Boot Integrity Services Application Programming Interface

Version 1.0

December 28, 1998
Intel Corporation

Version 1.0 December 28, 1998
Copyright © 1998, Intel Corporation. All rights reserved.

This document is for informational purposes only. **INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.**

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you any license to the patents, trademarks, copyrights, or other intellectual property rights except as expressly provided in any written license agreement from Intel Corporation.

Intel does not make any representation or warranty regarding specifications in this document or any product or item developed based on these specifications. INTEL DISCLAIMS ALL EXPRESS AND IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND FREEDOM FROM INFRINGEMENT. Without limiting the generality of the foregoing, Intel does not make any warranty of any kind that any item developed based on these specifications, or any portion of a specification, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country. It is your responsibility to seek licenses for such intellectual property rights where appropriate. Intel shall not be liable for any damages arising out of or in connection with the use of these specifications, including liability for lost profit, business interruption, or any other damages whatsoever. Some states do not allow the exclusion or limitation of liability or consequential or incidental damages; the above limitation may not apply to you.

† Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owners' benefit, without intent to infringe.

Copyright © 1997, 1998, Intel Corporation. All rights reserved.

Copyright © 1998, Intel Corporation. All rights reserved.

Table of Contents

1. INTRODUCTION..1

1.1 Overview...1
1.2 Audience 1
1.3 Relationship to other Standards and Specifications 2
1.4 References 2
1.4.1 Documentation conventions...3
1.5 Terms and Definitions...3

2. TYPICAL USAGE SCENARIOS...5

2.1 Remote-Boot Authorization Certificates..5
2.2 Configuration of Remote-Boot Authentication 7
2.2.1 Continuous-Security first-time setup example 9
2.2.2 Unattended first-time setup example 11
2.3 Remote-boot authentication 12
2.4 Initialization and Shutdown 14

3. INTERFACE DESCRIPTION 15

3.1 BIS Entry Point Structure..15
3.1.1 pBisEntry16, pBisEntry32..15
3.1.2 BIS Operation Codes 16
3.1.3 BIS_ENTRY_POINT..17
3.1.4 BIS_ENTRY_POINT_TYPE..17
3.2 BIS Calling Convention 17
3.2.1 16-bit C example 19
3.2.2 16-bit Assembly example 19
3.2.3 32-bit C example 20
3.2.4 32-bit Assembly example 20
3.3 Additional BIS caller requirements...21
3.4 Signature Algorithms and Key Lengths Supported..22
3.5 Digital Certificates...22
3.6 Signed Manifests...23
3.7 Request Credentials 24
3.8 Data structures 24
3.8.1 COMPILER_IS_16_BIT and COMPILER_IS_32_BIT...25
3.8.2 BIS_GEN_SCS_CONST 25
3.8.3 UINT8, UINT16, UINT32, UINT64, INT8, INT16, INT32..25
3.8.4 BIS_APPLICATION_HANDLE..26
3.8.5 BIS_ALG_ID...26
3.8.6 Predefined BIS_ALG_ID values 26
3.8.7 BIS_BOOLEAN..26
3.8.8 Predefined BIS_BOOLEAN values 27
3.8.9 BIS_BYTE_PTR...27
3.8.10 BIS_STATUS...27
3.8.11 Predefined BIS_STATUS values...28
3.8.12 BIS_NULL 28

Copyright © 1998, Intel Corporation. All rights reserved.

3.8.13 BIS_DATA *28*
3.8.14 BIS_VERSION *29*
3.8.15 BIS_CURRENT_VERSION_MAJOR, BIS_VERSION_1 ...... *29*
3.8.16 BIS_SIGNATURE_INFO *30*
3.8.17 BIS_GET_SIGINFO_COUNT *30*
3.8.18 BIS_GET_SIGINFO_ARRAY *31*
3.9 Boot Object Authentication functions **31**
3.9.1 BIS_GetBootObjectAuthorizationCertificate *32*
3.9.2 BIS_VerifyBootObject *34*
3.9.3 BIS_GetBootObjectAuthorizationCheckFlag *39*
3.9.4 BIS_GetBootObjectAuthorizationUpdateToken *40*
3.9.5 BIS_UpdateBootObjectAuthorization *42*
3.10 General-purpose security functions **47**
3.10.1 BIS_VerifyObjectWithCredential *48*
3.11 Initialization, Shutdown, and Utility functions **52**
3.11.1 BIS_Initialize *53*
3.11.2 BIS_Shutdown *55*
3.11.3 BIS_Free *57*
3.11.4 BIS_GetSignatureInfo *59*

Copyright © 1998, Intel Corporation. All rights reserved.

1. Introduction

1.1 Overview

The software interface described in this document is designed to cover basic security needs during part of the "boot" phase of system startup. In particular, the time span covered starts at the end of Basic I/O System (BIOS) initialization and ends when control is transferred to a high-level Operating System (OS) such as Windows†. Specific security needs that these functions cover are:

- An OS boot image obtained over a network needs to be checked for integrity and authority to run on a particular platform. One group of functions in this software interface addresses this integrity and authority check. This group embodies a boot image security usage model that is simple to adopt, yet flexible enough to cover a wide variety of needs.
- In some cases the pre-supplied boot image security usage model embodied by the first group of functions is not sufficient to cover a specific need, and a more sophisticated security usage model is required. Another group of functions allows software developers to implement their own custom security usage model that meets their specific needs.

The implementation of this software interface is subject to severe size constraints since the implementation generally cannot be loaded from mass-storage media. Consequently, the functionality supplied is kept to the bare minimum required to check the "next stage" of software loaded. Applications requiring a more sophisticated security software infrastructure may "bootstrap" themselves by loading their own security software support modules as a first stage.

1.2 Audience

This document is expected to meet the needs of two types of readers:

- Developers of software that will run in the preboot phase of system startup. This includes BIOS developers, developers of Network Interface Card firmware, Preboot Execution Environment implementers, developers of OS bootstraps, and preboot platform management applet developers. These readers need to know the services that will be available and the details of the software interfaces that invoke those services.
- Manufacturers of platforms that conform to the Wired for Management 2.0 specification. This specification defines a part of the Wired for Management requirements that must be met by the platform.

Copyright © 1998, Intel Corporation. All rights reserved.

It is assumed that the audience fully understands the security requirements for their products and has a basic understanding of the security mechanisms and services that could be used to meet those requirements.

1.3 Relationship to other Standards and Specifications

This document is a companion to several other specifications:

- The Wired for Management Baseline 2.0 specification document. This document describes the capability and interface requirements a conforming platform must meet.
- The Preboot Execution Environment (PXE) Specification, version 2.0.
- A TBD Software Development Kit (SDK) Specification that describes tools used to develop software that uses the interfaces and functionality described in this document.

In addition, this specification uses other industry specifications or standards for certificates, keys, signatures, and cryptographic algorithms. Standards used include:

- X.509v3 certificates as identity credentials.
- Signed Manifest Specification [SM Spec] as integrity credentials.
- Base64 encoding specification [BASE64] for encoding of binary data in signed manifests.
- PKCS#1 [PKCS] signatures.
- OIW algorithm identifiers [OIW] and parameters to encode the signature algorithms in certificates and signature blocks.

1.4 References

ASN.1 — *ITU (formerly CCITT) Abstract Syntax Notation X200*

BER — *ITU Basic Encoding Rules*

DER — *ITU Distinguished Encoding Rules*

OIW — *Stable Implementation Agreements,* Open Systems Environment Implementers Workshop, June 1995

PKCS — *The Public-Key Cryptography Standards*, RSA Laboratories, Redwood City, CA: RSA Data Security, Inc.

SM Spec — Signed Manifest Specification, The Open Group, 1997. http://www.opengroup.org/pubs/catalog/c707.htm

BASE64 — *RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. Section 5.2: Base64 Content-Transfer-Encoding. ftp://ftp.isi.edu/in-notes/rfc1521.txt

Copyright © 1998, Intel Corporation. All rights reserved.

X.509	*ITU. Recommendation X.509: The Directory – Authentication Framework*. 1988 ITU stands for the International Telecommunications Union (formerly CCITT). CCITT stands for Comite Consultatif Internationale Telegraphique et Telephonique (International Telegraph and Telephone Consultative Committee)
PXE Spec	*Preboot Execution Environment (PXE) Specification Version 2.0 December 23, 1998.*
SMBIOS Spec	*System Management BIOS Reference Specification Version 2.3 — 12 August 1998*. SMBIOS techniques are used to bind to and call functions defined in this specification. ftp://download.intel.com/ial/wfm/smbios.pdf
ACPI	*Advanced Configuration and Power Interface Specification*, Version 1.0, December 23 1996, http://www.teleport.com/~acpi

1.4.1 Documentation conventions

The document uses a [TAG] reference to refer back to the first time a particular specification is mentioned in the document.

1.5 Terms and Definitions

The following terms are briefly defined to compensate for the fact that the industry literature differs in use of these terms from one document to another. This is not intended to be a complete security primer.

Boot Integrity Services (BIS)	Refers in general to the overall Boot Integrity Service, including Application Programming Interface, implementation, stored parameters, etc.
platform	A single computer. In the Personal Computer (PC) arena this is a single PC. A multi-processor PC is a single platform. A server is a single platform. As computer systems move further and further from these examples, it becomes more and more the responsibility of the system designer to define the boundaries between one platform and another.
persistent	Refers to a class of storage or the data items stored there. Persistent data survives across platform reboots. Examples of persistent storage are FLASH memory, battery-backed CMOS memory, or disk.

Copyright © 1998, Intel Corporation. All rights reserved.

protected persistent	The degree to which the persistent storage used by a BIS implementation for BIS security parameters is protected against overwriting by erroneous or unauthorized software directly affects the degree of security provided by that implementation. This specification uses the term "protected" to indicate the situations in which this consideration applies. However, this specification does not define degrees of protection, nor require any particular kind or level of protection for such storage, leaving the resolution of these considerations completely in the realm of product definition.
(cryptographic) key	A digital quantity that can be used as an authentication parameter in various algorithms. The uses of cryptographic keys in this document are generally for producing or verifying digital signatures. This document usually refers to a "key" for brevity.
key pair	Some types of keys come in corresponding pairs. The "private" member of a pair is held by the key pair's owner and used to generate digital signatures. The "public" member of the pair is widely published and used to verify digital signatures. A public key is usually published in the form of a digital certificate.
(digital) signature	A digital quantity computed using a message and a (usually private) key as input parameters to a digital signature algorithm. Digital signature algorithms are designed to make it difficult to either construct a signature without knowing the private key, or construct a different message with the same signature. The signature can be verified using the message and the public key.
(digital) certificate	A digital "identity" document. The document essentially makes the claim that "the person holding a private key corresponding to this public key is so-and-so". A digital certificate is a message in a standardized format containing (at least) a "subject" public key and identifier, an "issuer" public key and identifier, and a digital signature created using the issuer's private key. When this document refers to "the public key from a certificate", it means the "subject" public key, that is, the public key of the identified party.

Copyright © 1998, Intel Corporation. All rights reserved.

2. Typical Usage Scenarios

A hammer is easiest to describe if you can describe how it is used to pound nails, even though nails are not part of the hammer. Similarly, the software interfaces defined in this document are easiest to describe and understand in the context in which they are used, even though the context is not an inherent part of the software interface. This section gives several brief in-context usage examples to describe the software interfaces. Each example concludes with a summary indicating which parts of the example are actually provided by the interfaces and which parts were just a context for the description. The following examples are given:

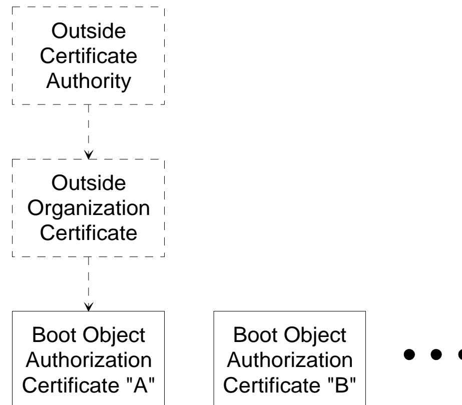
- Remote-Boot Authorization Certificates. Illustrates the typical way an Information Technology (IT) organization sets up digital certificates for signing boot images. A remote-boot authorization certificate is set up for each group of platforms to be managed. The authorization certificate for a group of platforms is configured into each of the platforms in the group as the source of “authority” for allowing boot images to be executed. IT organizations may create different authorization certificates for different groups to allow them to be managed by different authorities.
- Configuration of Remote-Boot Authentication. Illustrates the typical process by which a platform is configured to recognize a given IT organization as the authority for allowing boot images to be executed.
- Remote-Boot Authentication. Illustrates the typical context in which these software interfaces are called to perform the Remote-Boot authentication check.
- Initialization and Shutdown. Illustrates the general sequence rules that must be followed when calling the functions in these interfaces.

