 of this book, which covers all aspects of modern cryptography, provides mathematical background on finite fields.

18. A. Menezes, I. Blake, X. Gao, R. Mullin, S. Vanstone, and T. Yaghoobian. *Applications of Finite Fields.* Kluwer Academic Publishers, 1993. Provides further reference material on finite fields, including techniques for representing elements.

19. A. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.

20. Joseph H. Silverman and John Tate, *Rational Points on Elliptic Curves*, Springer-Verlag New York, Inc., 1992.

Index