Digital Signature Manager

This chapter describes the Digital Signature Manager, one of the AOCE security services. The Digital Signature Manager is a set of routines that allows you to add electronic signature capabilities to your application.

Read this chapter if you plan to let your users sign documents or other data electronically, so that recipients can be confident of the authenticity of the signature and the integrity of the signed data.

Note that other AOCE components provide limited use of digital signatures; the AOCE Standard Mail Package allows users to add digital signatures to electronic mail and to check the signatures of received mail. The Interprogram Messaging (IPM) Manager provides an application program interface to add digital signatures to IPM messages and to verify such signatures. A user who has the AOCE software installed can use the Finder to add a digital signature to any file. If, however, you want to allow a user to sign a file or verify a signature from within your application, you must use the routines described in this chapter.

This chapter first introduces the concept of digital signatures, including a brief introduction to public-key cryptography. It goes on to explain public-key certificates—necessary documents for creating digital signatures. It then explains how to use the Digital Signature Manager to create a signature for a file or portion of a file, and to verify a signature. It also explains how to get information from a digital signature.

About Digital Signatures

A ***digital signature*** is an encrypted number that is associated with a particular set of data. It has two purposes. First, it uniquely identifies the individual or entity that signed or authorized the content of the data. Second, it ensures the integrity of the data; the signature contains coded information that can be used to detect any changes made to the data after the creation of the signature.

A digital signature can be applied to an entire set of data or to any portion of it; anything that can be represented as a stream of bytes can be given a digital signature. You can use the functions in this chapter to sign a file, one or more fields in a form, data in memory, or even another digital signature. In terms of security and integrity, an item with a verifiable digital signature is comparable to a paper document that is signed and notarized. In most ways, digital signatures can provide better security than signed paper forms, because a digital signature cannot be forged and because a digitally signed document cannot be altered without the alteration being detected.

The digital signature capability is useful in networked organizations. Users on the network can fill out forms and route them electronically for signature, thus saving time and effort and enhancing security. Even users of computers that are not on a network can sign electronic forms before mailing them or physically delivering them. Digital signatures can also be used with data that is not transmitted at all; a user can assign a digital signature to important data left on a computer or server to ensure that the data is not tampered with. This capability could be used to detect viruses, for example.

6

Digital Signature Manager

Cryptography and Digital Signatures

The digital signature technology used by the AOCE toolbox involves the use of two ***keys***, large unique numbers that are computationally applied to data to encrypt or decrypt it.

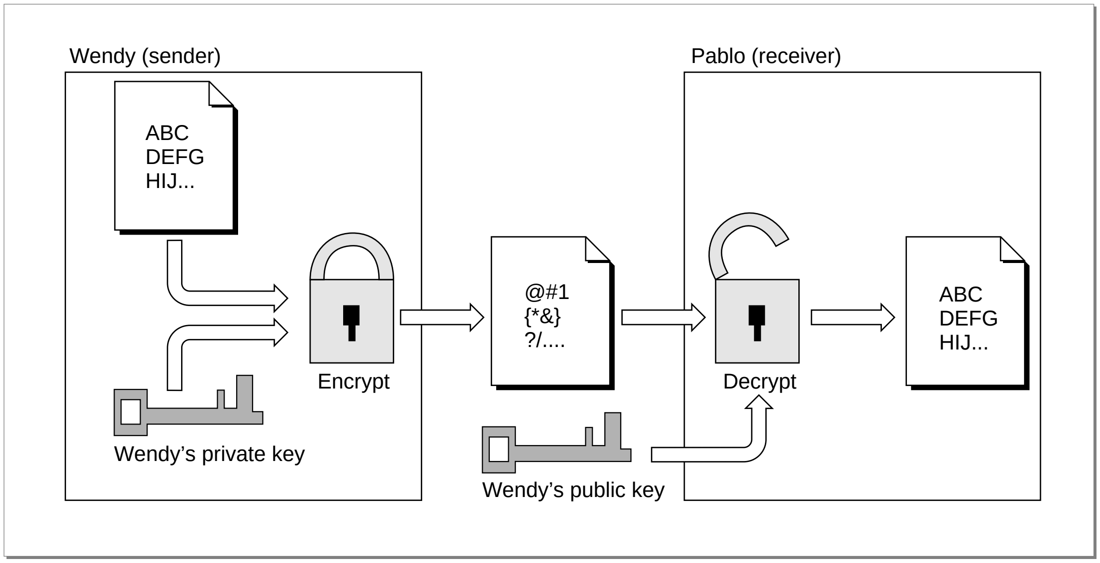
The Digital Signature Manager does not encrypt documents; encryption and decryption are applied to the digital signature only. When you send an electronically signed document, the contents of the document are as public as the channel over which you transmit.

Common cryptographic techniques typically involve a single key, one that both decrypts and encrypts information. Any holder of the key used in the encryption can use it to decrypt the information. Those wishing to exchange information must keep the key a secret among themselves. This type of cryptography is called ***secret-key cryptography***.

The AOCE services use another cryptographic technique, called ***public-key cryptography***. In this technique, key holders use a pair of keys to encrypt and decrypt information. Each key pair consists of a ***private key*** and a ***public key*** . A key holder must keep its private key secret and not share that key with anyone else. At the same time, it may freely publish and exchange its public key without compromising security.

Both the private and public keys can be used to encrypt information and decrypt information. Information encrypted with a private key can be decrypted only with its paired public key. Similarly, information encrypted with a public key can be decrypted only with its paired private key. Figure 6-1 illustrates the concept of public-key encryption.

Figure 6-1 Principles of public-key encryption



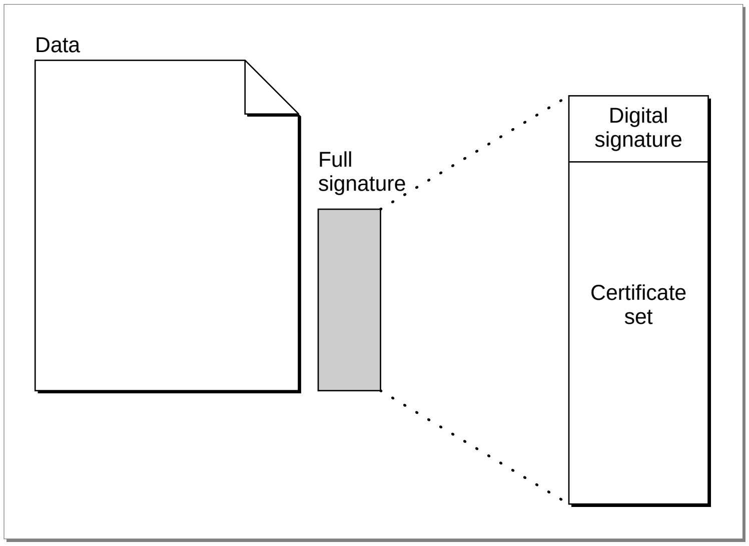
The sender, Wendy, uses her own private key to encrypt an item. The receiver, Pablo, can decrypt the item and read it because he has access to Wendy's public key. Wendy's public key is widely available, so the contents of the item are not hidden to anyone with the key. But because only Wendy's public key can decrypt the item, anyone who successfully decrypts the item knows that it must have come from Wendy.

Components of a Full Signature

As implemented by the Digital Signature Manager, an electronically signed item consists of (1) the item itself, any collection of data; and (2) a full digital signature, a stream of bytes that can be used to verify the integrity of the item's data and that uniquely identifies the signer.

A full signature has two components: the digital signature itself and the certificate set of the signer. Figure 6-2 illustrates the components of a full signature. Certificate sets provide the signer's public key, verification of the authenticity of that key, and the identity of the signer. They are described in "The Certificate Set" on page 6-6.

Figure 6-2 The components of a full signature



The Digital Signature

A digital signature is an encrypted digest. A ***digest*** is a 16-byte number, calculated by the Digital Signature Manager from a set of data, that reflects the content of that data. The digest is like a sophisticated checksum but far more reliable in verifying the integrity of data. It is very nearly impossible for any two sets of data that differ in any way to yield the same digest. The digest, therefore, ensures the integrity of the data; if someone changes even a single bit of a signed item, a recalculation of that item's digest will yield a different number from the digest contained in the signature.

Once the digest is created, it is encrypted. The ***encrypted digest (*** or ***signed digest)*** is the digital signature. The Digital Signature Manager encrypts the digest by applying the signer's private key to it. The encrypted digest is called a signed digest because it could have been created only by the signer (the holder of that private key).

Verifying a digital signature requires decrypting the encrypted digest and comparing it to a new digest of the same data. To decrypt the digest, the recipient of the signed data applies the signer's public key to it. Because an item encrypted with an individual's private key can be decrypted only with that same individual's public key, the very act of correctly decrypting a signature proves the identity of the signer.

Verifying a signature also requires making sure that the data has not changed since it was signed. The Digital Signature Manager creates a digest of the data in its present state and compares it with the decrypted digest from the signature. If they match, the signed data is unchanged.

Finally, verifying a signature requires establishing the authenticity of the public key used for the decryption. To allow for that, the Digital Signature Manager affixes a certificate set (described next) to every digital signature it creates.

The Certificate Set

The second part of a full signature—the certificate set—has three purposes: it provides the signer's public key to allow decryption of the signature, it allows verification of the authenticity of that public key, and it provides the identity of the signer.

Suppose, for example, that Mary (an impostor) claims to be Joe. She signs a document with her own private key and sends it off as a document from Joe. If Mary also sends along her own public key as Joe's public key, then the recipient of the document might use Mary's public key, thinking it was Joe's, to decrypt the signature. The decryption would be successful—because Mary's private key had performed the encryption—and the recipient would mistakenly think the message had been signed by Joe.

As a safeguard against deceptions of this kind, each public key in use is registered with a mutually trusted official issuing organization (such as a corporation or government bureau). That agency publishes a ***public-key certificate***, which includes not only the public key itself but the name of the owner of the key and the name of the organization that issued the certificate (as well as other information; see "About Public-Key Certificates" beginning on page 6-8). See the AOCE user documentation for information on how a user obtains a public-key certificate.

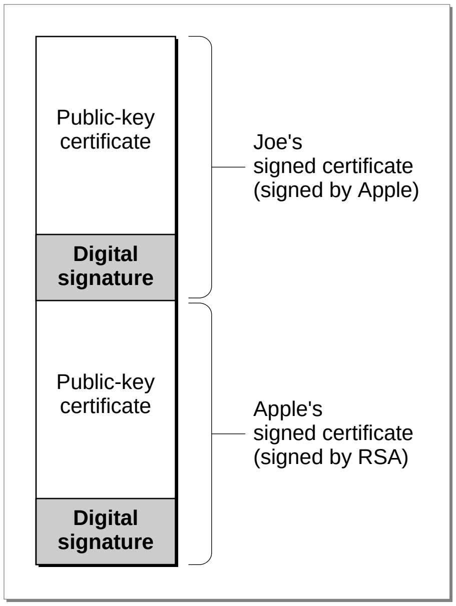
As a guarantee of authenticity, each public-key certificate is itself digitally signed by the issuer of the certificate; it then becomes a ***signed certificate.*** No change to the name or the public key in a signed certificate can go undetected.

The signature on a certificate must itself be verified before the certificate can be considered authentic. For that reason each issuer also has a public-key certificate, signed by *its* issuer. Verifying the signature on a certificate thus leads to another certificate whose signature must be verified, and so on.

For each digital signature this chain of certificates, or ***certificate set***, leads from the signer through all intermediate issuers and up to the prime issuing organization. Verifying a digital signature requires verifying the signatures on all the certificates in the certificate set associated with that signer. This certification process ensures that every public key in every certificate is authentic, as long as one public key—that of the prime issuer—is trusted.

For example, assume Joe has a certificate issued by Apple Computer, Inc. Joe's certificate includes his name and public key, and it is digitally signed by Apple. Apple's certificate includes Apple's name and public key, and like Joe's it is digitally signed. If, in this example, RSA Data Security, Inc., had issued Apple's certificate, then the certificate set in Joe's signed certificate would consist of two certificates, as shown in Figure 6-3.