2.1 Remote-Boot Authorization Certificates

This usage scenario is based on the following requirements:

- The platform must be protected against remote-boot images that have been damaged or tampered with either in transit or on the server from which they are downloaded.
- IT organizations require that they be able to designate and enforce which boot images are permitted to run on platforms they manage.
- IT organizations need a mechanism to limit the scope of management authorities within their organization to ease transitions of authority when organizational changes demand it.
- These requirements are addressed by letting an IT organization set up a collection of remote-boot authorization certificates as illustrated in the following figure:

Copyright © 1998, Intel Corporation. All rights reserved.



A collection of remote-boot authorization certificates is shown. An IT organization sets up multiple Boot Object Authorization Certificates (solid boxes) as the source of Remote-Boot authorization. Certificates may be members of an outside certificate hierarchy as with Certificate "A" at the left. The IT organization may also set up standalone (self-certified) Certificates as with Certificate "B" at the right. The interfaces defined in this document do not make any restrictions about the presence or absence of an outside certificate hierarchy.

Figure 1: Collection of Remote-Boot Authorization Certificates

An IT organization managing a group of platforms configures each of the platforms to recognize the IT organization's Boot Object Authorization Certificate as the source of authority for signing Remote-Boot objects. The IT organization evaluates potential Remote-Boot objects. When a Remote-Boot object is found to be suitable according to IT's criteria, the IT organization uses its certificate to create a signed manifest (as described in section 3.6) authorizing that object to be used as a Remote-Boot object on that set of managed platforms. In particular, the signing organization uses the private key corresponding to the public key in that organization's certificate to sign the manifest. Signing organizations do not expose their private keys outside the IT organization.

The interfaces described in this document do not provide any explicit mechanism for delegating remote-boot object manifest signing authority to different sub-groups. An IT organization that wishes to delegate the task of evaluating and approving remote-boot objects to IT sub-groups needs to set up an outside process for managing and distributing approved remote-boot objects. An IT sub-group managing a group of platforms takes IT-approved remote-boot objects and creates signed manifests for them using the sub-group's private key. The sub-group's private key corresponds to the Boot Object Authorization Certificate configured into each of the platforms managed by the sub-group. This allows the remote-boot objects to run on those managed platforms.

An IT organization can also delegate remote-boot object manifest signing authority to IT sub-groups by sharing a central private key used for signing remote-boot objects among IT sub-groups. This approach can be taken in cases where the increased risk of compromising the private key is acceptable and when it is acceptable for a central key to have wider management scope.

To check the authenticity and authorization of a boot object, each managed platform must be configured with a Boot Object Authorization Certificate. There is only one such certificate configured in each platform. The Boot Object Authorization Certificate designates the source of authorized boot-object manifest signatures for that platform.

Copyright © 1998, Intel Corporation. All rights reserved.

When a managed platform downloads a Remote-Boot object, it also downloads the corresponding signed manifest. The platform checks the object against the signed manifest to check the integrity of the downloaded Remote-Boot object. The platform checks the certificate of the manifest's signer to make sure the manifest was signed by the private key corresponding to the platform's Boot Object Authorization Certificate.

In a large corporation, an IT organization may wish to create several Boot Object Authorization Certificates for different groups of managed platforms. This may be done for several reasons, including:

- It provides a simple mechanism to authorize different sets of Remote-Boot objects for different groups of platforms. For example, platforms used by a software-evaluation group might allow some experimental Boot Objects to run, while platforms used for office-automation tasks might not.
- It lets the IT organization limit the scope of effect when organizational or other changes shift the management authority for a group of platforms.

Only two parts of this usage model are actually embodied in the interfaces described in this document:

- Configuration of a Boot Object Authorization Certificate in a managed platform, and
- Performing an integrity and authorization check of an object using a signed manifest and the configured Boot Object Authorization Certificate.

It is the responsibility of other parts of software inside and outside of the managed platform to perform the remaining parts of the usage model.

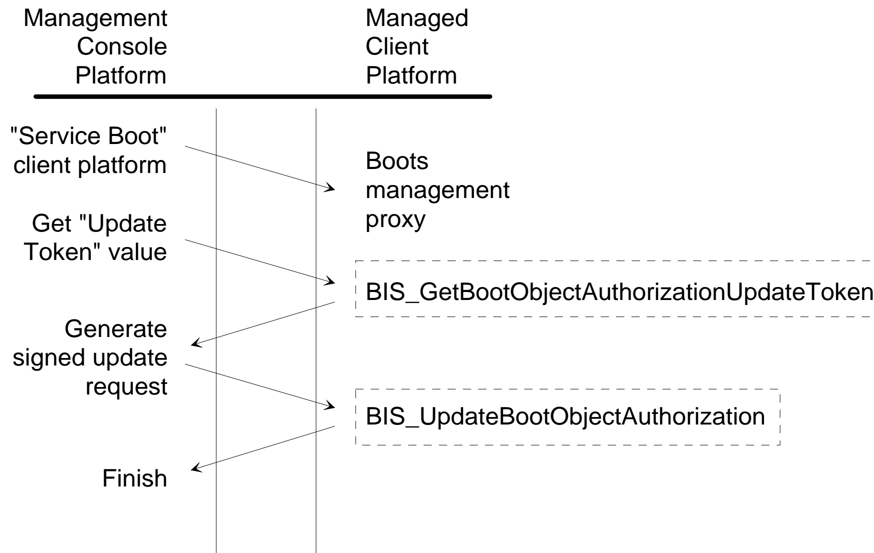
2.2 Configuration of Remote-Boot Authentication

The Remote-Boot Authentication support described in this document has two configurable parameters on each managed platform. Current values for these parameters are made persistent and protected across system reboots and power interruptions. The parameters are:

- The Boot Object Authorization Certificate that identifies the organization that shall be recognized as the source of Boot Object authority for this platform, and
- A flag, referred to as the Boot Authorization Check Flag, that indicates whether or not the Boot Object Verification function defined in this interface shall require authorization. The authorization, when required, requires a digital signature generated with the private key held by the Boot Object authority.

At the interface level, updating either of these configuration parameters uses two BIS functions, BIS_UpdateBootObjectAuthorization and BIS_GetBootObjectAuthorizationUpdateToken. The description below gives a high-level view of a typical configuration sequence for a managed client platform. The description shows the context in which these functions are used. The typical configuration operation involves interactions between a Management Console platform and a managed client platform as summarized in the following figure:

Copyright © 1998, Intel Corporation. All rights reserved.



A typical interaction between a Management Console (left of the vertical dividers) and BIS functions (dashed boxes) on a Managed Client (right) is shown. The Console causes the Managed Client to boot a small proxy application that performs operations on the Managed Client on behalf of the Console. The Console requests an "Update Token" value, which is returned by the Managed Client's BIS_GetBootObjectAuthorizationUpdateToken function. The Console assembles an update "request credential" message describing the configuration changes and including the update token. The Console signs the request credential using the private key corresponding to the Managed Client's Boot Object Authorization Certificate and sends it to the client. The Managed Client verifies the signature and performs the update using the BIS_UpdateBootObjectAuthorization function and returns a confirmation. The Management Console checks the confirmation and finishes.

Figure 2: Configuration Update Interactions

The details of a Management Console application, a service boot, and a management proxy are outside the scope of BIS and this example. This example introduces several new BIS concepts:

- A unique update token. BIS associates a token value with the Boot Object Authorization parameter set. This token value is a computed value unique to the parameter set and the platform. In addition, a new unique value is generated whenever any of the parameters in the set are updated.
- A signed request credential. The digital signature in this credential covers at least the current unique update token value and the new parameter values. BIS requires the identity of the signer to match the configured identity in the Boot Object Authorization Certificate for this platform.

The unique update token changes to a new unique value whenever any of the BIS configuration parameters are changed. In addition, the unique token is unique to the specific platform it came from and the BIS configuration parameter set. These uniqueness properties are important when the configuration-update process is performed with a managed client that is not running a high-level OS. In such an environment the client platform is unlikely to support private authenticated communications. Configuration update requests may be transmitted as clear text. The unique update token and signed request credential combine to guard against attacks based on capturing and replaying an identical or altered update request to the same or different target platform.

Copyright © 1998, Intel Corporation. All rights reserved.

This configuration usage model takes advantage of the Remote-Boot Authorization Certificates described earlier. The key used to sign a configuration update request credential must be the private key corresponding to the public key in the target platform's configured Boot Object Authorization Certificate. Thus the authority to re-configure a managed platform is restricted to the holder of that private key, in other words, the IT sub-organization that "owns" the management authority for that platform. The BIS interface does not provide any mechanism to delegate that authority.

The Boot Object Authorization Certificate in a platform used to validate update request authority is itself a configurable parameter under this usage model. This has the effect that a managing authority can transfer managing authority to another organization. The usage model does not offer any other way for an unauthorized organization to acquire managing authority for a platform from its current managing authority.

The following parts of this usage model are embodied in the interfaces described and referred to in this document:

- The actual configuration parameters in this parameter set (Boot Object Authorization Certificate and Boot Authorization Check Flag).
- The unique update token associated with this parameter set.
- The function BIS_GetBootObjectAuthorizationUpdateToken for retrieving the unique update token associated with the Boot Object Authorization parameter set.
- The generalized function BIS_UpdateBootObjectAuthorization for updating either parameter in the Boot Object Authorization parameter set.
- The signed configuration update request credential (but not the software for constructing and signing such a credential).
- The Boot Object Authorization Certificate configured in a platform to designate the authorized source of update request credential signatures.

It is the responsibility of other parts of software inside and outside of the managed platform to perform the remaining parts of the usage model.

2.2.1 Continuous-Security first-time setup example

The preceding general configuration example starts out with a platform that already has a configured Boot Object Authorization Certificate. This raises the question of how Boot Object Authorization is configured for the first time on a new platform, which depends on the initial state in which a manufacture delivers a new platform. In choosing this initial state, the manufacturer is faced with two conflicting requirements that affect first-time configuration setup of a new platform:

- The desire for uninterrupted security checking of all software executed on a platform from the time it is manufactured until the time it is placed in full use by the end-user. To meet this requirement, all software should be checked to see if the customer authorizes it before it is allowed to execute.
- The desire for low cost of acquiring a new platform and placing it in service. The manufacturing cost of the system must be low to keep the purchase price low. The manufacturer would like to manufacture platforms as similar as possible to keep

Copyright © 1998, Intel Corporation. All rights reserved.

manufacturing costs low. In addition, the customer would like unattended first-time setup of the platform to keep platform management costs low.

This second requirement ends up conflicting with the first. If platforms are manufactured alike, they have no way to automatically distinguish one customer's chosen management authority from another customer's management authority. The newly manufactured platform has to either use a manufacturer's preset boot object authorization policy (conflicting with the desire for customer-controlled authorization policy) or fall back to a platform-specific action such as user interaction to authorize boot objects (conflicting with the desire for unattended first-time setup).

This example illustrates one way a platform manufacturer can set up the initial state of delivered platforms for customers that emphasize the first, "continuous security", requirement. The example in section 2.2.2 illustrates one way a platform manufacturer can set up platforms for customers that emphasize the second, "unattended setup" requirement.