Figure 6-3 A certificate set consisting of two signed certificates


Public-key certificate
Digital signature
Joe's signed certificate (signed by Apple)
Public-key certificate
Digital signature
Apple's signed certificate (signed by RSA)

RSA has no certificate because there is no authorizing agency to issue it. RSA in this case is the prime issuer, so its public key cannot be verified. It must be trusted for reasons other than verifiability, such as wide public availability. The Digital Signature Manager has access to RSA's public key, so the key is available on every user's Macintosh computer.

In summary, when a recipient verifies the signature, the Digital Signature Manager (1) decrypts the digital signature with the public key provided in the certificate and compares the resulting digest with one it recalculates from the data; and (2) verifies all the digital signatures on the certificate set. If all the verifications are successful, the signer's public key is considered authentic.

Note
Because the number of attributes in a certificate is not limited, a full digital signature can be fairly large—as much as several kilobytes. ◆

Creating and Verifying Signatures

With the Digital Signature Manager you can let users sign documents, and you can verify the signatures on documents received by users. The Digital Signature Manager also provides routines with which you can get information from the certificate set.

When the user wants to sign a document, you call routines that prompt the user for private-key and certificate information, create a digest of the document, and append the user's certificate set to make the full signature. You then attach or otherwise associate the full signature with the document—in a way appropriate to your application—and it is ready to be sent to its recipient by any normal means.

When the user wants to validate a signed document, you locate the full signature by methods appropriate to your application, and you then call routines that verify the signature by decrypting the encrypted digest, creating a new digest from the document and comparing it with the decrypted one, and verifying the authenticity of the public keys in the certificate set. To process a digital signature created in another application, you must know how the other application created the signature.

You may also wish to record or provide the user with additional information, such as who signed the document and when they signed it. To get that information you can make calls that return information about a specified certificate within the full signature.

Note

Users can sign any file by dragging the file onto their signer file. They can verify the signature in a file by clicking the button in the Get Info window for the file. See "Dealing With Standard Signatures in Files" on page 6-22 for information on how to deal with this possibility. ◆

About Public-Key Certificates

Public-key certificates are an integral part of the digital signature concept. Only with an authentic public key can a signature be verified with confidence; the set of signed public-key certificates that accompany every signature is used to ensure that authenticity.

A public-key certificate is an electronic document that verifies the identity of a signer. A public-key certificate contains the following information:

- identifying information for the certificate owner—the entity, person, or organization that is authorized to use the certificate
- the public key of the owner of the certificate
- a time period (range of dates) during which the certificate is valid
- identifying information for the organization that issued the certificate
- a serial number (assigned by the issuer)

A public-key certificate is not valid unless it is digitally signed by the issuing organization of that certificate. It then becomes a signed certificate. The signature assures the authenticity of the certificate owner's name and public key.

A certificate can be owned by a person not affiliated with any organization, a person who is a member of an organization, an organizational role (such as vice president or administrator), or an issuer (a certified authority). A ***distinguished name*** is a set of attributes that fully specify the owner or issuer of a certificate. For example, the distinguished name of a private certificate owner consists of a common name (typically the proper name by which the person is known), a country, a state or province, a locality (such as a city), a zip code, and sometimes a street address.

The 1988 CCITT Recommendation X.520 sets guidelines for the definition and attributes of a distinguished name. The Digital Signature Manager supports a subset of the recommendation. Table 6-1 summarizes the attribute conventions supported by the Digital Signature Manager.

As the table shows, for example, every certification authority (issuing organization) must have either a country name or an organization name—and may have both—but cannot have a common name. Conversely, an individual residential certificate owner must have a common name but cannot have an organizational name or title.

IMPORTANT

When you display a distinguished name, be sure to show the entire set of attributes for that distinguished name. If you show only a portion of the distinguished name, the user might incorrectly identify the owner of the certificate. There may be two identical names, for example, two certificate owners named John Smith. ▲

Table 6-1 Conventions governing attributes of a distinguished name

Mandatory attributes	Optional attributes	Prohibited attributes
Attributes of a certification authority		
Country *or* organization	Country	Title
	Organization	Common name
	State or province	
	Locality	
	Organizational unit	
	Street address	
	Zip code	

continued

Table 6-1 Conventions governing attributes of a distinguished name (continued)

Mandatory attributes	Optional attributes	Prohibited attributes
Attributes of a residential person		
Country	Street address	Organization
State or province		Organizational unit
Locality		Title
Common name		
Zip code		
Attributes of an organizational person		
Organization	Country	
Common name	State or province	
	Locality	
	Street address	
	Organizational unit	
	Title	
	Zip code	
Attributes of an organizational role		
Organization	Country	Common name
Title	State or province	
	Locality	
	Organizational unit	
	Street address	
	Zip code	

A distinguished name can have one or more attributes of each mandatory or optional type, and the attributes can be arranged in a hierarchy to help indicate their relationships. You can use this hierarchy when you display the distinguished name for a user. Figure 6-4 illustrates a hierarchically arranged distinguished name of an organizational person.

Figure 6-4 Hierarchically arranged distinguished name

Country: USA
Organization: Apple Computer, Inc.
  Organizational unit: Research and Development
    Organizational unit: Collaborative Software
    Common name: Pablo Calamera
    Common name: ID number 123456
    Title: Software Engineer

Using the Digital Signature Manager

In using the Digital Signature Manager, the main tasks your application needs to perform are allowing the user to sign a document and verifying the signature on a document that the user receives.

Other Digital Signature Manager features give you added convenience in extracting certificate information—such as the name of the signer—from full signatures.

Note
Because the Digital Signature Manager is loaded into memory when it is used, in general it is a good idea to keep your calls to the Digital Signature Manager close together so that the memory is used only when it is needed. For example, if you call the `SIGNewContext` function when your application starts up, call several Digital Signature Manager routines sometime later, and call the `SIGDisposeContext` function when your application shuts down, the manager code segment remains in memory the whole time. ◆

Determining the Version Number of the Digital Signature Manager

To determine what version of the Digital Signature Manager is available, call the `Gestalt` function, using the selector value `gestaltDigitalSignatureVersion`. Upon completion of the call, the `response` parameter contains the version number in its low-order word. For example, a value of 0x0101 indicates version 1.0.1.

Using a Context

The Digital Signature Manager uses a private data structure called a ***context*** to hold information and the results of calculations while it is processing data. Before you call Digital Signature Manager routines to perform a specific task, you must call the `SIGNewContext` function (see page 6-28) to create a context. This function returns a pointer of type `SIGContextPtr`. You must provide this pointer to each subsequent routine that you call to perform that task. When you first create a context, it can be used for any task; however, once you pass a context to another routine (`SIGSignPrepare`, `SIGVerifyPrepare`, or `SIGDigestPrepare`), it can be used only for that specific task.

For example, to create a signature you first call the `SIGNewContext` function to create a context, then pass that context to the `SIGSignPrepare`, `SIGProcessData`, and `SIGSign` functions (see the following section for details). When you are finished creating the signature, you call the `SIGDisposeContext` function to dispose of the context. Once you have passed the context pointer to the `SIGSignPrepare` function, you cannot use that context to verify a signature or create a digest; you must create a new context for each such operation.

Table 6-2 summarizes the Digital Signature Manager tasks and the functions required to perform each task. The “Optional functions” column lists functions that you can call with the same context you used for the preceding function in the “Required functions” column.

Table 6-2 Digital Signature Manager tasks and functions

Task	Required functions	Optional functions
Creating a signature	`SIGNewContext`	
	`SIGSignPrepare`	`SIGGetSignerInfo`
		`SIGGetCertInfo`
		`SIGGetCertNameAttributes`
		`SIGGetCertIssuerNameAttributes`
	`SIGProcessData`	
	`SIGSign`	`SIGGetSignerInfo`
		`SIGGetCertInfo`
		`SIGGetCertNameAttributes`
		`SIGGetCertIssuerNameAttributes`
	`SIGDisposeContext`	

Table 6-2 Digital Signature Manager tasks and functions (continued)

Task	Required functions	Optional functions
Signing a file	`SIGNewContext`	
	`SIGSignPrepare`	`SIGGetSignerInfo`
		`SIGGetCertInfo`
		`SIGGetCertNameAttributes`
		`SIGGetCertIssuerNameAttributes`
	`SIGSignFile`	`SIGGetSignerInfo`
		`SIGGetCertInfo`
		`SIGGetCertNameAttributes`
		`SIGGetCertIssuerNameAttributes`
	`SIGDisposeContext`	
Checking for a standard signature	`SIGFileIsSigned`	
Verifying a file	`SIGNewContext`	
	`SIGVerifyPrepare`	
	`SIGProcessData`	
	`SIGVerify`	`SIGShowSigner`
		`SIGGetSignerInfo`
		`SIGGetCertInfo`
		`SIGGetCertNameAttributes`
		`SIGGetCertIssuerNameAttributes`
	`SIGDisposeContext`	
Verifying a signature	`SIGNewContext`	
	`SIGVerifyFile`	`SIGShowSigner`
		`SIGGetSignerInfo`
		`SIGGetCertInfo`
		`SIGGetCertNameAttributes`
		`SIGGetCertIssuerNameAttributes`
	`SIGDisposeContext`	

continued

Table 6-2 Digital Signature Manager tasks and functions (continued)

Task	Required functions	Optional functions

Creating a digest	`SIGNewContext`
	`SIGDigestPrepare`
	`SIGProcessData`
	`SIGDigest`
	`SIGDisposeContext`	

Creating a Full Signature

When the user wants to sign a document or a portion of a document, you are responsible for knowing the location and extent of the data to be signed and for attaching or associating the full signature with that data once the signature is created. The Digital Signature Manager expects you to provide a pointer to the data, a pointer to a memory block where it is to place the full signature, and a context pointer.

To create a signature, follow these steps:

1. First, call the `SIGNewContext` function to allocate and initialize a context. The function returns a context pointer. If the Digital Signature Manager is not already in memory, the Operating System loads it into memory.
2. Call the `SIGSignPrepare` function, passing it the context pointer. It opens the signer file you specify; if you do not specify one, it opens the default signer file, which is the last signer file used. If there is no default signer file, it prompts the user for a signer-file location. It also prompts the user for the password needed to decrypt the signer's private key. It returns to you the size that the full signature will be.
3. Call the `SIGProcessData` function as many times as necessary to process all of the data to be signed. Either move a pointer through the data each time you call the function, or create a buffer and put blocks of data into it. The `SIGProcessData` function creates a digest of the data to be signed.
4. Create a properly sized memory block to hold the signature, and call the `SIGSign` function. It encrypts the digest, assembles the full signature, and puts it in the memory block you allocated. The `SIGSign` function periodically calls a callback routine that you may provide, so that you can notify the user of the progress of the signing operation or perform other background tasks.
5. If you are finished creating signatures for the current signer, go on to the next step. If you are creating additional signatures on different data sets for the same signer, repeat steps 3 and 4 for each signature in turn. The user will not be prompted for a password or signer-file location as each additional signature is created.
6. When you are finished creating signatures for the current user, call the `SIGDisposeContext` function to release the memory used by the signing routines and to release the Digital Signature Manager from memory. The next time you call `SIGSignPrepare`, the user is prompted once more for a password.

Listing 6-1 shows an example of a function that creates a full signature for a piece of data. This function requires an application-defined function named DoGetDataToProcess, which cycles through all the data that is to be signed. At the end of the SignData function is a call to another application-defined function named DoSaveSignature, which controls how and where to save the signature.