To take advantage of the "continuous security" feature of BIS, a manufacturer should build platforms so that their Boot Authorization Check Flag is "on" but no Boot Object Authorization Certificate is configured. With this configuration, both the BIS_VerifyBootObject and Boot Object Authorization Certificate perform an authorization check. Ordinarily this authorization check would involve checking to make sure a signature was generated with the private key corresponding to the public key in the platform's Boot Object Authorization Certificate. Since there is no Boot Object Authorization Certificate configured, these functions "fall back" to a platform-specific action to do the authorization check. A typical platform-specific action is to interact with the user to confirm a digital signature or hash value. This in turn leads to some typical variations to the earlier configuration example:

- When the platform performs the "service boot" it attempts to verify the management proxy boot image. Since there is no Boot Object Authorization Certificate, the verification falls back on user interaction. The platform displays the digital signature (for example, in hex) and asks the operator to confirm it. The operator confirms it against the first few dozen or so digits printed on a piece of paper supplied by the IT organization that signed the proxy boot image. If the confirmation is successful, the authorization succeeds and the example continues on.
- When the management proxy calls BIS_UpdateBootObjectAuthorization, another verification occurs. There is still no Boot Object Authorization Certificate, so once again the verification falls back on user interaction. The platform displays the request-update manifest's digital signature. The operator confirms this verbally with the IT system administrator running the Management Console application that signed the request-update manifest. If the confirmation is successful, the authorization succeeds and the configuration example continues on. Usually the Boot Object Authorization Certificate would be the first thing configured on the new platform, so that subsequent configuration operations no longer require operator input.

Depending on the level of security and convenience desired, there can be minor variations of this example with different numbers of digits, requiring the user to enter the digits instead of doing a visual compare and a yes/no answer, etc.

This usage example introduces only two new features that are embodied in the interfaces described and referred to in this document:

Copyright © 1998, Intel Corporation. All rights reserved.

- A user-interaction based technique used to authenticate a boot object signature when no Boot Object Authorization Certificate is yet configured on the platform.
- A user-interaction based technique used to authenticate a configuration update request when no Boot Object Authorization Certificate is yet configured on the platform.

It is the responsibility of other parts of software inside and outside of the managed platform to perform the remaining parts of the usage model.

2.2.2 Unattended first-time setup example

The example in section 2.2.1 describes a manufacturer's initial platform configuration favoring customers that emphasize the "continuous security" requirement. This example describes a different initial platform configuration favoring customers that emphasize the "unattended first-time setup" requirement.

In this example, a manufacturer configures platforms with the Boot Authorization Check Flag turned "on" and a special "first-time setup" Boot Object Authorization Certificate configured. This special Boot Object Authorization Certificate is the same on all platforms supplied by the manufacturer. The manufacturer also supplies the "private" key corresponding to this "first-time setup" Boot Object Authorization Certificate to all customers.

An IT organization for a company that purchases these platforms signs a management proxy boot image using the well-known first-time setup key described above. When these new platforms are installed for the first time, the IT system administrator remote-boots the proxy boot image that was signed with the first-time setup key. The IT system administrator uses a Management Console application to send a configuration update request credential to the new platform. This first update request is signed using the first-time setup key. The first update request reconfigures the platform with a replacement Boot Object Authorization Certificate "owned" by the IT organization. Once this is done, the IT organization becomes the management authority for the platform and can enforce its own policies regarding which boot images may be executed and which configuration update requests can be performed. Since the platform always has a Boot Object Authorization Certificate configured, the verification operations never fall back to user interaction; the first-time setup is unattended.

From a security standpoint, this example is slightly weaker than the previous example because it includes a small window of time during which a new platform could boot software that was not approved by the customer's IT organization. This may be an acceptable cost of unattended first-time setup for many customers because of several points:

- The new platform has no user data at risk on the platform during the timing window. The only risk is the corruption of manufacturer-installed persistent data.
- In general IT organizations will have to have tools to diagnose corrupted persistent data and reset back to the manufacturer-installed state on managed client platforms.

There are other possible variations of these simplified examples. In addition there are opportunities for platform manufacturers to supply their own platform-specific features. A manufacturer should be able to find an appropriate tradeoff that meets the customer's needs.

Copyright © 1998, Intel Corporation. All rights reserved.

This usage example does not introduce any new features that are embodied in the interfaces described and referred to in this document. It is the responsibility of other parts of software inside and outside of the managed platform to perform the usage model described in this example.

2.3 Remote-boot authentication

A typical Remote-boot authentication example begins with activities outside the platform boot sequence. The IT organization that manages the platform(s) configures the Boot Object Authorization Certificate for the platform as described previously.

The IT organization evaluates a boot image and decides that it should be authorized to execute on the managed platform(s). The IT organization generates several boot image credential files for the boot image, one for each of the signature algorithm and key-length combinations mandated by this specification. The boot image credential is a signed manifest. The format of this credential and the tools to generate one are described in more detail in the associated Software Development Kit (SDK) specification. For the purpose of this example, the manifest includes the following:

- Integrity data for the boot image.
- The signing IT organization’s certificate. This is the IT organization’s Boot Object Authorization Certificate.
- A digital signature of the entire manifest. This digital signature is generated using the signing organization’s (secret) private key. This private key corresponds to the public key found in the signer’s certificate.

The IT organization puts the boot image file and the corresponding boot image credential files on a Boot Server accessible to the managed platform(s). The managed platform must know how to locate the boot image credential files corresponding to a boot image on the Boot Server.

Now that the boot image and corresponding boot image credential files are stored on a Boot Server, the example shifts to the managed platform boot sequence. A typical boot sequence starts becoming interesting to Remote-Boot authentication late in the BIOS initialization sequence. At a high level, the relevant sequence of typical steps are:

1. The remote-boot code for a Network Interface Card uses Dynamic Host Configuration Protocol (DHCP) to obtain a platform IP address, a Boot Server IP address, and a boot file name.
2. The remote-boot code uses the Preboot Execution Environment (PXE) code to download a boot file.
3. The remote-boot code uses the BIS_GetSignatureInfo function described in this document to find a signature algorithm and key-length combination that the platform supports. This also indirectly determines which of the corresponding boot image credential files to download from the Boot Server.
4. The remote-boot code downloads the corresponding boot image credential.
5. The remote-boot code calls the BIS_VerifyBootObject function described in this document to perform the integrity and authorization check of the image. The integrity check must succeed as described in detail in the function specification. The authorization check involves checking the signer’s certificate supplied in the credential. The public key in the certificate is compared against

Copyright © 1998, Intel Corporation. All rights reserved.

the public key in the Boot Object Authorization Certificate configured for this platform. If a match is found, the signature was generated by the accepted authority and the authorization check passes, otherwise the boot attempt fails.

6. Assuming all checks in the previous step pass, the remote-boot code branches to the downloaded boot image, which never returns.
7. This first downloaded boot image is subject to fairly tight size and memory-model constraints. Consequently in the typical example it is merely a first-stage bootstrap. It contains a rudimentary memory manager to make additional memory space available, and a more robust download protocol that can take advantage of the expanded memory. It downloads a second-stage bootstrap using a server, protocol, and file location that may be determined from information obtained in the first-stage download.
8. The second-stage boot image has its own integrity and authorization credentials, separate from the first-stage boot image. The first stage code may use the BIS_VerifyObjectWithCredential function to validate the second stage. This function is similar to the BIS_VerifyBootObject function except that it allows the caller to supply a certificate that shall be recognized as the source of authority. In particular, the source of authority for second-stage signature is typically the vendor of the software being booted, not the IT organization that manages the platform.
9. Assuming the checks in the previous step pass, the first stage code invokes the second stage code, which never returns.
10. The bootstrap process may involve several more stages, with each stage downloading the credential and image of the next, and validating it before executing it.
11. Eventually, an OS typically has enough capabilities initialized that it no longer needs the BIS service and can take advantage of the memory space that the BIS service occupies. To ensure clean termination of the BIS service, the client code calls a BIS_Shutdown function to terminate the service. The memory occupied by the BIS service may then be reclaimed, managed, and over-written by an OS. This makes the BIS service no longer available.

Most of this usage model example involves activities outside the direct scope of the interfaces defined or referenced in this document. The parts actually within this interface scope are:

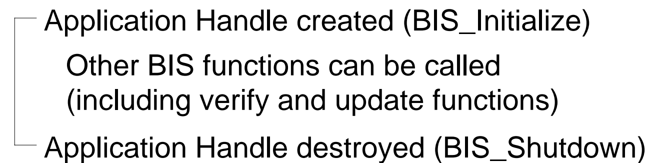
- The Boot Object Authorization Certificate and Boot Authorization Check Flag as platform configuration parameters.
- The generalized BIS_UpdateBootObjectAuthorization function for storing these parameters in a persistent and protected manner.
- A boot image credential including the certificate used to sign the boot image credentials. The public key in the signer's certificate must match the public key in the Boot Object Authorization Certificate of this platform.
- BIS_GetSignatureInfo function to find a signature algorithm and key-length combination that the platform supports.
- BIS_VerifyBootObject function to test integrity and authorization of the boot image.
- BIS_VerifyObjectWithCredential function for implementing a more customized authentication model.
- BIS_Shutdown function for terminating BIS usage and preparing to reclaim its resources.

Copyright © 1998, Intel Corporation. All rights reserved.

It is the responsibility of other parts of software inside and outside of the managed platform to perform the remaining parts of the usage model.

2.4 Initialization and Shutdown

The functions in this interface impose some restrictions over the allowable sequences in which they may be called. These restrictions can best be visualized as functions that must be nested within the lifetime of a handle created at initialization time. The handle lifetime and function permissions are shown in the following time line:



A handle lifetime (square bracket) and associated function-call permissions are shown, with time proceeding downwards through the diagram. The BIS service is initialized with BIS_Initialize, starting the lifetime of an application handle. Once this function returns, other BIS functions can be called to verify objects, update parameters, and so forth. These functions take an application handle as a parameter. The lifetime of the application handle is eventually ended by the BIS_Shutdown function.

Figure 3: Function availability nested within handle lifetime

The first BIS function that must be called is BIS_Initialize. This function returns a handle representing an "instance" of initialization of the BIS service. General-purpose BIS functions all require this handle as a parameter, so they can be called only when the handle is valid. Eventually the BIS_Shutdown function is called. This function invalidates the handle. The handle may no longer be used to call other BIS functions. BIS_Initialize and BIS_Shutdown can be invoked multiple times. Each BIS_Initialize call must be paired with a matching BIS_Shutdown call for that application handle. The lifetimes of the handles created and destroyed may overlap in any way. BIS is "active" any time there is an outstanding handle.

The BIS API is structured to support the development of "BIS Redirector" implementations. A BIS Redirector presents the BIS API and functionality to its callers. However, a BIS Redirector redirects invocation of BIS functions to a remote target platform's implementation of the BIS API. A BIS Redirector typically runs in the OS-present environment and support the needs of management applications that remotely configure BIS enabled platforms. As part of this support, BIS_Initialize includes a "targetAddress" parameter that provides the address of the target platform. Each BIS Redirector implementation defines the format and interpretation of this parameter for that implementation. The local platform implementation of BIS requires that a specific default value is supplied for "targetAddress", but otherwise ignores the value.

Copyright © 1998, Intel Corporation. All rights reserved.

3. Interface Description

The following sub-sections describe the interface in detail, broken down by major topics. Generally, topics are categories of functions and major data structures managed by the interface described in this document. Sub-sections covering a category of functions give the fully detailed descriptions of the functions in the category.

3.1 BIS Entry Point Structure

The BIS Entry Point Structure is an SMBIOS table entry, following the SMBIOS table entry layout guidelines. The structure can be discovered through the SMBIOS structure retrieval process as defined in the [SMBIOS Spec]. Briefly, the BIS Entry Point Structure consists of:

- A standard SMBIOS structure header including a Type indicator, a Length field, and a Handle field whose value is assigned at run-time.
- The main body of the structure, containing the contents of interest to BIS callers.
- A two-byte null terminator. These two bytes are NOT included in the Length value in the header.

The exact procedure for finding the BIS Entry Point Structure is described in the [SMBIOS Spec]. As part of that procedure, the caller needs to know the unique Type value allocated for this structure.