Listing 6-1 A sample signature-creation routine

```
OSErr SignData()
{
   OSErr                error;
   Boolean              moreToSign;
   Size                 signatureSize;
   Size                 dataSize;
   SIGSignaturePtr      signature = NULL;
   SIGContextPtr        context = NULL;
   Ptr                  dataBuffer = NULL;

   do {
      /* Allocate a new context and prepare it for signing. */

      if ((error = SIGNewContext(&context)) != noErr)
         break;

      if ((error = SIGSignPrepare(context, (FSSpecPtr)NULL, "\p",
            &signatureSize)) != noErr)
         break;

      /* Retrieve the data to be signed, in your application-specific way
         and pass it to the toolbox to generate the digest for our
         signature. */

      /* NOTE: DoGetDataToProcess can be the same function for signing and
         verifying. */

      do {
         if (error = DoGetDataToProcess(&dataBuffer, &dataSize, &moreToSign))
            break;

         if (error = SIGProcessData(context, dataBuffer, dataSize))
            break;

      } while (moreToSign);

      if (error != noErr)/* if encountered error above, go all the way out */
         break;
```

```
        /* Allocate a buffer of the size returned from SIGSignPrepare to hold
           the signature and create the signature by passing the buffer to
           SIGSign. */

        signature = (SIGSignaturePtr)NewPtr(signatureSize);

        if (error = MemError())
           break;

        if (error = SIGSign(context, signature, (SIGStatusProcPtr)NULL))
           break;

        /* Save the signature in your application-specific way. */

        error = DoSaveSignature(signature, signatureSize);

     } while (0);

     /* Free the context now, which forces user to reenter the password next
        time the SIGSignPrepare call is made. */

     if (context != NULL) SIGDisposeContext(context);

     if (dataBuffer != NULL) DisposPtr(dataBuffer);

     if (signature != NULL) DisposPtr((Ptr)signature);

     return error;
}
```

Verifying a Full Signature

When the user wants to verify the signature on a document or a portion of a document, you are responsible for knowing the location, processing order, and extent of the data to be verified, and for locating the full signature that applies to that data. The Digital Signature Manager expects you to provide a pointer to the data, a pointer to the full signature, the signature size, and a context pointer.

To verify a signature, follow these steps:

1. First, call the `SIGNewContext` function to allocate and initialize a context. The function returns a context pointer. If the Digital Signature Manager is not already in memory, the Operating System loads it into memory.
2. Call the `SIGVerifyPrepare` function, passing it a pointer to the signature to be verified, the signature size, and the context pointer. The `SIGVerifyPrepare` function periodically calls a callback routine that you may provide, so that you can notify the user of the progress of the operation or perform other background tasks.

The SIGVerifyPrepare function verifies the authenticity and currency of all certificates in the certificate set and returns the kSIGSignerErr result code if any of the certificates have been altered.

3. Call the SIGProcessData function as many times as necessary to process all of your data. Either move a pointer through your data each time you call the function, or create a buffer and put blocks of data into it. The SIGProcessData function creates a digest of the data whose signature is to be verified.
4. Call the SIGVerify function. It completes the digest and compares it with the digest in the signature.
5. Check the result code returned by the SIGVerify function to see if the verification was successful. A result code of noErr means the verification was successful and the signature is valid. A result code of kSIGInvalidCredentialErr means the verification was successful but the signer's credential is either pending or expired. A result code of kSIGVerifyFailedErr means the verification failed.
6. Call the SIGDisposeContext function to release the memory used by the verification routines and to release the Digital Signature Manager from memory. To verify another signature, you must start over from step 1.

After verifying a signature, you may want to get information about it and present that to the user. See "Getting Information From a Signature or Certificate" beginning on page 6-19.

Listing 6-2 shows an example of a function that verifies a signature. At the beginning of this function is a call to DoRetrieveSignature, an application-defined function that loads the signature in from where it is stored. The DoVerifyData function also requires an application-defined function named DoGetDataToProcess to cycle through all the data to be verified.

Listing 6-2 A sample signature-verification routine

```
OSErr DoVerifyData()
{
   OSErr             error;
   Boolean           moreToVerify;
   Size              signatureSize;
   Size              dataSize;
   SIGSignaturePtr   signature = NULL;
   SIGContextPtr     context = NULL;
   Ptr               dataBuffer = NULL;

   do {
      /* Get the signature and its size from wherever your application saved
         it. */
      if (error = DoRetrieveSignature(&signature, &signatureSize))
         break;

      /* Allocate a new context and prepare it for verifying. */
      if (error = SIGNewContext(&context))
         break;

      if (error = SIGVerifyPrepare(context, signature, signatureSize,
            (SIGStatusProcPtr)NULL))
         break;

      /* Get the data to be verified in your application-specific way, and
         pass it to the toolbox to generate a digest for verification. */

      /* NOTE: DoGetDataToProcess can be the same function for signing and
         verifying. */

      do {
         if (error = DoGetDataToProcess(&dataBuffer, &dataSize,
&moreToVerify))
            break;

         if (error = SIGProcessData(context, dataBuffer, dataSize))
            break;

         } while (moreToVerify);

      if (error)/* if encountered error above, go all the way out */
         break;

      /* Now, perform verification. */
      if (error = SIGVerify(context))
         break;

      /* Finally, display the name of the signer of the data. NOTE: you can
         call SIGShowSigner even if a kSIGInvalidCredentialErr was returned
         from SIGVerify. */

      error = SIGShowSigner(context, "\p");

   } while (0);

   /* Free the context. */

   if (context) SIGDisposeContext(context);

   if (dataBuffer) DisposPtr(dataBuffer);
```

```
    if (signature) DisposPtr((Ptr)signature);

    return error;
}
```

Creating a Simple (Unencrypted) Digest

As a convenience utility, the Digital Signature Manager allows you to create a digest of a document (or any stream of data you manipulate). The digest thus created cannot be encrypted or turned into a signature of the document, but its value as a sophisticated checksum makes it useful for other purposes, such as checking reliability in data transmission. And, like any data, the digest itself can be signed to ensure its integrity.

As one example, assume you are transmitting a massive document in separate blocks across a network. You want to ensure that the blocks are assembled in the right order at the receiving end. You can construct digests of individual blocks as they are sent and, after all the blocks have been sent, concatenate all the digests into a single file and send it. If the recipient has built a file of concatenated digests as the received blocks are reassembled, the concatenated digests should match each other if there has been no transmission or reassembly error. This method avoids the necessity of processing massive amounts of data at once, as would be necessary to create or verify a single signature on the entire document.

Creating a digest is similar to creating a signature. You first call the `SIGNewContext` function, then you call the `SIGDigestPrepare` function. Next you call `SIGProcessData` as many times as necessary to process all of your data. Finally you call `SIGDigest`, which returns the finished digest to you.

To create another digest, call the `SIGProcessData` as many times as necessary, then call the `SIGDigest` function. When you are finished creating digests, call the `SIGDisposeContext` function.

Getting Information From a Signature or Certificate

When you add a signature to a block of data or verify a signature, you are informed only of the success or failure of the operation. Neither you nor the user has direct access to any information in the signature—not even the name of the signer.

If you want to know (or tell the user) who created a signature, when it was signed, who issued the certificate to the signer, whether the signer's certificate has expired, or any other information available from the signature, you can call Digital Signature Manager routines that return that information.

After you successfully verify a signature, you can display a dialog box containing the full distinguished name of the signer by calling the `SIGShowSigner` function (page 6-46).

After you successfully verify a signature, after you call the `SIGSignPrepare` function to initiate the signing process, or after you call the `SIGSign` function to sign a block of data, you can call the `SIGGetSignerInfo` function (page 6-48) to determine when a block of data was signed and how many certificates constitute the certificate set for the signature. The SIGGetSignerInfo function also tells you whether the entire certificate set is valid and, if not, whether it has expired or has not yet become valid.

You can use the SIGGetCertInfo function (page 6-49) to obtain the beginning and ending dates of a certificate's validity, and the total number of attributes in the distinguished names of the certificate's signer and issuer.

You can use the SIGGetCertNameAttributes function (page 6-51) and the SIGGetCertIssuerNameAttributes function (page 6-52) to obtain the attributes that compose the distinguished names of the certificate's signer and issuer.

To obtain complete information on a newly applied or verified signature, you might follow a procedure something like this:

1. Call the SIGGetSignerInfo function to get the date of the signing and the total number of certificates in the full signature.
2. Call the SIGGetCertInfo function for the first certificate in the signature to get the dates for which the certificate is valid, the serial number of the certificate, and the number of name attributes in the distinguished name of the certificate.
3. Call the SIGGetCertNameAttributes function once for each name attribute in the certificate to get the full distinguished name for each certificate. This function returns the string for each attribute and the type of the attribute. It also specifies whether the attribute is the same level in the name hierarchy as the previous attribute (See "About Public-Key Certificates" beginning on page 6-8 for a description of distinguished names.)
4. Repeat steps 2 and 3 for each additional certificate in the certificate set. The certificates are always in order: the signer's certicate is first, the issuer of the signer's certificate is next, and so forth.
5. You can use the SIGGetCertIssuerNameAttributes function to get the full distinguished name of the prime issuer.

Listing 6-3 is an example of a function that extracts information from a certificate set. The DoDisplayCertificateSet sample function displays the name of the signer of a verified signature and searches through a certificate set, displaying information about the owner of each certificate. The DoDisplayCertificateSet function assumes that the input is a valid context that has gone through a successful call to either the SIGVerify, SIGSignPrepare, or SIGSign functions.

In addition, the DoDisplayCertificateSet function requires the following support functions to actually display the data to the user: DoDisplaySignatureInfo, DoDisplayCertificateInfo, and DoDisplayCertNameAttribute.

Listing 6-3 A sample routine that returns information in a certificate set

```
OSErr DoDisplayCertificateSet(SIGContextPtr context)
{
   unsigned short           attrIndex;
   SIGNameAttributesInfo    attrInfo;
   unsigned short           certIndex;
   SIGCertInfo              certInfo;
   SIGSignerInfo            signerInfo;
   OSErr                    error;

   do {
      /* Get and display general signature information first. */

      if (error = SIGGetSignerInfo(context, &signerInfo))
         break;

      DoDisplaySignatureInfo(&signerInfo);

      /* Traverse entire certificate set and for each certificate, display
         its certificate information. Then traverse the name attribute
         information for that certificate and display the attributes. */

      for (certIndex = kSIGSignerCertIndex; certIndex < signerInfo.certCount;
            certIndex++)
         {
         if (error = SIGGetCertInfo(context, certIndex, &certInfo))
            break;
         DoDisplayCertificateInfo(&certInfo);
         for (attrIndex = 0; attrIndex < certInfo.certAttributeCount;
               attrIndex++)
         {
            if (error = SIGGetCertNameAttributes(context, certIndex,
                  attrIndex, &attrInfo))
               break;

            DoDisplayCertNameAttribute(&attrInfo);
         }
      }

      /* Finally, display the root issuers' name attributes. */
      /* NOTE: there's no certificate information for the root; it's always
         valid.*/

      for (attrIndex = 0; attrIndex < certInfo.issuerAttributeCount;
            attrIndex++)
      {
         if (error = SIGGetCertIssuerNameAttributes(context, certIndex-1,
               attrIndex, &attrInfo))
            break;

         DoDisplayCertNameAttribute(&attrInfo);
      }
   } while (0);
   return error;
}
```

Dealing With Standard Signatures in Files

On the desktop, a user can add a standard signature to any file by dragging the icon for the file to be signed onto the icon of his or her signer file. When a user signs a file this way, the Digital Signature Manager adds a resource of type `'dsig'` to the resource fork of the file. Whenever you open a file, you should use the `SIGFileIsSigned` function to determine if the file contains such a signature. If a file contains a standard signature, you should not allow the user to alter the file without first displaying a dialog box warning that the file has been signed and that changing the file in any way will invalidate the signature. You should also not make any changes of your own to the file, such as saving a new window position, unless the user has chosen to allow changes that invalidate the signature.

All resources in the file are also signed, except any resources of type `'nods'` (no digital signature). You can store anything that you don't want to be signed in this resource, such as a new window position, and verification will still work.

Note
The '`dsig`' resource is mentioned here for your information only and may change in the future. Therefore, any attempt to manipulate this resource directly could cause incompatibilities with future versions of the Digital Signature Manager. ◆

You can verify a standard signature by calling the `SIGVerifyFile` function. You can add a standard signature to a file from within your application or replace an existing standard signature in a file by calling the `SIGSignFile` function. A user can also use the Finder to verify the signature in a file signed in this way.