The Type value for the BIS Entry Point Structure is 31 (decimal).

The BIS Entry Point Structure is described below as a “byte alignment packed” C structure:

3.1.1 pBisEntry16, pBisEntry32

```
#if defined(COMPILER_IS_16_BIT)
    typedef UINT8 (_cdecl __far *pBisEntry16)(
        UINT32    opCode,                 // BIS opcode.
        UINT32    pParamBundle,           // Opcode's parm bundle.
        UINT32    checkFlag);             // Checksum bis request flag.
#else
    typedef UINT32 pBisEntry16;           // Not used if 32 bit compiler.
#endif

#if defined(COMPILER_IS_32_BIT)
    typedef UINT8 (_cdecl *pBisEntry32)(
        UINT32    opCode,                 // BIS opcode.
        void     *pParamBundle,           // Opcode's parm bundle.
        UINT32    checkFlag);             // Checksum bis request flag.
#else
    typedef UINT32 pBisEntry32;           // Not used if 16 bit compiler.
#endif
```


Copyright © 1998, Intel Corporation. All rights reserved.

These types give fully-prototyped procedure pointers for the two functions BIS callers invoke for all BIS operations. The pBisEntry16 type gives the function prototype used by 16-bit real-mode callers. The pBisEntry32 type gives the function prototype used by 32-bit flat physical address mode callers. In both cases, the actual BIS operation to be requested through these functions is selected by an operation code. The actual parameters for the operation are passed in and out through a parameter bundle structure whose contents vary depending on the operation.

Definitions:

opCode - An operation code selecting the appropriate BIS operation to be performed.

pParamBundle - A flat 32-bit physical address pointer to the data structure used for passing in and out parameters for the BIS operation.

checkFlag - A flag value that can be used to request an integrity check of the BIS implementation before performing the BIS operation.

(return value) - Zero indicates that no implementation integrity check failure was detected. Non-zero indicates that an implementation integrity check failed.

3.1.2 BIS Operation Codes

```
#define BISOPBASE (0)

#define BISOP_Initialize                                (BISOPBASE+ 1)
#define BISOP_Free                                      (BISOPBASE+ 2)
#define BISOP_Shutdown                                  (BISOPBASE+ 3)
#define BISOP_GetBootObjectAuthorizationCertificate     (BISOPBASE+ 4)
#define BISOP_VerifyBootObject                          (BISOPBASE+ 5)
#define BISOP_GetBootObjectAuthorizationCheckFlag       (BISOPBASE+ 6)
#define BISOP_GetBootObjectAuthorizationUpdateToken     (BISOPBASE+ 7)
#define BISOP_UpdateBootObjectAuthorization             (BISOPBASE+ 8)
#define BISOP_VerifyObjectWithCredential                (BISOPBASE+ 9)
#define BISOP_GetSignatureInfo                          (BISOPBASE+ 10)

#define BISOP_LAST (BISOP_GetSignatureInfo)
```

These C preprocessor symbols define the operation codes for BIS operations. These values are used as actual parameters for the opCode formal parameter of either the 16-bit or 32-bit BIS entry point function.

Definitions:

BISOPBASE - A C preprocessor symbol defining a base operation code value. This value is not used as an operation code, however, all actual operation codes are defined relative to this value.

BISOP_LAST - A C preprocessor symbol equal to the highest BIS operation code. This C preprocessor symbol will be redefined as appropriate if new BIS operation codes are specified in a future interface version.

Copyright © 1998, Intel Corporation. All rights reserved.

3.1.3 BIS_ENTRY_POINT

```
typedef struct _BIS_ENTRY_POINT
{
    UINT8  smBiosType;
    UINT8  length;
    UINT16 structHandle;
    UINT8  structChecksum;
    UINT8  reserved1;
    UINT16 reserved2;
    pBisEntry16 bisEntry16;
    pBisEntry32 bisEntry32;
    UINT64 reserved3;
    UINT32 reserved4;
    UINT16 doubleNull;
}
BIS_ENTRY_POINT,
*pBIS_ENTRY_POINT;
```

This type defines the SMBIOS table entry holding information about the BIS interface.

Definitions:

smBiosType - The SMBIOS type value reserved for the BIS Entry Point Structure. The value is 31 (decimal).

length - The length of this structure in bytes, not including the two-byte NULL terminator. The value is sizeof(BIS_ENTRY_POINT) - 2.

structHandle - A handle value assigned by SMBIOS during system initialization.

structChecksum - Used to make the 8-bit checksum of this structure equal zero.

reserved1 - Reserved for future use. The value must be set to zero until a use is defined.

reserved2 - Reserved for future use. The value must be set to zero until a use is defined.

bisEntry16 - BIS entry point pointer for use by 16-bit real-mode callers. This is a segmented pointer (in segment:offset form).

bisEntry32 - BIS entry point pointer for use by 32-bit flat physical address mode callers. This is a 32-bit physical address.

reserved3 - Reserved for future use. The value must be set to zero until a use is defined.

reserved4 - Reserved for future use. The value must be set to zero until a use is defined.

doubleNull - 0000h structure terminator. This is described in more detail in the [SMBIOS Spec].

3.1.4 BIS_ENTRY_POINT_TYPE

```
#define BIS_ENTRY_POINT_TYPE ((UINT8)31)
```

This C preprocessor symbol gives the value for the BIS_ENTRY_POINT.smBiosType field. For more information on reserved SMBIOS type values, see the [SMBIOS Spec].

3.2 BIS Calling Convention

The structure described in the previous section includes two entry points. The two entry points are for callers in different processor modes.

Copyright © 1998, Intel Corporation. All rights reserved.

- Real Mode: In Real Mode all addresses are composed of Segment:Offset pairs, with both Segment and Offset being 16 bits. These combine to form a 20-bit physical address that cannot access above approximately the 1 Megabyte boundary. This mode is referred to in this section as "16-bit real-mode" or "16-bit mode". Callers in this mode may only invoke BIS through the bisEntry16 entry point. The bisEntry16 function returns with the processor in 16-bit real-mode.
- Flat Mode: A 32-bit (protected) IA-32 processor mode where CS:0, DS:0, and SS:0 all refer to physical location 0 and all have 4 GB of address space. This mode is referred to in this section as "32-bit flat-mode" or "32-bit mode". Callers in this mode may only invoke BIS through the bisEntry32 entry point. The bisEntry32 function returns with the processor in 32-bit flat-mode.

Invoking BIS in any mode other than these supported modes is not supported and may cause an unpredictable failure.

The entry points for the two different modes are the addresses of common dispatch functions that dispatch to the actual BIS functions described in detail later in this document. The actual BIS function descriptions later in this document are given in the form of example calls to the 32-bit dispatch entry point with parameters set up to invoke the appropriate BIS function. The input and output parameters of each actual BIS function differ from one another. This section describes how callers invoke BIS functions with differing parameter lists through the appropriate dispatch function, which has a fixed parameter list.

The dispatch function (for either 16-bit real-mode callers or 32-bit flat-mode callers) has the following parameters:

- An Operation Code. This parameter is a pre-defined integer that selects the appropriate BIS function. This is a 32-bit quantity for both the 32-bit and 16-bit dispatch functions. A file defining the exact value for each of the functions will be included with the SDK.
- A pointer to a Parameter Bundle. The parameter bundle is a unique data structure for each different BIS function. The BIS functions described later in this document give a detailed description of the appropriate parameter bundle type for each function. In most cases the parameter bundle for each function consists of a "sizeOfStruct" field for consistency checking; a "returnValue" field indicating the success or failure result of the BIS function; an opaque "appHandle" field whose value was retrieved at initialization time, followed by function-specific input and output parameters. Within these parameter-bundle structures, all pointers are flat 32-bit physical addresses. Integer and boolean quantities are 32-bit fields. Output parameter values are written into the structure itself. The pointer to the parameter bundle itself is a flat 32-bit physical address for both the 16-bit and 32-bit dispatch functions. A file defining structure types for each of the parameter bundles will be included with the SDK.
- An Integrity-Check flag. This flag indicates whether or not the dispatch function should do an integrity check of the BIS implementation before performing the function. A non-zero (BIS_TRUE) flag value indicates that the check should be done. If the check is requested and fails, the BIS function is not performed and the dispatch function returns non-zero, otherwise the BIS function is performed and the dispatch function returns zero to indicate that no integrity fault was detected. Callers should use the Integrity-Check flag judiciously, since the integrity check may be computationally intensive. The check flag is a 32-bit quantity for both the 16-bit and 32-bit dispatch function.

Copyright © 1998, Intel Corporation. All rights reserved.

Both dispatch functions have an 8-bit return value. A non-zero return value indicates that an implementation integrity check failure was detected. A zero value indicates that no integrity check failure was detected.

Dispatch functions for both the 16-bit and 32-bit dispatch functions use the "_cdecl" calling convention. Parameters are pushed on the stack from right to left. The caller is responsible for removing parameters from the stack after the return.

The sub-sections below show code fragments illustrating how the dispatch functions can be called from common language and programming model combinations.

3.2.1 16-bit C example

This example illustrates a typical sequence used in calling the 16-bit dispatch function from 16-bit real-mode C code. The example shows some typical declarations and calls the dispatch routine through the Entry Point structure. Details of converting a segmented pointer to a physical address are omitted since callers may choose to do this in different ways depending on their environment.

```
pBIS_ENTRY_POINT     EntryStruct;
BIS_VBO_PARMS ParamBundle;
UINT8                DispatchReturn;
UINT32               ParamBundlePtrPhys;

. . .

// Convert Segmented pointer to physical address
ParamBundlePtrPhys = ConvertToPhysical(
    (void FAR *) & ParamBundle);  // SegmentedPointer

// Call the dispatch routine
DispatchReturn = (* EntryStruct->bisEntry16)(
    BISOP_VerifyBootObject         // opCode
    ParamBundlePtrPhys,            // pParamBundle
    BIS_TRUE);                     // checkFlag
```

3.2.2 16-bit Assembly example

This example illustrates a typical sequence used in calling the 16-bit dispatch function from 16-bit real-mode assembly code. The example is the equivalent of the 16-bit real-mode C code example in the previous section, except that only the calling sequence is illustrated. The "_cdecl" calling convention is used, so parameters are pushed on the stack from right to left. The caller is responsible for removing parameters from the stack after the return. The return value comes back in the 8-bit AL register. The call itself is a "FAR" call, that is, the address consists of a 16-bit segment and a 16-bit offset.

Copyright © 1998, Intel Corporation. All rights reserved.

```
PUSH      00                                ; checkFlag Most Significant
                                            ; 16-bit word
PUSH      01                                ; checkFlag Least Significant
                                            ; 16-bit word
PUSH      WORD PTR [ParamBundlePtrPhys_MSW]  ; pParamBundle Most
                                             ; Significant
                                            ; 16-bit word
PUSH      WORD PTR [ParamBundlePtrPhys_LSW]  ; pParamBundle Least
                                             ; Significant
                                            ; 16-bit word
PUSH      00                                ; opCode Least Significant
                                            ; 16-bit word
PUSH      05                                ; opCode Most Significant
                                            ; 16-bit word
LES       BX,[EntryStruct]
CALL      DWORD PTR ES:[BX+08]              ; call EntryStruct->bisEntry16
ADD       SP,0C                             ; Remove parameters from stack
MOV       BYTE PTR [DispatchReturn],AL   ; Result
```

3.2.3 32-bit C example

This example illustrates a typical sequence used in calling the 32-bit dispatch function from 32-bit flat-mode C code. The example shows some declarations and calls the dispatch routine through the Entry Point structure. In this example no conversion of the parameter bundle pointer is required, since the caller's pointer is already a 32-bit physical address.

```
pBIS_ENTRY_POINT     EntryStruct;
BIS_VBO_PARMS        ParamBundle;
UINT8                DispatchReturn;

. . .

// Call the dispatch routine
DispatchReturn = (* EntryStruct->bisEntry32)(
    BIS_VBO_PARMS,                 // opCode
    (void *) & ParamBundle,        // pParamBundle
    BIS_TRUE);                     // checkFlag
```

3.2.4 32-bit Assembly example

This example illustrates a typical sequence used in calling the 32-bit dispatch function from 32-bit flat-mode assembly code. The example is the equivalent of the 32-bit flat-mode C code example in the previous section, except that only the calling sequence is illustrated. The "_cdecl" calling convention is used, so parameters are pushed on the stack from right to left. The caller is responsible for removing parameters from the stack after the return. The return value comes back in the AL register. The call itself is a call through a 32-bit address. The "FAR" modifier for pointer declarations has no effect in the 32-bit compilation environment.

```
push        1                               ; checkFlag
lea         eax,dword ptr [ParamBundle]
push        eax                             ; pParamBundle
push        5                               ; opCode
mov         ecx,dword ptr [EntryStruct]
call        dword ptr [ecx+0Ch]             ; call EntryStruct->bisEntry32
add         esp,0Ch                         ; Remove parameters from stack
mov         byte ptr [DispatchReturn],al    ; Result
```


Copyright © 1998, Intel Corporation. All rights reserved.