You must call the SIGNewContext function before you call either the SIGSignFile function or the SIGVerifyFile function. To add a standard signature to a file, you must call the SIGSignPrepare function and the SIGSignFile function. The SIGSignFile function processes the data and adds the signature to the file. To verify a standard signature, you call the SIGVerifyFile function. You do not have to call the SIGProcessData function when you are working with standard signatures in files. You cannot add a standard signature to a file or verify such a signature if the file is in use.

Digital Signature Manager Reference

This section describes the data types and routines provided by the Digital Signature Manager and the interface to a status callback routine that you may provide.

Constants and Data Types

This section describes the constants and data types that are used by the SIGGetSignerInfo and SIGGetCertInfo functions to return information about signers and certificates. The SIGDigestData data type is described with the SIGDigest routine on page 6-44.

Signer Information Structure

The SIGGetSignerInfo function (page 6-48) uses a signer information structure to return information about a signature. The signer information structure is defined by the SIGSignerInfo data type.

```
struct SIGSignerInfo
{
   unsigned long        signingTime;      /* local sign time */
   unsigned long        certCount;        /* # of certificates
                                             in the set */
   unsigned long        certSetStatusTime;/* expiration time*/
   SIGSignatureStatus   signatureStatus;  /* certificate status */
};
```

Field descriptions

`signingTime` The time at which the data was signed. The time is in standard Macintosh format: the number of seconds elapsed since Midnight, January 1, 1904. The time is converted from Greenwich Mean Time (GMT) to the local time of the user's Macintosh. To convert to local time, the AOCE toolbox uses the local system clock and Map control panel on the signer's Macintosh computer. Thus, the time cannot be considered reliable.

`certCount` The number of certificates in the certificate set.

`certSetStatusTime`

If all the certificates in the certificate set are valid, this field holds the expiration time of the certificate that will be the first to expire. If one or more certificates have expired, this field holds the time when the first certificate in the set expired. If none of the certificates have expired but one or more is not yet valid, this field holds the time that the last pending certificate will become valid. The time is given as the number of seconds elapsed since midnight, January 1, 1904.

`signatureStatus`

If all the certificates in the certificate set are valid, this field holds the value `kSIGValid`. If any of the certificates have expired since the data was signed, this field holds the value `kSIGExpired`. If any of the certificates had already expired before the data was signed, this field holds the value `kSIGInvalid`. If none of the certificates have expired but any have not yet become valid, this field holds the value `kSIGPending`.

This field can have any of the following values:

```
enum {
   kSIGValid,     /* all valid */
   kSIGPending,   /* none expired; some pending
                     or unknown */
   kSIGExpired,   /* some expired, unknown, or
                     pending */
   kSIGInvalid    /* some invalid, pending, expired
                     or unknown */
};

typedef unsigned short SIGSignatureStatus;
```

Certificate Information Structure

The `SIGGetCertInfo` function (page 6-49) uses a certificate information structure to return information about a specific certificate in a signature. The certificate information structure is defined by the `SIGCertInfo` data type.

```
struct SIGCertInfo
{
   unsigned long  startDate;           /* validity start date */
   unsigned long  endDate;             /* validity end date */
   SIGCertStatus  certStatus;          /* certificate status */
   unsigned long  certAttributeCount;  /* number of name
                                          attributes in cert*/
   unsigned long  issuerAttributeCount;/* # of name attributes
                                          in cert’s issuer */
   Str255         serialNumber;        /* cert serial number */
};
```

Field descriptions

`startDate` The time at which the certificate became (or will become) valid. The time is in standard Macintosh format: the number of seconds elapsed since midnight, January 1, 1904.

`endDate` The expiration time of the certificate in seconds since midnight, January 1, 1904.

`certStatus` The status of the certificate: `kSIGValid`, `kSIGPending`, or `kSIGExpired`.

`certAttributeCount`
The number of attributes in the distinguished name for this certificate (see Table 6-1 on page 6-9). You can use the `SIGGetCertNameAttributes` function (page 6-51) to list the attributes.

`issuerAttributeCount`
The number of attributes in the distinguished name of the issuer of this certificate (see Table 6-1 on page 6-9). You can use the `SIGGetCertIssuerNameAttributes` function (page 6-52) to list the attributes.

`serialNumber` A certificate number assigned by the issuer.

Standard Signature Icon Suite

The Digital Signature Manager provides an icon suite that you use to represent a digital signature in your document. This suite contains all bit depths and sizes.

```
#define kSIGSignatureIconResID          -16797

#define kSIGValidSignatureIconResID     -16799

#define kSIGInvalidSignatureIconResID   -16798
```

Name Attribute Information Structure

The SIGGetCertNameAttributes function (page 6-51) and the SIGGetCertIssuerNameAttributes function (page 6-52) use a name attribute information structure to return information about a name attribute. The name attribute information structure is defined by the SIGNameAttributesInfo data type.

```
struct SIGNameAttributesInfo
{
   Boolean                 onNewLevel;
   SIGNameAttributeType    attributeType;
   ScriptCode              attributeScript;
   Str255                  attribute;
};
```

Field descriptions

onNewLevel
A Boolean value that indicates whether the name attribute is at the same level of the name hierarchy as the previous value returned.

attributeType
The type of attribute returned.

attributeScript
The script code for the name attribute. Script codes are defined by the Script Manager.

attribute
The name attribute value.

The attributeType field can have any of the following values:

```
enum {
            kSIGCountryCode,
            kSIGOrganization,
            kSIGStreetAddress,
            kSIGState,
            kSIGLocality,
            kSIGCommonName,
            kSIGTitle,
            kSIGOrganizationUnit,
            kSIGPostalCode
};

typedef unsigned short SIGNameAttributeType;
```

You can use the hierarchy information in the `onNewLevel` parameter to arrange the distinguished name for display to the user.

Distinguished names and name hierarchies are described in detail in "About Public-Key Certificates" beginning on page 6-8.

Digital Signature Manager Functions

You can use Digital Signature Manager functions to perform the following tasks: creating a signature (page 6-31), verifying a signature (page 6-38), creating an unencrypted digest (page 6-43), signing a file (page 6-36), and verifying a file (page 6-41). All of these tasks, except signing and verifying a file, require you to process data (page 6-30). You begin each of these operations by creating a new context and end the operation by disposing of the context (page 6-28). After you prepare a context for a signature, create a signature, or verify a signature, you can extract information from the certificate or signature (page 6-45).

Assembly-Language Interface

To call a Digital Signature Manager function from assembly language, you must do the following:

1. Allot space for the function result and all routine parameters (in Pascal calling-convention order) on the stack.
2. In the D0 register, put a long word consisting of the parameter word count for the routine followed by the routine selector. The parameter word count indicates how many words of parameters you are placing on the stack; for example, if the function has two parameters and each is a pointer, the parameter word count for the function is $0004.
3. Call the Digital Signature Manager trap, $AA5D.

Each routine description in the following sections lists the parameter word count and routine selector for that routine.

Creating and Disposing of a Context

The Digital Signature Manager uses a private data structure called a ***context*** to hold information and the results of calculations while it is processing data. Before you call Digital Signature Manager routines to perform a specific task, you must call the `SIGNewContext` function to create a context and obtain a context pointer. To free the memory used by the context, call the `SIGDisposeContext` function.

You can use a new context for any type of operation; however, once you have called the first task-specific function (`SIGSignPrepare`, `SIGVerifyPrepare`, or `SIGDigestPrepare`), you can use the context only with other functions associated with that task. Table 6-2 on page 6-12 summarizes the Digital Signature Manager tasks and the functions required to perform each task.

SIGNewContext

The `SIGNewContext` function creates a new context and returns a context pointer.

```
pascal OSErr SIGNewContext (SIGContextPtr *context);
```

`context` A pointer to the new context created by this function.

DESCRIPTION

You must pass the context pointer returned by this function to either the `SIGSignPrepare`, `SIGVerifyPrepare`, or `SIGDigestPrepare` function.

SPECIAL CONSIDERATIONS

This function causes the Digital Signature Manager to be loaded into memory if it is not already in memory.

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0002	$076C

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`memFullErr`	–108	Not enough room in heap

SEE ALSO

Use the `SIGDisposeContext` function (described next) to dispose of a context.

The `SIGNewContext` function is normally followed by either the `SIGSignPrepare` function (page 6-31), the `SIGVerifyPrepare` function (page 6-38), or the `SIGDigestPrepare` function (page 6-43).

SIGDisposeContext

The `SIGDisposeContext` function frees the memory used by a context.

```
pascal OSErr SIGDisposeContext (SIGContextPtr context));
```

`context` A pointer to the context you wish to dispose of.

DESCRIPTION

You must call the `SIGDisposeContext` function to dispose of a context when you are finished creating a signature, verifying a signature, creating a digest, or extracting information from a signature or certificate.

SPECIAL CONSIDERATIONS

Because this function removes the Digital Signature Manager (as well as the context) from memory, you must call this function even if the previous function returned an error.

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0002	$076D

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation

Processing Data to Generate a Digest

To process data during the creation or verification of a signature, or during the creation of a digest, call the `SIGProcessData` function one or more times.

SIGProcessData

The `SIGProcessData` function processes the data passed to it and revises the digest accordingly.

```
pascal OSErr SIGProcessData (SIGContextPtr context,
                              const void *data, Size dataSize);
```

`context`	A pointer to the context that you passed to the `SIGSignPrepare`, `SIGVerifyPrepare`, or `SIGDigestPrepare` function.
`data`	A pointer to a buffer containing the data to be processed.
`dataSize`	The number of bytes of data to be processed.

DESCRIPTION

Call the `SIGProcessData` function to generate a digest for a set of data. If you have more data than is convenient to process all at once, you can call the function several times, passing it a block of any size each time. Note, however, that it is more efficient to process data in large blocks than in small blocks.

You can place each block of data into a buffer, or you can change the `data` parameter each time to point at the next starting position in your data. You are responsible for keeping track of where the data is and how much of it to process during each call to the `SIGProcessData` function, and for knowing when all the data has been processed.

The data must be processed in the same order during the corresponding sign and verify operations but need not be processed in blocks of the same size. To the `SIGProcessData` function, the data is a continuous byte stream.

SPECIAL CONSIDERATIONS

You can call the `SIGProcessData` function at interrupt time; it does not move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0006	$0774

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation
`kSIGInternalsErr`	–1977	Bad digest, context, or signature

SEE ALSO

The `SIGProcessData` function is preceded by a call to `SIGSignPrepare` (page 6-31), `SIGVerifyPrepare` (page 6-38), or `SIGDigestPrepare` (page 6-43).

After calling `SIGProcessData`, you call either `SIGSign` (page 6-34), `SIGVerify` (page 6-40), or `SIGDigest` (page 6-44).

Creating a Signature

To create a full signature, first call the `SIGNewContext` function (page 6-28) to create a new context, then call the `SIGSignPrepare` function (described next).

Next, to sign some portion of the data in a file, call the `SIGProcessData` function (page 6-30) as many times as necessary to process all the data. When you are finished processing the data, call the `SIGSign` function (page 6-34). To create additional signatures for the same signer, you can call the `SIGProcessData` and `SIGSign` functions again, without first creating a new context or calling the `SIGSignPrepare` function.

If you want to add a standard signature to a file, call the `SIGSignFile` function (page 6-36) instead of the `SIGProcessData` and `SIGSign` functions. To add signatures to additional files, you can call the `SIGSignFile` function again, without first creating a new context or calling the `SIGSignPrepare` function.

When you no longer need the context you used for creating the signatures, call the `SIGDisposeContext` function (page 6-29).

This section describes the `SIGSignPrepare`, `SIGSign`, and `SIGSignFile` functions.

SIGSignPrepare

The `SIGSignPrepare` function notifies the Digital Signature Manager that you are about to create a signature.The function returns the size that the full signature will be when it is created.

```
pascal OSErr SIGSignPrepare (SIGContextPtr context,
                                  const FSSpec *signerFile,
                                  ConstStr255Param prompt,
                                  Size *signatureSize);
```

`context` A pointer to the context that the Digital Signature Manager will use while creating the signature. Call the `SIGNewContext` function to obtain the context pointer.