3.3 Additional BIS caller requirements

Code that calls BIS functions (through the dispatch functions as described earlier) must observe several additional requirements:

- The caller must supply at least 1KB of available stack space.
- The BIS functions may execute with interrupts turned off, for either 16-bit or 32-bit callers. Although the original setting of interrupts will be restored when the BIS function returns, the caller should be prepared to have interrupts disabled for up to 100 milliseconds for any of the BIS functions except BIS_UpdateBootObjectAuthorization, which may disable interrupts for up to 10 seconds.
- The BIS functions may be non-reentrant. Generally, this should not be an issue for callers, since with interrupts turned off, there is little opportunity for the functions to be called in a reentrant fashion.
- Some IA-PC platform chipsets have special address-line A20 gating logic external to the processor that can be set up to mimic a memory addressing wraparound characteristic of the 8086 processor at the 1MB boundary. Upon return from BISEntry16 the system is in its "standard" boot-time real-mode configuration, either 16-bit real mode or big real mode. For example, if the BIOS calls option ROMS at boot time in big real mode, then the return from BISEntry16 will have the system in big real mode.

The BIOS provides a "Query System Address Map" call (INT 15, E820h) used to explore the system memory map. The Query System Address Map call is described in [ACPI] in more detail than this brief summary. The Query System Address Map call returns a memory map of all the installed RAM and of physical memory ranges reserved by the BIOS. The address map is returned by making successive Query System Address Map calls, each returning one run of physical address information. Each run has a type that dictates how this run of physical address range is to be treated by the operating system. The types defined include a memory classification (type 3) for "ACPI Reclaim Memory". A range of memory addresses with this memory classification is available RAM usable by the operating system after it reads the ACPI tables. In general, this (type 3) memory classification is preserved until the target OS has read ACPI information, at which time the OS may reclaim the space and overwrite the memory.

If the BIS Services are implemented using Extended memory, the required Extended memory space shall be reported as type 3 by the INT 15, E820h Query System Address Map call. This allows an OS aware of type 3 memory classification to reclaim this memory after Initial Program Load completion. Callers that depend on BIS services should avoid overwriting this memory classification. Once this memory is overwritten, BIS services are no longer available, and further BIS functions shall not be called. These rules allow the BIS caller to preserve the ability to use the service as long as needed, while still allowing the memory space to be reclaimed for use by an OS.

Note: ACPI-unaware operating systems such as Windows 95 and NT† 4 are unaware of the "type 3" memory classification. They will treat this memory classification as (permanently) "reserved" and never reclaim it.

Copyright © 1998, Intel Corporation. All rights reserved.

3.4 Signature Algorithms and Key Lengths Supported

The functions in these interfaces are designed to work with a variety of different digital signature algorithms and key lengths. Generally they will work with whatever is supplied with the platform. Digital signatures are self-describing in the sense that the algorithm(s) and key lengths used to produce the signature are specified in the signature itself. Thus operations involving verifying existing signatures generally do not need an explicit parameter describing the signature algorithm(s) to be used. This information is conveyed implicitly by the signature itself.

The BIS interface includes a function, BIS_GetSignatureInfo, that allows the caller to query the platform to see what combinations of digital signature algorithms, hash algorithms, and key lengths are supported. Together with the self-describing nature of signatures themselves, this mechanism allows an application to "negotiate" which type of digital signature to supply.

Every conforming platform must support at least one of a required set of combinations of digital signature algorithms, hash algorithms, and key lengths. A platform may support additional algorithms, key lengths, and combinations as long it supports at least one out of the required set. The required set is:

- Signature algorithm is DSA, (which implies hash algorithm is SHA-1), with 1024-bit key length.
- Signature algorithm is RSA, hash algorithm is MD5, with 512-bit key length.

Note that digital signatures verified through the BIS interface are usually produced on a different platform. To guarantee interoperability with any platform complying with this specification, the external platform must support all of the combinations listed above.

3.5 Digital Certificates

Digital certificates used in this interface must have a standardized data representation format to allow interoperability between suppliers and consumers of the certificates. The standardized representation format is X.509v3, defined in [X.509]. The X.509v3 standard is a complex standard allowing for an open-ended set of attributes to be included in certificates, defining a variety of different data encodings, and so forth. To keep the implementation simple, this specification requires only a small subset of the full possibilities of X.509v3. In particular, to be used with BIS, X.509v3 digital certificates must include at least the following:

- The "subject" public key.
- The "issuer" public key.
- The signature of the certificate, generated with the issuer's private key.

Other attributes may be included, but they are not used by this specification. Certificates vary in size according to the additional attributes included, the key lengths, the signature algorithm, etc. A "minimal" certificate for one of the required set of signature algorithm and key length combinations would fit within 2KB. For various reasons certificates may need to include additional attributes both now and in the future. To allow such additions, the BIS implementation is required to be able to handle digital certificates of at least 4KB in size.

Copyright © 1998, Intel Corporation. All rights reserved.

This includes the requirement to be able to store a Boot Object Authorization Certificate of at least this minimum requirement.

An implementation of BIS may choose to support larger digital certificates. Developers should be cautious about using large certificates, however. Certificates that need to be stored on the platform compete for limited persistent storage space. Even certificates that are only used temporarily on the platform compete for memory space in an environment with limited memory-management capabilities.

Digital certificates must obey an additional uniformity requirement to be interoperable with BIS. The certificate must match one of the digital signature algorithm, hash algorithm, and key length algorithm combinations supported by the platform (referred to as the "matching signature combination"). In particular:

- The issuer public key and subject public key must be the same length as the key length in the matching signature combination.
- The certificate's signature must have been generated using the same digital signature algorithm and hash algorithm as in the matching signature combination.

3.6 Signed Manifests

Signed manifests are described more completely in [SM Spec]. This is a brief summary for those familiar with PKCS#7 signed data. Signed manifests are similar to PKCS#7 signed data. The main differences are:

- A signed manifest may contain integrity information for multiple data objects.
- For each data object, a manifest may contain an optional reference to locate the data object and "metadata" associated with the object in the form of <name, value> attribute pairs. The manifest also ensures the integrity of each data object and its associated metadata.
- A signed manifest may contain multiple signature blocks generated by different signers. Each signature block covers a *subset* (one, some, or all) of the data objects.
- A signed manifest contains only integrity information and metadata, not the data objects themselves. *Note:* If the data to be signed comprises only attributes, then a signed manifest can represent the data in a standalone fashion.

In the simple case of a signed manifest for one external object with one signer, the manifest acts just like a PKCS#7 structure that does not contain the signed data. BIS functions are used to verify the integrity and authenticity of a signed manifest and the associated data objects.

Typically signed manifests are created in a controlled, manufacturing-like environment. Tools are available for OS-present environments that enable general manifest construction.

As with digital certificates, signed manifests must obey a uniformity requirement to be interoperable with BIS. The certificates, signatures, and hashes in the manifest must match one of the digital signature algorithm, hash algorithm, and key length algorithm combinations supported by the platform (referred to as the "matching signature combination"). In particular:

Copyright © 1998, Intel Corporation. All rights reserved.

- All certificates in the signer's certificate chain must obey the uniformity requirements for certificates as described above, with the matching signature combination.
- All hashes used as integrity data for the manifest's data objects must use the same hash algorithm as the matching signature combination.

3.7 Request Credentials

A request credential, as used by the interface described in this document, is a special type of signed manifest. This type of manifest carries enough information to describe an authenticated parameter-update request. In particular, the manifest includes at least these items:

- An unique token for the parameter set containing the parameters to be updated by this request. The token is designed to guard against attacks based on capturing and replaying an identical or altered update request to the same or different target platform.
- The identities of all the parameters to be updated.
- The corresponding new values.
- A signature of the entire manifest.

The interfaces described here do not include explicit functions for fully interpreting the contents of a request credential. The internal manipulation of a request credential for updating a configuration parameter is an internal implementation detail of the supplied function that performs the update on a managed platform. Nevertheless, software developers need a complete description of how to create a properly constructed request credential for this purpose. The required syntax of a request credential is described in the 3.9.5 section.

3.8 Data structures

This section describes the general-purpose BIS data types, constant declarations, and data structures in detail in the C syntax form in which they appear in header files.

To support both 16-bit and 32-bit callers in a unified way, all pointers used in the interface are 32-bit physical addresses. The pointer declarations are set up so that in the 32-bit C compilation environment, these can be used directly as pointers. In the 16-bit C compilation environment, pointer declarations are set up as 32-bit quantities so they occupy the correct amount of bytes in data structures, however, these quantities are not declared as pointers. This prevents developers using a 16-bit compilation environment from inadvertently attempting to use a 32-bit physical address as though it were a segmented pointer. Callers of BIS that develop 16-bit code must use other techniques to access memory identified by physical addresses. A full discussion of such techniques is outside the scope of this specification.

IMPORTANT NOTE:

> In the 16-bit C compilation environment, there is no direct language support for 32-bit pointers. The pointer declarations are set up so that pointers are 32-bit unsigned integers in the 16-bit C compilation environment. The compiler will flag any accidental attempt to use these directly as pointers. It is up to the caller to generate physical addresses for

Copyright © 1998, Intel Corporation. All rights reserved.

pointers passed to the BIS interface and appropriately access memory referenced by physical address pointers returned from the BIS interface.

3.8.1 COMPILER_IS_16_BIT and COMPILER_IS_32_BIT

```
#define COMPILER_IS_16_BIT
#define COMPILER_IS_32_BIT
```

The developer should #define only one of these C preprocessor before any #include for the BIS header files. These symbols determine whether the BIS header files compile to definitions for 16-bit or 32-bit compilation, respectively. If neither of these C preprocessor symbols is defined, COMPILER_IS_32_BIT is assumed.

3.8.2 BIS_GEN_SCS_CONST

```
#define BIS_GEN_SCS_CONST
```

Defining this C preprocessor symbol causes compilation of BIS headers to emit source-control version stamps in the resulting object files. Generally, a developer should not #define this C preprocessor symbol.

3.8.3 UINT8, UINT16, UINT32, UINT64, INT8, INT16, INT32

```
#if defined(COMPILER_IS_32_BIT)
    typedef unsigned __int8         UINT8;
    typedef unsigned __int16        UINT16;
    typedef unsigned __int32        UINT32;
    typedef unsigned __int64        UINT64;
    typedef          __int8         INT8;
    typedef          __int16        INT16;
    typedef          __int32        INT32;
#endif

#if defined(COMPILER_IS_16_BIT)
    typedef unsigned char           UINT8;
    typedef unsigned short          UINT16;
    typedef unsigned long           UINT32;
    typedef struct _uint64 { UINT8 eightbytes[8]; } UINT64;
    typedef          char           INT8;
    typedef          short          INT16;
    typedef          long           INT32;
#endif
```

These types define integer quantities of the indicated sizes. The types prefixed with “INT” are signed integers. The types prefixed with “UINT” are unsigned integers. Note that many popular 16-bit compilers do not provide 64-bit types supporting arithmetic operations, so the 64-bit unsigned integer type is defined simply as a structure occupying 8 bytes of storage in the case of 16-bit compilation.