`signerFile`
A pointer to a file-specification structure for the user's signer file. If you specify `NULL` for this parameter, the function opens the previously used signer file, or, if there is no record of a previously used signer file, the function displays a Standard File dialog box prompting the user for the location of a signer file.

`prompt` A string to display in the dialog box that prompts the user for a password. Pass a zero-length Pascal-style string to use the default prompt.

`signatureSize`
A pointer to the size of the signature that is to be created. The function returns this parameter.

DESCRIPTION

The `SIGSignPrepare` function displays a password-prompting dialog box into which the user types the private-key password. The function displays a Standard File dialog box prompting the user for a signer file if you do not specify a signer file in the `signerFile` parameter and there is no default signer file.

If you pass `NULL` in the `signerFile` parameter, the first time the user signs something, the function displays a Standard File dialog box prompting the user for the location of the signer file. The Digital Signature Manager then stores an alias to that file in the user's Preferences folder. The next time you specify `NULL` in the `signerFile` parameter, the `SIGSignPrepare` function uses that signer file as the default and does not display the standard file dialog box.

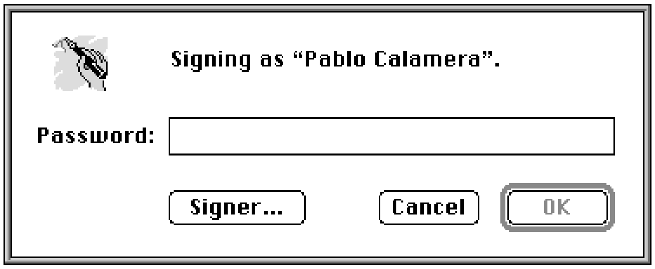
If you already know the location of the user's signer file, you can bypass the Standard File dialog box by passing a pointer to the signer file's file-specification structure in the `signerFile` parameter. You can also use this procedure to override the default signer file.

The `prompt` parameter can contain whatever string you wish displayed in the dialog box to prompt the user for a private-key password. Use the parameter-text designator `^1` for the user's name; the Digital Signature Manager replaces `^1` in your string with the user's common name or title (depending on whether the user is signing as a person or as an organizational role—see Table 6-1 on page 6-9) as it appears in the signer file. If you pass a zero-length string, the function uses the default string.

The password-prompting dialog box also contains a Signer button that allows the user to select a different signer file. Figure 6-5 shows how the dialog box would appear to a user whose common name is Pablo Calamera.

Note
If you specify a signer file to use, the password dialog box does not contain a Signer button allowing users to switch signer files. ◆

Figure 6-5 The password-prompting dialog box



Signing as “Pablo Calamera”.

Password:

This function returns the size the signature will be once it is created. Use the result to allocate memory for the signature before calling the `SIGSign` function.

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

This function is stack-intensive, requiring approximately 7 KB of memory for its stack.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	**Routine selector**
$0008	$076E

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`memFullErr`	–108	Not enough room in heap zone
`userCanceledErr`	–128	User canceled password-prompt dialog box
`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation
`kSIGSignerErr`	–1975	Problem with the signer file or signature
`kSIGPasswordErr`	–1976	Password is incorrect
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGContextPrepareErr`	–1979	Context already prepared by `SIGVerifyPrepare`, `SIGSignPrepare`, or `SIGDigestPrepare`
`kSIGConversionErr`	–1981	Unable to convert an attribute to Macintosh format
`kSIGSignerNotValidErr`	–1982	Signer file has either expired or is not yet valid

SEE ALSO

Before you call the `SIGSignPrepare` function, you must call the `SIGNewContext` function (page 6-28) to create a new context.

After calling the `SIGSignPrepare` function, you can extract information from the certificate set; see "Getting Information From a Signature or Certificate" beginning on page 6-45.

After you call the `SIGSignPrepare` function, call the `SIGProcessData` function (page 6-30) as many times as necessary to process all the data.

SIGSign

The `SIGSign` function creates a full signature for the data most recently processed by the `SIGProcessData` function, using signer-file information from the most recent call to the `SIGSignPrepare` function.

```
pascal OSErr SIGSign (SIGContextPtr context,
                      SIGSignaturePtr signature,
                      SIGStatusProcPtr statusProc);
```

`context` The context pointer that you passed to the `SIGSignPrepare` function.

`signature` A pointer to a buffer you provide to hold the signature returned by the function. Use the result of the `SIGSignPrepare` function to allocate a buffer of the correct size.

`statusProc`
A pointer to a callback routine you may provide to notify the user of the progress of the signature creation or to perform other background tasks. Specify `NULL` for this parameter if you do not wish to provide a callback routine.

DESCRIPTION

Call this function after having called the `SIGProcessData` function enough times to finish processing the document or data that is to be signed. After creating a signature, `SIGSign` places it in the buffer pointed to by the `signature` parameter.

Because the `SIGSign` function can take a long time to complete, you can provide a pointer to a callback routine to notify the user of the progress of the operation, allow the user to cancel it, and perform background tasks such as spinning the cursor.

To create additional signatures for the same signer, you can call the `SIGProcessData` and `SIGSign` functions again, without first creating a new context or calling the `SIGSignPrepare` function. Call the `SIGDisposeContext` function when you have finished creating signatures with that signer.

Note
You should call the SIGDisposeContext function as soon as possible after you finish creating signatures so that the Operating System can free the memory used by the Digital Signature Manager. ◆

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0006	$076F

RESULT CODES

noErr	0	No error
paramErr	–50	Illegal parameter value
userCanceledErr	–128	User canceled signing process
kSIGOperationIncompatibleErr	–1970	Context in use for different type of operation
kSIGSignerErr	–1975	Problem with the signer file or signature
kSIGInternalsErr	–1977	Bad digest, context, or signature

SEE ALSO

You call the SIGSign function after calling SIGSignPrepare (page 6-31) to initiate the signing process and SIGProcessData (page 6-30) to process the data.

You may provide a callback status routine when you call the SIGSign function; see “Application-Defined Function” on page 6-54.

After calling the SIGSign function, you can extract information from the signature; see “Getting Information From a Signature or Certificate” beginning on page 6-45.

As soon as possible after you finish creating signatures, call the SIGDisposeContext (page 6-29) function to dispose of the context and to allow the Operating System to remove the Digital Signature Manager from memory.

SIGSignFile

The SIGSignFile function adds a standard signature to a file.

```
pascal OSErr SIGSignFile (SIGContextPtr context,
                          Size signatureSize,
                          const FSSpec *fileSpec,
                          SIGStatusProcPtr statusProc);
```

`context` The context pointer that you passed to the SIGSignPrepare function.

`signatureSize` The size of the signature as returned by the SIGSignPrepare function.

`fileSpec` A pointer to the file system specification structure for the file to which you want to add a signature.

`statusProc` A pointer to a callback routine you may provide to notify the user of the progress of the signature creation or to perform other background tasks. Specify NULL for this parameter if you do not wish to provide a callback routine.

DESCRIPTION

The SIGSignFile function processes a signature for a complete file and places it in the resource fork of the file as a resource of type 'dsig'. You must call the SIGSignPrepare function before calling the SIGSignFile function.

Note

The 'dsig' resource is mentioned here for your information only. Because it may change in the future, you should not attempt to manipulate this resource directly. Any change could cause incompatibilities with future versions of the Digital Signature Manager. ◆

A signature you add to a file using this function is identical to one added by the Finder when the user drags the icon for the file onto the icon of their signer file. If the file is already signed, the SIGSignFile function creates a new signature and replaces the old one.

All resources in the file are also signed, except any resources of type 'nods' (no digital signature). You can store anything that you don't want to be signed in this resource, such as a new window position, and verification will still work.

Because the SIGSignFile function can take a long time to complete, you can provide a pointer to a callback routine to perform background tasks such as spinning the cursor and to allow the user to cancel the operation.

To sign additional files for the same signer, you can call the SIGSignFile function again, without first creating a new context or calling the SIGSignPrepare function.

Call the `SIGDisposeContext` function when you are finished signing files for that signer.

Note

You should call the `SIGDisposeContext` function as soon as possible after you finish creating signatures so that the Operating System can free the memory used by the Digital Signature Manager. ◆

IMPORTANT

The `SIGSignFile` function will not work on a file that is open. ▲

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0008	$09C5

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	Directory full
`dskFulErr`	–34	Disk full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad name error
`tmfoErr`	–42	Too many files open
`fnfErr`	–43	File not found
`fBsyErr`	–47	File is busy
`opWrErr`	–49	File already open with write permission
`paramErr`	–50	Illegal parameter value
`wrPermErr`	–61	File not available
`memFullErr`	–108	Not enough room in heap zone
`dirNFErr`	–120	Directory not found
`userCanceledErr`	–128	User canceled signing process
`addResFailed`	–194	Adding resource failed
`rmvResFailed`	–196	Removing resource failed
`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`afpAccessDenied`	–5000	Disk full

SEE ALSO

You call the `SIGSignFile` function after initiating the signing process with the `SIGSignPrepare` function (page 6-31).

You may provide a callback status routine when you call the SIGSign function; see "Application-Defined Function" on page 6-54.

You can use the SIGFileIsSigned function (page 6-45) to determine if a file already contains a standard signature.

As soon as possible after you finish creating signatures, call the SIGDisposeContext function (page 6-29) to dispose of the context and to allow the Operating System to remove the Digital Signature Manager from memory.

Verifying a Signature

When you use the Digital Signature Manager to verify a signature, it checks the validity of the certificate set, creates a digest of the data whose signature you wish to verify, and compares that digest to the digest in the signature.

To verify a signature of some portion of data in a file, first call the SIGNewContext function (page 6-28), then call the SIGVerifyPrepare function (described next). Next, call the SIGProcessData function (page 6-30) as many times as necessary to prepare a digest of the data. When you have finished processing the data, call the SIGVerify function (page 6-40) to compare the digest you prepared with the one in the signature.

To verify a standard signature in a file (that is, one added by the Finder or by the SIGSignFile function), first call the SIGNewContext function to create a new context, then call the SIGVerifyFile function (page 6-41).

When you are finished with the context you used for verifying the signature, call the SIGDisposeContext function (page 6-29).

This section describes the SIGVerifyPrepare function, the SIGVerify function, and the SIGVerifyFile function.

SIGVerifyPrepare

The SIGVerifyPrepare function notifies the Digital Signature Manager that you have a signature to be verified and initializes the verification process.

```
pascal OSErr SIGVerifyPrepare (SIGContextPtr context,
                               SIGSignaturePtr signature,
                               Size signatureSize,
                               SIGStatusProcPtr statusProc);
```

context
: A pointer to the context that the Digital Signature Manager will use while verifying the signature. Call the SIGNewContext function to obtain the context pointer.

signature
: A pointer to the full signature that is to be verified.

signatureSize
: The size of the signature that is to be verified.

`statusProc`

A pointer to a callback routine you may provide to notify the user of the progress of the verification operation or perform other background tasks. Specify `NULL` for this parameter if you do not wish to provide a callback routine.

DESCRIPTION

You must provide the `SIGVerifyPrepare` function with a pointer to the signature to be verified and the size of the signature. You may release the memory used by the signature after the `SIGVerifyPrepare` function has completed.

Because the `SIGVerifyPrepare` function verifies each certificate in the certificate set and reads in the digest, it can take a long time to complete. You can provide a pointer to a callback routine to perform background tasks such as spinning the cursor and to allow the user to cancel the operation.

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

This function is stack-intensive, requiring approximately 7 KB of memory for its stack.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0008	$0770

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`memFullErr`	–108	Not enough room in heap zone
`kSIGSignerErr`	–1975	Problem with the signer file or signature
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGContextPrepareErr`	–1979	Context already prepared by `SIGVerifyPrepare`, `SIGSignPrepare`, or `SIGDigestPrepare`
`kSIGNoDigestErr`	–1980	No digest in the signature

SEE ALSO

Before you call the `SIGVerifyPrepare` function, you must call the `SIGNewContext` function (page 6-28) to create a new context.

You may provide a callback status routine when you call the `SIGVerifyPrepare` function; see “Application-Defined Function” on page 6-54.

After you call the `SIGVerifyPrepare` function, call the `SIGProcessData` function (page 6-30) as many times as necessary to process all the data whose signature you wish to verify.

SIGVerify

The `SIGVerify` function tests the validity of the specified signature. To do so, it compares the digest in the signature with the digest you prepared by calling the `SIGProcessData` function. It also checks the validity of the credentials in the signature's certificate set.

```
pascal OSErr SIGVerify (SIGContextPtr context);
```

`context` The context pointer that you passed to the `SIGVerifyPrepare` function.

DESCRIPTION

Call this function after having called the `SIGProcessData` function enough times to finish processing data whose signature is to be verified. Note that you must process the data in the same sequence that it was processed when the signature was created.

Check the result code from this function to see if the signature verification was successful.

Note

You should call the `SIGDisposeContext` function as soon as possible after you finish verifying a signature so that the Operating System can free the memory used by the Digital Signature Manager. ◆

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	**Routine selector**
$0002	$0771

RESULT CODES

`noErr`	0	No error
`ParamErr`	–50	Illegal parameter value
`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation
`kSIGVerifyFailedErr`	–1972	Verification failed
`kSIGInvalidCredentialErr`	–1973	Verified OK but credential pending or expired
`kSIGInternalsErr`	–1977	Bad digest, context, or signature

SEE ALSO

You call the `SIGVerify` function after calling `SIGVerifyPrepare` (page 6-38) to initiate the signing process and `SIGProcessData` (page 6-30) to process the data.

After a successful verification, you can extract information from the signature; see "Getting Information From a Signature or Certificate" beginning on page 6-45.

As soon as possible after calling the `SIGVerify` function, call the `SIGDisposeContext` (page 6-29) function to dispose of the context and allow the Operating System to remove the Digital Signature Manager from memory.

SIGVerifyFile

The `SIGVerifyFile` function verifies a standard signature in a file.

```
pascal OSErr SIGVerifyFile (SIGContextPtr context,
                            const FSSpec *fileSpec,
                            SIGStatusProcPtr statusProc);
```

`context` A pointer to the context that the Digital Signature Manager will use while verifying the signature. Call the `SIGNewContext` function to obtain the context pointer.

`fileSpec` A pointer to the file system specification structure for the file whose signature you want to verify.

`statusProc`
A pointer to a callback routine you may provide to notify the user of the progress of the verification operation or perform other background tasks. Specify `NULL` for this parameter if you do not wish to provide a callback routine.

DESCRIPTION

If a file contains a standard signature, you can use the `SIGVerifyFile` function to verify it.

Because the `SIGVerifyFile` function verifies each certificate in the certificate set and reads in the digest, it can take a long time to complete. You can provide a pointer to a callback routine to perform background tasks such as spinning the cursor and to allow the user to cancel the operation.

Note
You should call the `SIGDisposeContext` function as soon as possible after you finish verifying a signature so that the Operating System can free the memory used by the Digital Signature Manager. ◆

IMPORTANT

You cannot use the `SIGVerifyFile` function on a file that is in use by another application. ▲

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0006	$09C6

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	Directory full
`dskFulErr`	–34	Disk full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad name error
`tmfoErr`	–42	Too many files open
`fnfErr`	–43	File not found
`fBsyErr`	–47	File is busy
`opWrErr`	–49	File already open with write permission
`paramErr`	–50	Illegal parameter value
`permErr`	–54	Permissions error on file open
`memFullErr`	–108	Not enough room in heap zone
`dirNFErr`	–120	Directory not found
`kSIGSignerErr`	–1975	Problem with the signer file or signature
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGContextPrepareErr`	–1979	Context already prepared by `SIGVerifyPrepare`, `SIGSignPrepare`, or `SIGDigestPrepare`
`kSIGNoDigestErr`	–1980	No digest in the signature
`kSIGNoSignature`	–1983	Standard file signature not found

SEE ALSO

Before you call the `SIGVerifyFile` function, you must call the `SIGNewContext` function (page 6-28) to create a new context.

You may provide a callback status routine when you call the `SIGVerifyFile` function; see “Application-Defined Function” on page 6-54.

You can call the `SIGFileIsSigned` function (page 6-45) to determine if a file contains a standard signature.

Creating a Digest

You can create an unencrypted digest of a document without creating a digital signature. To create a digest, first call the `SIGNewContext` function (page 6-28) to create a new context, then call the `SIGDigestPrepare` function (described next). Next, call the `SIGProcessData` function (page 6-30) as many times as necessary to process all the data. When you have finished processing the data, call the `SIGDigest` function (page 6-44). To create additional digests, you can call the `SIGProcessData` and `SIGDigest` functions again, without first creating a new context or calling the `SIGDigestPrepare` function. When you no longer need the context you used for creating the digests, call the `SIGDisposeContext` function (page 6-29).

This section describes the `SIGDigestPrepare` function and the `SIGDigest` function.

SIGDigestPrepare

The `SIGDigestPrepare` function notifies the Digital Signature Manager that you are about to create a digest.

```
pascal OSErr SIGDigestPrepare (SIGContextPtr context);
```

`context` A pointer to the context that the Digital Signature Manager will use while creating the digest. Call the `SIGNewContext` function to obtain the context pointer.

DESCRIPTION

The `SIGDigestPrepare` function notifies the Digital Signature Manager that the context is to be used to create a digest.

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	**Routine selector**
$0002	$0772

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`memFullErr`	–108	Not enough room in heap zone
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGContextPrepareErr`	–1979	Context already prepared by `SIGVerifyPrepare`, `SIGSignPrepare`, or `SIGDigestPrepare`

SEE ALSO

Before you call the `SIGDigestPrepare` function, you must call the `SIGNewContext` function (page 6-28) to create a new context.

After you call the `SIGDigestPrepare` function, call the `SIGProcessData` function (page 6-30) as many times as necessary to process all the data.

SIGDigest

The `SIGDigest` function returns a pointer to a digest of the data most recently processed by the `SIGProcessData` function.

```
pascal OSErr SIGDigest (SIGContextPtr context,
                        SIGDigestData digest);
```

`context` The context pointer that you passed to the `SIGDigestPrepare` function.

`digest` A `SIGDigestData` array that you provide to hold the result of this function.

DESCRIPTION

You can call the `SIGProcessData` function and the `SIGDigest` function as many times as you wish to prepare digests of data without calling the `SIGDigestPrepare` function again or creating a new context.

You must allocate a `SIGDigestData` structure to hold the digest before calling this function.

```
#define kSIGDigestSize 16

typedef Byte SIGDigestData[kSIGDigestSize], *SIGDigestDataPtr;
```

Note
You should call the `SIGDisposeContext` function as soon as possible after you finish making digests so that the Operating System can free the memory used by the Digital Signature Manager. ◆

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0004	$0773

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation
`kSIGInternalsErr`	–1977	Bad digest, context, or signature

SEE ALSO

You call the `SIGDigest` function calling `SIGDigestPrepare` (page 6-43) to initiate the digest process and `SIGProcessData` (page 6-30) to process the data.

As soon as possible after you finish preparing digests, call the `SIGDisposeContext` (page 6-29) function to dispose of the context and to allow the Operating System to remove the Digital Signature Manager from memory.

Getting Information From a Signature or Certificate

The first routine in this section, `SIGFileIsSigned`, indicates whether a file includes a standard signature. Use the other routines in this section to get information about the date, size, or contents of a full signature and its components.

SIGFileIsSigned

The `SIGFileIsSigned` function indicates whether a file contains a standard signature.

```
pascal OSErr SIGFileIsSigned(const FSSpec *fileSpec);
```

`fileSpec` A pointer to the file system specification structure for the file that you want to check for a signature.

DESCRIPTION

A file that has been signed by the finder or by the `SIGSignFile` function contains a digital signature in the form of a resource of type `'dsig'`. The `SIGFileIsSigned` function checks a file for this resource and returns a result code of `noErr` if it finds one. If the function finds no such resource, it returns the result code `kSIGNoSignature`.

Note
The 'dsig' resource is mentioned here for your information only. Because it may change in the future, you should not attempt to manipulate this resource directly. Any change could cause incompatibilities with future versions of the Digital Signature Manager. ◆

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0002	$09C4

RESULT CODES

noErr	0	File is signed
kSIGNoSignature	–1983	Standard file signature not found

SEE ALSO

You can call the SIGVerifyFile function (page 6-41) to verify a standard signature in a file.

You can add a standard signature to a file by calling the SIGSignFile function (page 6-36).

SIGShowSigner

The SIGShowSigner function displays the entire distinguished name of the signer of a block of data. You can call this function only after successfully verifying a signature.

```
pascal OSErr SIGShowSigner(SIGContextPtr context,
                           ConstStr255Param prompt);
```

context
: The context pointer you used the last time you called the SIGVerify or SIGVerifyFile function.

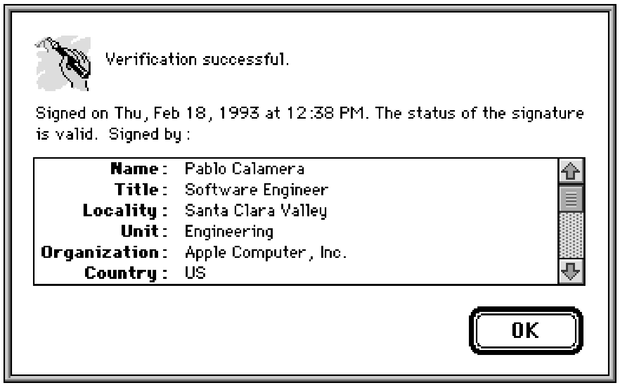
prompt
: The prompt you want to appear in the dialog box displayed by the SIGShowSigner function. If you specify a zero-length Pascal string for this parameter, the function displays a default string.

DESCRIPTION

After you call the `SIGVerify` function and it returns either the `noErr` or the `kSIGInvalidCredentialErr` result code, you can call the `SIGShowSigner` function to display a modal dialog box with the full distinguished name of the signer. Figure 6-6 shows an example of this dialog box.

Note
The time displayed is the local time determined by the user's local system clock and Map control panel. ◆

Figure 6-6 Show-signer dialog box



Verification successful.

Signed on Thu, Feb 18, 1993 at 12:38 PM. The status of the signature is valid. Signed by :

Name : Pablo Calamera
Title : Software Engineer
Locality : Santa Clara Valley
Unit : Engineering
Organization : Apple Computer, Inc.
Country : US

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0004	$0775

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`memFullErr`	–108	Not enough room in heap zone
`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation
`kSIGSignerErr`	–1975	Problem with the signer file or signature
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGConversionErr`	–1981	Unable to convert to Macintosh format

SEE ALSO

You cannot call the SIGShowSigner function until after you have called the SIGVerify function (page 6-40) or the SIGVerifyFile function (page 6-41).

Distinguished names are defined in Table 6-1 on page 6-9.

SIGGetSignerInfo

The SIGGetSignerInfo function returns information about a signer.

```
pascal OSErr SIGGetSignerInfo (SIGContextPtr context,
                               SIGSignerInfo *signerInfo);
```

context — The context pointer you used the last time you called the SIGVerify, SIGVerifyFile, SIGSignPrepare, or SIGSign function.

signerInfo — A pointer to a signer information structure returning information about the signer. You must allocate this structure.

DESCRIPTION

The SIGGetSignerInfo function returns information about the signer whose context pointer you provide to the function. You can call the SIGGetSignerInfo function after you call the SIGSignPrepare function, the SIGSign function, the SIGVerify function, or the SIGVerifyFile function.