Copyright © 1998, Intel Corporation. All rights reserved.

3.8.4 BIS_APPLICATION_HANDLE

```
#if defined(COMPILER_IS_32_BIT)
    typedef void*                    BIS_APPLICATION_HANDLE;
    typedef BIS_APPLICATION_HANDLE* BIS_APPLICATION_HANDLE_PTR;
#endif

#if defined(COMPILER_IS_16_BIT)
    typedef UINT32                   BIS_APPLICATION_HANDLE;
    typedef UINT32                   BIS_APPLICATION_HANDLE_PTR;
#endif
```

This type is an opaque handle representing an initialized instance of the BIS interface. A BIS_APPLICATION_HANDLE value is returned by the BIS_Initialize function as an "out" parameter. Other BIS functions take a BIS_APPLICATION_HANDLE as an "in" parameter to identify the BIS instance.

The 16-bit pointer type is declared as a 32-bit unsigned integer instead of a pointer because the actual value used by the BIS interface is a 32-bit physical address. Any necessary translation between physical address and 16-bit segment:offset addressing must be accomplished by the caller.

3.8.5 BIS_ALG_ID

```
typedef UINT32                  BIS_ALG_ID;
```

This type represents a digital signature algorithm. A digital signature algorithm is often composed of a particular combination of secure hash algorithm and encryption algorithm. This type also allows for digital signature algorithms that cannot be decomposed.

3.8.6 Predefined BIS_ALG_ID values

```
#define BIS_ALG_DSA     (41)    //CSSM_ALGID_DSA
#define BIS_ALG_RSA_MD5 (42)    //CSSM_ALGID_MD5_WITH_RSA
```

These values represent the two digital signature algorithms predefined for BIS. Each implementation of BIS must support at least one of these digital signature algorithms. Values for the digital signature algorithms are chosen by an industry group known as The Open Group. Developers planning to support additional digital signature algorithms or define new digital signature algorithms should refer to The Open Group for interoperable values to use.

3.8.7 BIS_BOOLEAN

```
#if defined(COMPILER_IS_32_BIT)
    typedef UINT32                   BIS_BOOLEAN;
    typedef BIS_BOOLEAN*             BIS_BOOLEAN_PTR;
#endif

#if defined(COMPILER_IS_16_BIT)
    typedef UINT32                   BIS_BOOLEAN;
    typedef UINT32                   BIS_BOOLEAN_PTR;
    typedef UINT32                   BIS_BYTE_PTR;
#endif
```

This type is used to represent TRUE and FALSE values. When testing values of this type, developers are advised to use the standard conservative-programming practice of testing

Copyright © 1998, Intel Corporation. All rights reserved.

boolean expressions rather than testing for exact equality to predefined values. (E.g., use "if (expression)" rather than "if (variable == BIS_TRUE)").

The 16-bit pointer type is declared as a 32-bit unsigned integer instead of a pointer because the actual value used by the BIS interface is a 32-bit physical address. Any necessary translation between physical address and 16-bit segment:offset addressing must be accomplished by the caller.

3.8.8 Predefined BIS_BOOLEAN values

```
#define BIS_TRUE  (1)
#define BIS_FALSE (0)
```

These represent TRUE and FALSE values for the BIS_BOOLEAN type.

3.8.9 BIS_BYTE_PTR

```
#if defined(COMPILER_IS_32_BIT)
    typedef UINT8*                  BIS_BYTE_PTR;
#endif

#if defined(COMPILER_IS_16_BIT)
    typedef UINT32                  BIS_BYTE_PTR;
#endif
```

This type represents a pointer to a data buffer. It is generally used as a component of other structured types.

The 16-bit pointer type is declared as a 32-bit unsigned integer instead of a pointer because the actual value used by the BIS interface is a 32-bit physical address. Any necessary translation between physical address and 16-bit segment:offset addressing must be accomplished by the caller.

3.8.10 BIS_STATUS

```
typedef UINT32                  BIS_STATUS;
```

This type is a result code that comes back from each BIS function as an "out" parameter., It indicates whether the function was successful or not and possibly gives some detailed information on a reason for failure.

Copyright © 1998, Intel Corporation. All rights reserved.

3.8.11 Predefined BIS_STATUS values

```
#define BIS_OK                  (0)
#define BIS_INVALID_OPCODE      (1) //Returned by BIS_FunctionDispatch()
#define BIS_INVALID_PARMSTRUCT  (2) //Null parm or bundle length is wrong.
#define BIS_MEMALLOC_FAILED     (3) //Couldn't alloc requested memory.
#define BIS_BAD_APPHANDLE       (4) //Invalid BIS_APPLICATION_HANDLE passed.
#define BIS_NOT_IMPLEMENTED     (5) //Unimplemented BIS function called.
#define BIS_BAD_PARM            (6) //A parm in the parm struct is invalid.
#define BIS_BOA_CERT_READ_ERR   (7) //An error occurred on CERT READ.
#define BIS_BOA_CERT_NOTFOUND   (8) //A BOA CERT is not configured.
#define BIS_SECURITY_FAILURE    (9) //A security check failed.
#define BIS_INIT_FAILURE        (10) //An internal failure occured in init.
#define BIS_INCOMPAT_VER        (11) //BIS interface version requested is
                                     //not compatible with available version
#define BIS_NVM_AREA_IO_LENGTH_ERROR    (12) //Length+offset combination is
                                             //incorrect.
#define BIS_NVM_AREA_UNKNOWN            (13) //Unknown area guid was
                                             //specified.
#define BIS_NVM_CREATE_ERR_NO_ROOM      (14) //No space to create the new
                                             //area.
#define BIS_NVM_CREATE_ERR_DUPLICATE_ID (15) //Area already exists.
#define BIS_NVM_BAD_HANDLE              (16) //Invalid handle passed.
#define BIS_NVM_PSI_FXNS_NOT_AVAIL      (17) //Bad/No PSI fxns passed
                                             //to BIS_main.
```

These predefined values for the BIS_STATUS type describe the success or failure of BIS functions.

3.8.12 BIS_NULL

```
#if defined(COMPILER_IS_32_BIT)
#define BIS_NULL ((void*)0)
#endif

#if defined(COMPILER_IS_16_BIT)
#define BIS_NULL (0)
#endif
```

This constant defines a default “invalid” value for any pointer types.

3.8.13 BIS_DATA

```
typedef struct _BIS_DATA
{
    UINT32       length;        //Length of data in 8 bit bytes.
    BIS_BYTE_PTR data;          //32 Bit Flat Address of data.
}
BIS_DATA;

#if defined(COMPILER_IS_32_BIT)
typedef struct _BIS_DATA  *BIS_DATA_PTR;
typedef struct _BIS_DATA **BIS_DATA_PTR_PTR;
#endif

#if defined(COMPILER_IS_16_BIT)
typedef UINT32  BIS_DATA_PTR;
typedef UINT32  BIS_DATA_PTR_PTR;
#endif
```


Copyright © 1998, Intel Corporation. All rights reserved.

This type defines a structure that describes a buffer. BIS uses this type to pass back and forth most large objects such as digital certificates, strings, etc.. Several of the BIS functions allocate a BIS_DATA_PTR and return it as an "out" parameter. The caller must eventually free any allocated BIS_DATA_PTR using the BIS_Free function.

Definitions:

length - The length of the data buffer in bytes.

data - A 32-bit physical-address pointer to the raw data buffer.

The 16-bit pointer type is declared as a 32-bit unsigned integer instead of a pointer because the actual value used by the BIS interface is a 32-bit physical address. Any necessary translation between physical address and 16-bit segment:offset addressing must be accomplished by the caller. This consideration applies to the internal "data" buffer pointer inside BIS_DATA as well.

3.8.14 BIS_VERSION

```
typedef struct _BIS_VERSION
{
    UINT32 major;               //BIS Interface version number.
    UINT32 minor;               //Build number.
}
BIS_VERSION;

#if defined(COMPILER_IS_32_BIT)
typedef struct _BIS_VERSION *BIS_VERSION_PTR;
#endif

#if defined(COMPILER_IS_16_BIT)
typedef UINT32  BIS_VERSION_PTR;
#endif
```

This type represents a version number of the BIS interface. This is used as an "in out" parameter of the BIS_Initialize function for a simple form of negotiation of the BIS interface version between the caller and the BIS implementation.

Definitions:

major - This describes the major BIS version number. The major version number defines version compatibility. That is, when a new version of the BIS interface is created with new capabilities that are not available in the previous interface version, the major version number is increased.

minor - This describes a minor BIS version number. This version number is increased whenever a new BIS implementation is built that is fully interface compatible with the previous BIS implementation. This number may be reset when the major version number is increased.

The 16-bit pointer type is declared as a 32-bit unsigned integer instead of a pointer because the actual value used by the BIS interface is a 32-bit physical address. Any necessary translation between physical address and 16-bit segment:offset addressing must be accomplished by the caller.

3.8.15 BIS_CURRENT_VERSION_MAJOR, BIS_VERSION_1

```
#define BIS_CURRENT_VERSION_MAJOR   BIS_VERSION_1
#define BIS_VERSION_1                           1
```


Copyright © 1998, Intel Corporation. All rights reserved.

These C preprocessor symbols provide values for the BIS_VERSION.major field. BIS_CURRENT_VERSION_MAJOR represents the latest version. The actual value will be changed as new BIS interface versions get defined. BIS_VERSION_1 is the stable version number of version 1 of the interface. New C preprocessor symbols of the form BIS_VERSION_n will be defined as new versions are defined.

3.8.16 BIS_SIGNATURE_INFO

```
typedef struct _BIS_SIGNATURE_INFO
{
    BIS_ALG_ID    algorithmID;  //A signature algorithm number.
    UINT32        keyLength;    //Length of alg. keys in bits.
}
BIS_SIGNATURE_INFO;

#if defined(COMPILER_IS_32_BIT)
typedef struct _BIS_SIGNATURE_INFO *BIS_SIGNATURE_INFO_PTR;
#endif

#if defined(COMPILER_IS_16_BIT)
typedef UINT32 BIS_SIGNATURE_INFO_PTR;
#endif
```

This type defines a digital signature algorithm and key-length combination that may be supported by the BIS implementation. This type is returned by BIS_GetSignatureInfo to describe the combination(s) supported by the implementation.

Definitions:

algorithmID - A predefined constant representing a particular digital signature algorithm. Often this represents a combination of hash algorithm and encryption algorithm, however, it may also represent a standalone digital signature algorithm.

keyLength - The length of the public key, in bits, supported by this digital signature algorithm.

The 16-bit pointer type is declared as a 32-bit unsigned integer instead of a pointer because the actual value used by the BIS interface is a 32-bit physical address. Any necessary translation between physical address and 16-bit segment:offset addressing must be accomplished by the caller.

3.8.17 BIS_GET_SIGINFO_COUNT

```
#define BIS_GET_SIGINFO_COUNT(bisDataPtr)                  \
        ((bisDataPtr)->length/sizeof(BIS_SIGNATURE_INFO))
```

This macro computes how many BIS_SIGNATURE_INFO elements are contained in a BIS_DATA structure returned from BIS_GetSignatureInfo. It represents the list of supported digital signature algorithm and key-length combinations.

Definitions:

bisDataPtr - Supplies the 32-bit physical-address pointer of the target BIS_DATA structure.

(return value) - The number of BIS_SIGNATURE_INFO elements contained in the array.

This macro is supplied for use only in the 32-bit compilation environment. In the 16-bit compilation environment, it is the caller’s responsibility to accomplish this computation,

Copyright © 1998, Intel Corporation. All rights reserved.

including any necessary translation between physical address and 16-bit segment:offset addressing.

3.8.18 BIS_GET_SIGINFO_ARRAY

```
#define BIS_GET_SIGINFO_ARRAY(bisDataPtr)                    \
        ((BIS_SIGNATURE_INFO_PTR)(bisDataPtr)->data)
```

This macro returns a 32-bit physical address pointer to the BIS_SIGNATURE_INFO array contained in a BIS_DATA structure returned from BIS_GetSignatureInfo representing the list of supported digital signature algorithm and key-length combinations.