You allocate a signer information structure, and the function fills it in. The signer information structure tells you the time (and date) that the data was signed, the number of certificates in the certificate set, and the status of the certificate set. (Note that if you call the SIGGetSignerInfo function immediately after calling the SIGSignPrepare function, the time of signing is meaningless because the data has not yet been signed.) If all the certificates are valid, the structure lists the earliest expiration date for any certificate in the set. If one or more certificates have expired, the structure lists the expiration date of the one that expired first. If none of the certificates have expired but one or more are not yet valid, the structure lists the date at which the last certificate to become valid will do so.

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	**Routine selector**
$0004	$0776

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`kSIGCertificateQueryDenied`	–1971	Can’t query certificates with this context
`kSIGSignerErr`	–1975	Problem with the signer file or signature
`kSIGInternalsErr`	–1977	Bad digest, context, or signature

SEE ALSO

You can call the `SIGGetSignerInfo` function after you call the `SIGSignPrepare` function (page 6-31), the `SIGSign` function (page 6-34), the `SIGVerify` function (page 6-40), or the `SIGVerifyFile` function (page 6-41).

The signer information structure is described on page 6-23.

You can call the `SIGShowSigner` function (page 6-46) to display a modal dialog box showing the distinguished name of the signer of a verified signature.

You can call `SIGGetCertInfo` function (described next) to get more information about any certificate in the certificate set, including that of the signer.

SIGGetCertInfo

The `SIGGetCertInfo` function returns information about a specific certificate in a certificate set.

```
pascal OSErr SIGGetCertInfo (SIGContextPtr context,
                             unsigned long certIndex,
                             SIGCertInfo *certInfo);
```

`context` The context pointer you used the last time you called the `SIGVerify`, `SIGVerifyFile`, `SIGSignPrepare`, or `SIGSign` function.

`certIndex` The index number of the certificate about which you want information. The certificates are always in order: the signer’s certicate has index number 0, the issuer of the signer’s certificate has index number 1, and so forth. You can use the `SIGGetSignerInfo` function to determine the total number of certificates in the certificate set.

`certInfo` A pointer to a certificate information structure returning information about the certificate. You must allocate this structure.

DESCRIPTION

The `SIGGetCertInfo` function returns information about one certificate in the certificate set of the signer whose context pointer you provide to the function. You allocate a certificate information structure and specify the index number of the certificate about which you want information, and the function fills in the structure. The certificate information structure tells you the beginning and ending dates for the validity period of the certificate, the status of the certificate (pending, expired, or valid), the number of attributes in the distinguished name of the signer of the certificate, the number of attributes in the distinguished name of the issuer of the certificate, and the serial number of the certificate.

The serial number and issuer name together uniquely identify a certificate. This information may be of use to a user who needs to contact the issuing organization (for example, to ensure a certificate has not been revoked).

The certificate of the signer of the data always has index number 0. You can use the following constant for this number:

```
#define kSIGSignerCertIndex 0
```

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0006	$0777

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`kSIGCertificateQueryDenied`	–1971	Can’t query certificates with this context
`kSIGIndexErr`	–1974	Index value is outside allowable range
`kSIGInternalsErr`	–1977	Bad digest, context, or signature

SEE ALSO

You can call the `SIGGetCertInfo` function after you call the `SIGSignPrepare` function (page 6-31), the `SIGSign` function (page 6-34), the `SIGVerify` function (page 6-40), or the `SIGVerifyFile` function (page 6-41).

Call the `SIGGetSignerInfo` function (page 6-48) to determine the total number of certificates in the certificate set.

The certificate information structure is described on page 6-25.

You can use the SIGGetCertNameAttributes function (described next) to obtain the contents of each attribute in the distinguished name of the signer of the certificate.

You can use the SIGGetCertIssuerNameAttributes function (page 6-52) to obtain the contents of each attribute in the distinguished name of the issuer of the certificate.

The attributes that compose a distinguished name are shown in Table 6-1 on page 6-9.

SIGGetCertNameAttributes

The SIGGetCertNameAttributes function returns information about a specific attribute of a distinguished name in a specific certificate of a signature.

```
pascal OSErr SIGGetCertNameAttributes (SIGContextPtr context,
                                unsigned long certIndex,
                                unsigned long attributeIndex,
                                SIGNameAttributesInfo *attributeInfo);
```

context
: The context pointer you used the last time you called the SIGVerify, SIGVerifyFile, SIGSignPrepare, or SIGSign function.

certIndex
: The index number of the certificate about which you want information. The certificates are always in order: the signer's certicate has index number 0, the issuer of the signer's certificate has index number 1, and so forth. You can use the SIGGetSignerInfo function to determine the total number of certificates in the certificate set.

attributeIndex
: The index number of the name attribute about which you want information. The SIGGetCertInfo function returns the total number of attributes in a certificate.

attributeInfo
: A pointer to a SIGNameAttributesInfo structure.

DESCRIPTION

After you use the SIGGetCertInfo function to determine the total number of attributes in the distinguished name of a certificate, you can use the SIGGetCertNameAttributes function to obtain the attribute strings.

The SIGNameAttributesInfo structure returns information about the hierarchical level of the attribute, the type of name attribute, and the script code of the attribute, as well as returning the attribute string. You can use the hierarchy information to arrange the distinguished name for display to the user.

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0008	$0778

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`kSIGCertificateQueryDenied`	–1971	Can't query certificates with this context
`kSIGIndexErr`	–1974	Index value is outside allowable range
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGConversionErr`	–1981	Unable to convert an attribute to Macintosh format

SEE ALSO

You can call the `SIGGetNameAttributes` function after you call the `SIGSignPrepare` function (page 6-31), the `SIGSign` function (page 6-34), the `SIGVerify` function (page 6-40), or the `SIGVerifyFile` function (page 6-41).

Call the `SIGGetCertInfo` function (page 6-49) to determine the total number of attributes in the distinguished name. You can use the `SIGGetSignerInfo` function (page 6-48) to determine the total number of certificates in the certificate set.

The `SIGNameAttributesInfo` structure is described on page 6-26.

You can use the `SIGGetCertIssuerNameAttributes` function (described next) to obtain the contents of each attribute in the distinguished name of the issuer of the certificate.

Distinguished names and name hierarchies are described in detail in "About Public-Key Certificates" beginning on page 6-8.

SIGGetCertIssuerNameAttributes

The `SIGGetCertIssuerNameAttributes` function returns information about a specific attribute of the distinguished name of the issuer of a specific certificate of a signature.

```
pascal OSErr SIGGetCertIssuerNameAttributes
                              (SIGContextPtr context,
                              unsigned long certIndex,
                              unsigned long attributeIndex,
                              SIGNameAttributesInfo *attributeInfo);
```

`context` The context pointer you used the last time you called the `SIGVerify`, `SIGVerifyFile`, `SIGSignPrepare`, or `SIGSign` function.

`certIndex` The index number of the certificate about for whose issuer you want information. The certificates are always in order: the signer's certicate has index number 0, the issuer of the signer's certificate has index number 1, and so forth. You can use the `SIGGetSignerInfo` function to determine the total number of certificates in the certificate set.

`attributeIndex`
The index number of the name attribute about which you want information. The `SIGGetCertInfo` function returns the total number of attributes in the issuer of a certificate.

`attributeInfo`
A pointer to a `SIGNameAttributesInfo` structure.

DESCRIPTION

After you use the `SIGGetCertInfo` function to determine the total number of attributes in the distinguished name of the issuer of a certificate, you can use the `SIGGetCertIssuerNameAttributes` function to obtain the attribute strings.

The `SIGNameAttributesInfo` structure returns information about the hierarchical level of the attribute, the type of name attribute, and the script code of the attribute, as well as returning the attribute string. You can use the hierarchy information to arrange the distinguished name for display to the user.

This function is useful if you want information about the issuer of a certificate. If you are using the `SIGCertInfo` and `SIGCertNameAttributes` functions to obtain information about all the certificates in a certificate set, then you must use the `SIGGetCertIssuerNameAttributes` function to determine the prime issuer.

SPECIAL CONSIDERATIONS

This function may move or purge memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Parameter count	Routine selector
$0008	$0779

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Illegal parameter value
`kSIGCertificateQueryDenied`	–1971	Can't query certificates with this context
`kSIGIndexErr`	–1974	Index value is outside allowable range
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGConversionErr`	–1981	Unable to convert an attribute to Macintosh format

SEE ALSO

You can call the `SIGGetNameAttributes` function after you call the `SIGSignPrepare` function (page 6-31), the `SIGSign` function (page 6-34), the `SIGVerify` function (page 6-40), or the `SIGVerifyFile` function (page 6-41).

Call the `SIGGetCertInfo` function (page 6-49) to determine the total number of attributes in the distinguished name. You can use the `SIGGetSignerInfo` function (page 6-48) to determine the total number of certificates in the certificate set.

The `SIGNameAttributesInfo` structure is described on page 6-26.

You can use the `SIGGetCertNameAttributes` function (page 6-51) to obtain the contents of each attribute in the distinguished name of the signer of the certificate.

Distinguished names and name hierarchies are described in detail in "About Public-Key Certificates" beginning on page 6-8.

Application-Defined Function

The `SIGSign`, `SIGSignFile`, `SIGVerifyPrepare`, and `SIGVerifyFile` functions all take a `statusProc` parameter, which is a pointer to a callback routine. You may provide this routine to notify the user of the progress of the signing or verification process. Your routine may perform typical "busy-notification" actions, such as spinning the cursor, or it may offer the user the opportunity to cancel the operation.

MyStatusCallBack

Your status callback function can perform background tasks during the signing and verification processes.

```
pascal Boolean MyStatusCallBack (void);
```

DESCRIPTION

To provide a status callback function, pass a pointer (of type `SIGStatusProcPtr`) to your function in the `statusProc` parameter of the `SIGSign`, `SIGSignFile`, `SIGVerifyPrepare`, and `SIGVerifyFile` functions. If you return `false` as your function result, the Digital Signature Manager halts the signing or verifying operation.

This interface is available because the signing and verifying operations can take a relatively long time to complete. Your status callback function should provide some sort of feedback to the user, such as a spinning cursor or a dialog box, that indicates that the process is proceeding. This function should poll for Command-period keystrokes and return `false` if it detects one. Your status callback function can also perform any other background tasks you wish.

In addition to this routine, you may wish to have other progress-notification routines that are not callback routines. For example, you may wish to have a routine that keeps the user posted of progress between calls to `SIGProcessData`.

Note

It is impossible to determine ahead of time how many times your callback routine will be executed. ◆

SPECIAL CONSIDERATIONS

If you return `false` to halt the signing or verifying operation, the state of the context is undefined.

On entry, this routine restores the A5 register to the value it had when the routine was first called.

SEE ALSO

The `SIGSign` function is described on page 6-34.

The `SIGSignFile` function is described on page 6-36.

The `SIGVerifyPrepare` function is described on page 6-38.

The `SIGVerifyFile` function is described on page 6-41.

The `SIGProcessData` function is described on page 6-30.

Summary of the Digital Signature Manager

C Summary

Constants and Data Types

```
#define kSIGDigestSize           16

#define kSIGSignerCertIndex      0

#define kSIGSignatureIconResID          –16797

#define kSIGValidSignatureIconResID     –16799

#define kSIGInvalidSignatureIconResID   –16798

/* Name attribute types returned from SIGGetCertNameAttributes or
SIGGetCertIssuerNameAttributes */

typedef enum
{
   kSIGCountryCode,
   kSIGOrganization,
   kSIGStreetAddress,
   kSIGState,
   kSIGLocality,
   kSIGCommonName,
   kSIGTitle,
   kSIGOrganizationUnit,
   kSIGPostalCode
} ;

typedef unsigned short SIGNameAttributeType;

/* Signature status codes returned in SIGCertInfo or SIGSignerInfo */

typedef enum {
   kSIGValid,         /* all valid */
   kSIGPending,       /* none expired; some pending or unknown */
   kSIGExpired,       /* some expired, unknown, or pending */
   kSIGInvalid       /* some invalid, pending, expired, or unknown */
};

typedef unsigned short SIGCertStatus;
typedef unsigned short SIGSignatureStatus;

#define gestaltDigitalSignatureVersion 'dsig'

typedef Byte SIGDigestData[kSIGDigestSize], *SIGDigestDataPtr;

struct SIGSignerInfo
{
   unsigned long        signingTime;      /* time of signing */
   unsigned long        certCount;        /* number of certs in cert set */
   unsigned long        certSetStatusTime;/* expiration time */
   SIGSignatureStatus   signatureStatus;  /* certificate status */
};
typedef struct SIGSignerInfo SIGSignerInfo;
typedef SIGSignerInfo *SIGSignerInfoPtr;

struct SIGCertInfo
{
   unsigned long  startDate;           /* cert start validity date */
   unsigned long  endDate;             /* cert end validity date */
   SIGCertStatus  certStatus;          /* certificate status*/
   unsigned long  certAttributeCount;  /* number of name attributes in cert*/
   unsigned long  issuerAttributeCount;/* number of name attributes in
                                          cert’s issuer */
   Str255         serialNumber;        /* cert serial number */
};
typedef struct SIGCertInfo SIGCertInfo;
typedef SIGCertInfo *SIGCertInfoPtr;

typedef Ptr SIGContextPtr;
typedef Ptr SIGSignaturePtr;

struct SIGNameAttributesInfo
{
   Boolean                 onNewLevel;
   SIGNameAttributeType    attributeType;
   ScriptCode              attributeScript;
   Str255                  attribute;
};
```

```
typedef struct SIGNameAttributesInfo SIGNameAttributesInfo;
typedef SIGNameAttributesInfo *SIGNameAttributesInfoPtr;
```

Digital Signature Manager Functions

Creating and Disposing of a Context

```
pascal OSErr SIGNewContext  (SIGContextPtr *context);
pascal OSErr SIGDisposeContext
                            (SIGContextPtr context));
```

Processing Data to Generate a Digest

```
pascal OSErr SIGProcessData
                            (SIGContextPtr context,
                             const void *data,
                             Size dataSize);
```

Creating a Signature

```
pascal OSErr SIGSignPrepare
                            (SIGContextPtr context,
                             const FSSpec *signerFile,
                             ConstStr255Param prompt,
                             Size *signatureSize);
pascal OSErr SIGSign        (SIGContextPtr context,
                             SIGSignaturePtr signature,
                             SIGStatusProcPtr statusProc);
pascal OSErr SIGSignFile    (SIGContextPtr context,
                             Size signatureSize,
                             const FSSpec *fileSpec,
                             SIGStatusProcPtr statusProc)
```

Verifying a Signature

```
pascal OSErr SIGVerifyPrepare
                            (SIGContextPtr context,
                             SIGSignaturePtr signature,
                             Size signatureSize,
                             SIGStatusProcPtr statusProc);
pascal OSErr SIGVerify      (SIGContextPtr context);
pascal OSErr SIGVerifyFile  (SIGContextPtr context,
                             const FSSpec *fileSpec,
                             SIGStatusProcPtr statusProc)
```

Creating a Digest

```
pascal OSErr SIGDigestPrepare
                            (SIGContextPtr context);
pascal OSErr SIGDigest      (SIGContextPtr context,
                             SIGDigestData digest);
```

Getting Information From a Signature or Certificate

```
pascal OSErr SIGFileIsSigned
                            (const FSSpec *fileSpec);
pascal OSErr SIGShowSigner  (SIGContextPtr context,
                             ConstStr255Param prompt);
pascal OSErr SIGGetSignerInfo
                            (SIGContextPtr context,
                             SIGSignerInfo *signerInfo);
pascal OSErr SIGGetCertInfo (SIGContextPtr context,
                             unsigned long certIndex,
                             SIGCertInfo *certInfo);
pascal OSErr SIGGetCertNameAttributes
                            (SIGContextPtr context,
                             unsigned long certIndex,
                             unsigned long attributeIndex,
                             SIGNameAttributesInfo *attributeInfo);
pascal OSErr SIGGetCertIssuerNameAttributes
                            (SIGContextPtr context,
                             unsigned long certIndex,
                             unsigned long attributeIndex,
                             SIGNameAttributesInfo *attributeInfo);
```

Application-Defined Function

```
pascal Boolean MyStatusCallBack
                            (void);
```

Pascal Summary

Constants and Data Types

```
CONST

{ Number of bytes needed for a digest record when using SIGDigest }

kSIGDigestSize                  = 16;

kSIGSignerCertIndex             = 0;

kSIGSignatureIconResID          = –16197

kSIGValidSignatureIconResID     = –16799

kSIGInvalidSignatureIconResID   = –16798

{ values of SIGNameAttributeType }

   kSIGCountryCode      = 0;
   kSIGOrganization     = 1;
   kSIGStreetAddress    = 2;
   kSIGState            = 3;
   kSIGLocality         = 4;
   kSIGCommonName       = 5;
   kSIGTitle            = 6;
   kSIGOrganizationUnit = 7;
   kSIGPostalCode       = 8;

{ values for SIGCertStatus or SIGSignatureStatus }

   kSIGValid      = 0;     { possible for either a SIGCertStatus or
                                 SIGSignatureStatus }
   kSIGPending    = 1;     { possible for either a SIGCertStatus or
                              SIGSignatureStatus }
   kSIGExpired    = 2;     { possible for either a SIGCertStatus or
                              SIGSignatureStatus }
   kSIGInvalid    = 3;     { possible only for a SIGSignatureStatus }

{ Gestalt selector code - returns toolbox version in low-order word }

   gestaltDigitalSignatureVersion = 'dsig';
```

```
TYPE

SIGNameAttributeType = INTEGER;

SIGCertStatus = INTEGER;

SIGSignatureStatus = INTEGER;

SIGDigestData = PACKED ARRAY[1..kSIGDigestSize] OF Byte;
SIGDigestDataPtr = ^SIGDigestData;

SIGSignerInfo = RECORD

   signingTime:      LONGINT;          { time of signing }
   certCount:        LONGINT;          { number of certificates in cert set }
   certSetStatusTime:LONGINT;          { expiration time }
   signatureStatus:  SIGSignatureStatus;{ status of the certificate }
END;

SIGSignerInfoPtr = ^SIGSignerInfo;

SIGCertInfo = RECORD
   startDate: LONGINT;           { cert start validity date }
   endDate: LONGINT;             { cert end validity date }
   certStatus: SIGCertStatus;    { signature status}
   certAttributeCount: LONGINT;  { number of name attributes in this cert }
   issuerAttributeCount: LONGINT;{ # of name attributes in certs issuer }
   serialNumber: Str255;         { cert serial number }
END;

SIGCertInfoPtr = ^SIGCertInfo;

SIGContextPtr     = Ptr;

SIGSignaturePtr   = Ptr;

SIGStatusProcPtr  = ProcPtr;  { FUNCTION SIGStatusProcPtr(): BOOLEAN;}

SIGNameAttributesInfo = RECORD
   onNewLevel: BOOLEAN;
   attributeType: SIGNameAttributeType;
   attributeScript: ScriptCode;
   attribute: Str255;
END;

SIGNameAttributesInfoPtr = ^SIGNameAttributesInfo;
```

Digital Signature Manager Functions

Creating and Disposing of a Context

```
FUNCTION SIGNewContext      (VAR context: SIGContextPtr): OSErr;
FUNCTION SIGDisposeContext  (context: SIGContextPtr): OSErr;
```

Processing Data to Generate a Digest

```
FUNCTION SIGProcessData     (context: SIGContextPtr; data: UNIV Ptr;
                             dataSize: Size): OSErr;
```

Creating a Signature

```
FUNCTION SIGSignPrepare     (context: SIGContextPtr; signerFile: FSSpecPtr;
                             prompt: StringPtr; VAR signatureSize: Size):
                             OSErr;
FUNCTION SIGSign            (context: SIGContextPtr; signature:
                             SIGSignaturePtr;statusProc: SIGStatusProcPtr):
                             OSErr;
FUNCTION SIGSignFile        (context: SIGContextPtr; signatureSize: Size;
                             fileSpec: FSSpec;statusProc:
                             SIGStatusProcPtr): OSErr;
```

Verifying a Signature

```
FUNCTION SIGVerifyPrepare   (context: SIGContextPtr; signature:
                             SIGSignaturePtr; signatureSize: Size;
                             statusProc: SIGStatusProcPtr): OSErr;
FUNCTION SIGVerify          (context: SIGContextPtr): OSErr;
FUNCTION SIGVerifyFile      (context: SIGContextPtr; fileSpec: FSSpec;
                             statusProc: SIGStatusProcPtr): OSErr;
```

Creating a Digest

```
FUNCTION SIGDigestPrepare   (context: SIGContextPtr): OSErr;
FUNCTION SIGDigest          (context: SIGContextPtr; digest:
                             SIGDigestData): OSErr;
```

Getting Information From a Signature or Certificate

```
FUNCTION SIGFileIsSigned    (fileSpec: FSSpec): OSErr;
FUNCTION SIGShowSigner      (context: SIGContextPtr; prompt: StringPtr):
                             OSErr;
FUNCTION SIGGetSignerInfo   (context: SIGContextPtr;
                             VAR signerInfo: SIGSignerInfo): OSErr;
```

```
FUNCTION SIGGetCertInfo      (context: SIGContextPtr; certIndex: LONGINT;
                              VAR certInfo: SIGCertInfo): OSErr;
FUNCTION SIGGetCertNameAttributes
                             (context: SIGContextPtr; certIndex: LONGINT;
                              attributeIndex: LONGINT; VAR attributeInfo:
                              SIGNameAttributesInfo): OSErr;
FUNCTION SIGGetCertIssuerNameAttributes
                             (context: SIGContextPtr; certIndex: LONGINT;
                              attributeIndex: LONGINT; VAR attributeInfo:
                              SIGNameAttributesInfo): OSErr;
```

Application-Defined Function

```
FUNCTION MyStatusCallBack (): BOOLEAN;
```

Assembly-Language Summary

Trap Macros Requiring Routine Selectors

$AA5D

Selector	Count	Routine
$076C	2	SIGNewContext
$076D	2	SIGDisposeContext
$076E	8	SIGSignPrepare
$076F	6	SIGSign
$0770	8	SIGVerifyPrepare
$0771	2	SIGVerify
$0772	2	SIGDigestPrepare
$0773	4	SigDigest
$0774	6	SIGProcessData
$0775	4	SIGShowSigner
$0776	4	SIGGetSignerInfo
$0777	6	SIGGetCertInfo
$0778	8	SIGGetCertNameAttributes
$0779	8	SIGGetCertIssuerNameAttributes
$09C4	2	SIGFileIsSigned
$09C5	8	SIGSignFile
$09C6	6	SIGVerifyFile

Result Codes

In addition to standard Macintosh Operating System errors such as `memFullErr` and `paramErr`, the Digital Signature Manager returns the result codes listed in this section.

Result codes in the range of –1970 to –1999 are reserved for the Digital Signature Manager.

`kSIGOperationIncompatibleErr`	–1970	Context in use for different type of operation
`kSIGCertificateQueryDenied`	–1971	Can’t query certificates with this context
`kSIGVerifyFailedErr`	–1972	Verification failed
`kSIGInvalidCredentialErr`	–1973	Verified OK but credential either pending or expired
`kSIGIndexErr`	–1974	Index given is outside the range of allowable values
`kSIGSignerErr`	–1975	Problem with the signer file or signature
`kSIGPasswordErr`	–1976	Password is incorrect
`kSIGInternalsErr`	–1977	Bad digest, context, or signature
`kSIGToolboxNotPresentErr`	–1978	For servers; not returned by the toolbox
`kSIGContextPrepareErr`	–1979	Context either corrupted or already prepared with `SIGVerifyPrepare`, `SIGSignPrepare`, or `SIGDigestPrepare`
`kSIGNoDigestErr`	–1980	No digest in the signature
`kSIGConversionErr`	–1981	Unable to convert an attribute to Macintosh format
`kSIGSignerNotValidErr`	–1982	Signer file has either expired or is not yet valid
`kSIGNoSignature`	–1983	Standard file signature not found