Definitions:

bisDataPtr - Supplies the 32-bit physical-address pointer of the target BIS_DATA structure.

(return value) - The 32-bit physical address pointer to the BIS_SIGNATURE_INFO array, cast as a BIS_SIGNATURE_INFO_PTR.

This macro is supplied for use only in the 32-bit compilation environment. In the 16-bit compilation environment, it is the caller's responsibility to accomplish this computation, including any necessary translation between physical address and 16-bit segment:offset addressing.

3.9 Boot Object Authentication functions

As described in earlier sections, this interface includes a set of functions that embody a particular usage model for authenticating boot images. The Boot Object Authentication functions defined in this specification are:

BIS_GetBootObjectAuthorizationCertificate - Retrieves the currently configured Boot Object Authorization Certificate. The caller must supply functions to interpret the contents of the certificate, if needed.

BIS_VerifyBootObject - Verifies the integrity and authorization of a data object according to supplied credentials and the platform's configured Boot Object Authorization Certificate.

BIS_GetBootObjectAuthorizationCheckFlag - Retrieves the currently configured setting of the Boot Authorization Check Flag.

BIS_GetBootObjectAuthorizationUpdateToken - Retrieves a unique token value to be included in the request credential for the next update of any parameter in the Boot Object Authorization set (Boot Object Authorization Certificate and Boot Authorization Check Flag).

BIS_UpdateBootObjectAuthorization - Updates one of the configurable parameters of the Boot Object Authorization set (Boot Object Authorization Certificate or Boot Authorization Check Flag).

The Boot Object Authentication functions are described in detail in this document in the specification sections that follow.

Copyright © 1998, Intel Corporation. All rights reserved.

3.9.1 BIS_GetBootObjectAuthorizationCertificate

```
typedef
struct _BIS_GetBootObjectAuthorizationCertificate_PARMS
{
    UINT32                   sizeOfStruct;   //[in]  Byte length of this
                                             // structure.
    BIS_STATUS               returnValue;    //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE  appHandle;       //[in]  From BIS_Initialize( ).
    BIS_DATA_PTR             certificate;    //[out] Pointer to certificate.
}
BIS_GBOAC_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_GBOAC_PARMS         params;
BIS_APPLICATION_HANDLE  appHandle;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_GetBootObjectAuthorizationCertificate, // opCode
    (void *) & params,                           // pParamBundle
    BIS_TRUE);                                   // checkFlag
```

The function selected with the BISOP_GetBootObjectAuthorizationCertificate operation code retrieves the certificate that has been configured as the identity of the organization designated as the source of authorization for signatures of boot objects.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_GetBootObjectAuthorizationCertificate.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_GBOAC_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_GBOAC_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

certificate (output) - The function writes an allocated BIS_DATA_PTR containing the Boot Object Authorization Certificate object. The caller must eventually free the memory allocated by this function using the function BIS_Free.

Copyright © 1998, Intel Corporation. All rights reserved.

See Also

BIS_VerifyBootObject,
BIS_GetBootObjectAuthorizationCheckFlag,
BIS_Free

Copyright © 1998, Intel Corporation. All rights reserved.

3.9.2 BIS_VerifyBootObject

```
typedef
struct _BIS_VerifyBootObject_PARMS
{
    UINT32                  sizeOfStruct; //[in]  Byte length of this
                                          //structure.
    BIS_STATUS              returnValue;  //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE appHandle;    //[in]  From BIS_Initialize( ).
    BIS_DATA                credentials;  //[in]  Verification signed
                                          //manifest.
    BIS_DATA                dataObject;   //[in]  Boot object to verify.
    BIS_BOOLEAN             isVerified;   //[out] Result of verifcation.
}
BIS_VBO_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_VBO_PARMS           params;
BIS_APPLICATION_HANDLE  appHandle;
BIS_DATA                credentials;
BIS_DATA                dataObject;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
params.credentials  = credentials;
params.dataObject   = dataObject;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_VerifyBootObject,   // opCode
    (void *) & params,        // pParamBundle
    BIS_TRUE);                // checkFlag
```

The function selected with the BISOP_VerifyBootObject operation code verifies the integrity and authorization of the indicated data object according to the indicated credentials. The rules for successful verification depend on whether or not Boot Authorization Check is currently required on this platform.

If Boot Authorization Check is *not* currently required on this platform, no authorization check is performed. However, the following rules are applied for an integrity check:

- In this case, the credentials are optional. If they are *not* supplied (credentials.data is BIS_NULL), no integrity check is performed, and the function returns immediately with a "success" indication and isVerified is BIS_TRUE.
- If the credentials *are* supplied (credentials.data is other than BIS_NULL), integrity checks are performed as follows:
  - Verify the credentials - The credentials parameter is a valid signed Manifest, with a single signer. The signer's identity is included in the credential as a certificate.
  - Verify the data object - The Manifest must contain a section named "memory:BootObject", with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the specified dataObject data.

Copyright © 1998, Intel Corporation. All rights reserved.

- If these checks succeed, the function returns with a "success" indication and isVerified is BIS_TRUE. Otherwise, isVerified is BIS_FALSE and the function returns with a "security failure" indication.

If Boot Authorization Check *is* currently required on this platform, authorization and integrity checks are performed. The integrity check is the same as in the case above, except that it is required. The following rules are applied:

- Verify the credentials - The credentials parameter is required in this case (credentials.data must be other than BIS_NULL). The credentials parameter is a valid signed Manifest, with a single signer. The signer's identity is included in the credential as a certificate.
- Verify the data object - The Manifest must contain a section named "memory:BootObject", with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the specified dataObject data.
- Do Authorization check - This happens one of two ways depending on whether or not the platform currently has a Boot Object Authorization Certificate configured.
  - If a Boot Object Authorization Certificate is not currently configured, this function interacts with the user in a platform-specific way to determine whether the operation should succeed.
  - If a Boot Object Authorization Certificate *is* currently configured, this function uses the Boot Object Authorization Certificate to determine whether the operation should succeed. The public key certified by the signer's certificate must match the public key in the Boot Object Authorization Certificate configured for this platform. The match must be direct, that is, the signature authority cannot be delegated along a certificate chain.
  - If these checks succeed, the function returns with a "success" indication and isVerified is BIS_TRUE. Otherwise, isVerified is BIS_FALSE and the function returns with a "security failure" indication.

Note that if Boot Authorization Check is currently required on this platform this function *always* performs an authorization check, either through user interaction or through a signature generated with the private key corresponding to the public key in the platform's Boot Object Authorization Certificate.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_VerifyBootObject.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_VBO_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Copyright © 1998, Intel Corporation. All rights reserved.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_VBO_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

credentials (input) - A Signed Manifest containing verification information for the indicated data object. The Manifest signature itself must meet the requirements described above. This parameter is optional if Boot Authorization Check is currently not required on this platform (credentials.data may be BIS_NULL), otherwise this parameter is required. The required syntax of the Signed Manifest is described in the Manifest Syntax section below.

dataObject (input) - An in-memory copy of the raw data object to be verified.

isVerified (output) - The function writes BIS_TRUE if the verification succeeded, otherwise BIS_FALSE.

Manifest Syntax

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer's information file, and a signature block file. These three parts along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses should appear exactly as shown. Also note that manifest files and signer's information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single "space" character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Manifest File Example

The manifest file must include a section referring to a memory-type data object with the reserved name as shown in the example below. This data object is the Boot Object to be verified. An example manifest file is shown below.

```
Manifest-Version: 2.0
ManifestPersistentId: (base64 representation of a unique GUID)

Name: memory:BootObject
Digest-Algorithms: SHA-1
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 boot object)
```

A line-by-line description of this manifest file is as follows.

```
Manifest-Version: 2.0
```

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
ManifestPersistentId: (base64 representation of a unique GUID)
```

The left-hand string should appear exactly as shown. The right-hand string should be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base64 encoded. Base64 is a simple encoding scheme for representing binary values that uses only printing characters. Base64 encoding is described in [BASE64].

Copyright © 1998, Intel Corporation. All rights reserved.

`Name: memory:BootObject`

This identifies the section that carries the integrity data for the Boot Object. The string "memory:BootObject" must appear exactly as shown. This string can be found in manifest_defs.h as the preprocessor symbol BOOT_OBJECT_SECTION_NAME. Note that the Boot Object cannot be found directly from this manifest. A caller verifying the Boot Object integrity must load the Boot Object into memory and specify its memory location explicitly to this verification function through the DataObject parameter.

`Digest-Algorithms: SHA-1`

This enumerates the digest algorithms for which integrity data is included for the data object. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm should be "SHA-1". For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm should be "MD5". Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

```
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 boot object)
```

Gives the corresponding digest value for the data object. The value is base64 encoded.

Signer's Information File Example

The signer's information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer's information file carries the integrity data for the corresponding section in the manifest file. An example signer's information file is shown below.

```
Signature-Version: 2.0
SignerInformationPersistentId: (base64 representation of a
 unique GUID)
SignerInformationName: BIS_VerifiableObjectSignerInfoName

Name: memory:BootObject
Digest-Algorithms: SHA-1
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 corresponding manifest section)
```

A line-by-line description of this signer's information file is as follows.

`Signature-Version: 2.0`

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
SignerInformationPersistentId: (base64 representation of a
 unique GUID)
```

The left-hand string should appear exactly as shown. The right-hand string should be a unique GUID for every signer's information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base64 encoded. Base64 is a simple encoding scheme for representing binary values that uses only printing characters. Base64 encoding is described in [BASE64].

`SignerInformationName: BIS_VerifiableObjectSignerInfoName`

The left-hand string should appear exactly as shown. The right-hand string should appear exactly as shown. This string is defined in manifest_defs.h under the preprocessor symbol VERIFIABLE_OBJECT_SIGINFO_NAME.

`Name: memory:BootObject`

This identifies the section in the signer's information file corresponding to the section with the same name in the manifest file described earlier. The string "memory:BootObject" must appear exactly as shown. This string can be found in manifest_defs.h as the preprocessor symbol BOOT_OBJECT_SECTION_NAME.

Copyright © 1998, Intel Corporation. All rights reserved.

`Digest-Algorithms: SHA-1`

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

`SHA-1-Digest: (base64 representation of a SHA-1 digest of the corresponding manifest section)`

Gives the corresponding digest value for the corresponding manifest section. The value is base64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening "Name:" keyword and continues up to, but not including, the next section's "Name:" keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next "Name:" keyword or end-of-file.

Signature Block File Example

A signature block file is a raw binary file (not base64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer's information file. There must be a correspondence between the name of the signer's information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer's information file name of "myinfo.SF", the corresponding DSA signature block file name would be "myinfo.DSA".

The format of a signature block file is defined in [PKCS].

Copyright © 1998, Intel Corporation. All rights reserved.

3.9.3 BIS_GetBootObjectAuthorizationCheckFlag

```
typedef
struct _BIS_GetBootObjectAuthorizationCheckFlag_PARMS
{
    UINT32                  sizeOfStruct;    //[in]  Byte length of this
                                             //structure.
    BIS_STATUS              returnValue;     //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE  appHandle;       //[in]  From BIS_Initialize( ).
    BIS_BOOLEAN             checkIsRequired; //[out] Value of check flag.
}
BIS_GBOACF_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_GBOACF_PARMS        params;
BIS_APPLICATION_HANDLE  appHandle;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_GetBootObjectAuthorizationCheckFlag,  // opCode
    (void *) & params,                          // pParamBundle
    BIS_TRUE);                                  // checkFlag
```

The function selected with the BISOP_GetBootObjectAuthorizationCheckFlag operation code retrieves the current status of the Boot Authorization Check Flag (in other words, whether or not Boot Authorization Check is currently required on this platform).

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_GetBootObjectAuthorizationCheckFlag.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_GBOACF_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_GBOAF_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

checkIsRequired (output) - The function writes the value BIS_TRUE if Boot Authorization Check is currently required on this platform, otherwise the function writes BIS_FALSE.

Copyright © 1998, Intel Corporation. All rights reserved.

3.9.4 BIS_GetBootObjectAuthorizationUpdateToken

```
typedef
struct _BIS_GetBootObjectAuthorizationUpdateToken_PARMS
{
    UINT32                  sizeOfStruct;   //[in]  Byte length of this
                                            //structure.
    BIS_STATUS              returnValue;    //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE  appHandle;      //[in]  From BIS_Initialize( ).
    BIS_DATA_PTR            updateToken;    //[out] Value of update token.
}
BIS_GBOAUT_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_GBOAUT_PARMS        params;
BIS_APPLICATION_HANDLE  appHandle;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_GetBootObjectAuthorizationUpdateToken,  // opCode
    (void *) & params,                            // pParamBundle
    BIS_TRUE);                                    // checkFlag
```

The function selected with the BISOP_GetBootObjectAuthorizationUpdateToken operation code retrieves a unique token value to be included in the request credential for the next update of any parameter in the Boot Object Authorization set (Boot Object Authorization Certificate and Boot Authorization Check Flag). The token value is unique to this platform, parameter set, and instance of parameter values. In particular, the token changes to a new unique value whenever any parameter in this set is changed.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_GetBootObjectAuthorizationUpdateToken.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_GBOAUT_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_GBOAUT_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

Copyright © 1998, Intel Corporation. All rights reserved.

updateToken (output) - The function writes an allocated BIS_DATA_PTR containing the new unique update token value. The caller must eventually free the memory allocated by this function using the function BIS_Free.

See Also

BIS_UpdateBootObjectAuthorization,
BIS_Free

Copyright © 1998, Intel Corporation. All rights reserved.

3.9.5 BIS_UpdateBootObjectAuthorization

```
typedef
struct _BIS_UpdateBootObjectAuthorization_PARMS
{
    UINT32                  sizeOfStruct;   //[in]  Byte length of this
                                            //struct.
    BIS_STATUS              returnValue;    //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE  appHandle;      //[in]  From BIS_Initialize( ).
    BIS_DATA                requestCredential; //[in]  Update Request
                                               //Manifest.
    BIS_DATA_PTR            newUpdateToken;    //[out] Next update token.
}
BIS_UBOA_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_UBOA_PARMS          params;
BIS_APPLICATION_HANDLE  appHandle;
BIS_DATA                requestCredential;
. . .
// Set "in" parameter values
params.sizeOfStruct      = sizeof(params);
params.appHandle         = appHandle;
params.requestCredential = requestCredential;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_UpdateBootObjectAuthorization,  // opCode
    (void *) & params,                    // pParamBundle
    BIS_TRUE);                            // checkFlag
```

The function selected with the BISOP_UpdateBootObjectAuthorization operation code updates one of the configurable parameters of the Boot Object Authorization set (Boot Object Authorization Certificate or Boot Authorization Check Flag). It passes back a new unique update token that must be included in the request credential for the next update of any parameter in the Boot Object Authorization set. The token value is unique to this platform, parameter set, and instance of parameter values. In particular, the token changes to a new unique value whenever any parameter in this set is changed.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_UpdateBootObjectAuthorization.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_UBOA_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Copyright © 1998, Intel Corporation. All rights reserved.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_UBOA_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

requestCredential (input) - This is a Signed Manifest with embedded attributes that carry the details of the requested update. The required syntax of the Signed Manifest is described in the Manifest Syntax section below. The key used to sign the request credential must be the private key corresponding to the public key in the platform's configured Boot Object Authorization Certificate. Authority to update parameters in the Boot Object Authorization set cannot be delegated.

If there is no Boot Object Authorization Certificate, the request credential may be signed with any private key. In this case, this function interacts with the user in a platform-specific way to determine whether the operation should succeed.

newUpdateToken (output) - The function writes an allocated BIS_DATA_PTR containing the new unique update token value. The caller must eventually free the memory allocated by this function using the function BIS_Free.

See Also

BIS_GetBootObjectAuthorizationUpdateToken,
BIS_Free

Manifest Syntax

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer's information file, and a signature block file. These three parts, along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses should appear exactly as shown. Also note that manifest files and signer's information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single "space" character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Manifest File Example

The manifest file must include a section referring to a memory-type data object with the reserved name as shown in the example below. This data object is a zero-length object whose sole purpose in the manifest is to serve as a named collection point for the attributes that carry the details of the requested update. The attributes are also contained in the manifest file. An example manifest file is shown below.

Copyright © 1998, Intel Corporation. All rights reserved.

```
Manifest-Version: 2.0
ManifestPersistentId: (base64 representation of a unique GUID)

Name: memory:UpdateRequestParameters
Digest-Algorithms: SHA-1
SHA-1-Digest: (base64 representation of a SHA-1 digest of zero-length
 buffer)
X-Intel-BIS-ParameterSet: (base64 representation of
 BootObjectAuthorizationSetGUID)
X-Intel-BIS-ParameterSetToken: (base64 representation of the current
 update token)
X-Intel-BIS-ParameterId: (base64 representation of
 "BootObjectAuthorizationCertificate" or
 "BootAuthorizationCheckFlag")
X-Intel-BIS-ParameterValue: (base64 representation of
 certificate or
 single-byte boolean flag)
```

A line-by-line description of this manifest file is as follows.

`Manifest-Version: 2.0`

This is a standard header line that all signed manifests have. It must appear exactly as shown.

`ManifestPersistentId: (base64 representation of a unique GUID)`

The left-hand string should appear exactly as shown. The right-hand string should be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base64 encoded. Base64 is a simple encoding scheme for representing binary values that uses only printing characters. Base64 encoding is described in [BASE64].

`Name: memory:UpdateRequestParameters`

This identifies the manifest section that carries a dummy zero-length data object serving as the collection point for the attribute values appearing later in this manifest section (lines prefixed with "X-Intel-BIS-"). The string "memory:UpdateRequestParameters" must appear exactly as shown. This string can be found in manifest_defs.h as the preprocessor symbol UPDATE_PARMS_SECTION_NAME.

`Digest-Algorithms: SHA-1`

This enumerates the digest algorithms for which integrity data is included for the data object. These are required even though the data object is zero-length. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm should be "SHA-1". For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm should be "MD5". Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

```
SHA-1-Digest: (base64 representation of a SHA-1 digest of zero-length
 buffer)
```

Gives the corresponding digest value for the dummy zero-length data object. The value is base64 encoded. Note that for SHA-1, the digest value for a zero-length data object is not zero.

```
X-Intel-BIS-ParameterSet: (base64 representation of
 BootObjectAuthorizationSetGUID)
```

A named attribute value that distinguishes updates of BIS parameters from updates of other parameters. The left-hand attribute-name keyword is found in manifest_defs.h under the preprocessor symbol PARMSET_ATTR_NAME. The GUID value for the right-hand side is always the same. This GUID value is found in manifest_defs.h under the preprocessor symbol

Copyright © 1998, Intel Corporation. All rights reserved.

BOOT_OBJECT_AUTHORIZATION_PARMSET_GUIDVALUE. The representation inserted into the manifest is base64 encoded.

Note the "X-Intel-BIS-" prefix on this and the following attributes. The "X-" part of the prefix was chosen to avoid collisions with future reserved keywords defined by future versions of the signed manifest specification. The "Intel-BIS-" part of the prefix was chosen to avoid collisions with other user-define attribute names within the user-defined attribute name space.

```
X-Intel-BIS-ParameterSetToken: (base64 representation of the current
 update token)
```

A named attribute value that makes this update of BIS parameters different from any other on the same target platform. The left-hand attribute-name keyword is found in manifest_defs.h under the preprocessor symbol UPDATETOKEN_ATTR_NAME. The value for the right-hand side is generally different for each update-request manifest generated. The value to be base64 encoded is retrieved through the functions BIS_GetBootObjectAuthorizationUpdateToken or BIS_UpdateBootObjectAuthorization.

```
X-Intel-BIS-ParameterId: (base64 representation of
 "BootObjectAuthorizationCertificate" or
 "BootAuthorizationCheckFlag")
```

A named attribute value that indicates which BIS parameter is to be updated. The left-hand attribute-name keyword is found in manifest_defs.h under the preprocessor symbol PARMID_ATTR_NAME. The value for the right-hand side is the base64 encoded representation of one of the two strings shown. These strings are found in manifest_defs.h under the preprocessor symbols BOAC_PARMID (for the certificate) and BOACF_PARMID (for the check flag).

```
X-Intel-BIS-ParameterValue: (base64 representation of
 certificate or
 single-byte boolean flag)
```

A named attribute value that indicates the new value to be set for the indicated parameter. The left-hand attribute-name keyword is found in manifest_defs.h under the preprocessor symbol PARMVALUE_ATTR_NAME. The value for the right-hand side is the appropriate base64 encoded new value to be set. In the case of the Boot Object Authorization Certificate, the value is the new digital certificate raw data. A zero-length value removes the certificate altogether. In the case of the Boot Authorization Check Flag, the value is a single byte boolean value, where a non-zero value "turns on" the check and a zero value "turns off" the check.

Signer's Information File Example

The signer's information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer's information file carries the integrity data for the attributes in the corresponding section in the manifest file. An example signer's information file is shown below.

```
Signature-Version: 2.0
SignerInformationPersistentId: (base64 representation of a unique
 GUID)
SignerInformationName: BIS_UpdateManifestSignerInfoName

Name: memory:UpdateRequestParameters
Digest-Algorithms: SHA-1
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 corresponding manifest section)
```

A line-by-line description of this signer's information file is as follows.

```
Signature-Version: 2.0
```


Copyright © 1998, Intel Corporation. All rights reserved.

This is a standard header line that all signed manifests have. It must appear exactly as shown.

`SignerInformationPersistentId: (base64 representation of a unique GUID)`

The left-hand string should appear exactly as shown. The right-hand string should be a unique GUID for every signer's information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base64 encoded. Base64 is a simple encoding scheme for representing binary values that uses only printing characters. Base64 encoding is described in [BASE64].

`SignerInformationName: BIS_UpdateManifestSignerInfoName`

The left-hand string should appear exactly as shown. The right-hand string should appear exactly as shown. This string is defined in manifest_defs.h under the preprocessor symbol UPDATE_MANIFEST_SIGINFO_NAME.

`Name: memory:UpdateRequestParameters`

This identifies the section in the signer's information file corresponding to the section with the same name in the manifest file described earlier. The string "memory:UpdateRequestParameters" must appear exactly as shown. This string can be found in manifest_defs.h under the preprocessor symbol UPDATE_PARMS_SECTION_NAME.

`Digest-Algorithms: SHA-1`

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

`SHA-1-Digest: (base64 representation of a SHA-1 digest of the corresponding manifest section)`

Gives the corresponding digest value for the corresponding manifest section. The value is base64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening "Name:" keyword and continues up to, but not including, the next section's "Name:" keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next "Name:" keyword or end-of-file.

Signature Block File Example

A signature block file is a raw binary file (not base64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer's information file. There must be a correspondence between the name of the signer's information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer's information file name of "myinfo.SF", the corresponding DSA signature block file name would be "myinfo.DSA".

The format of a signature block file is defined in [PKCS].

Copyright © 1998, Intel Corporation. All rights reserved.

3.10 General-purpose security functions

BIS supplies a general-purpose function that a preboot application can use to build a customized authentication model that differs from the authentication model embodied by the BIS_VerifyBootObject function and its closely related functions. In particular this function, BIS_VerifyObjectWithCredential, allows the signing authority to be different than the Boot Object Authorization Certificate implicitly used by BIS_VerifyBootObject. This function is also limited to pure digital verification, that is, it does not involve user interaction under any circumstances.

This function is typically used by a first-stage Network Bootstrap Program in verifying the next stage downloaded. The first stage would include an embedded authority certificate from the software vendor. This authority certificate would be supplied as the required signature authority when verifying the second stage. The second stage credentials would have to be signed by the vendor's corresponding private key.

The general-purpose security function is described in detail in the next section.

Copyright © 1998, Intel Corporation. All rights reserved.

3.10.1 BIS_VerifyObjectWithCredential

```
typedef
struct _BIS_VerifyObjectWithCredential_PARMS
{
    UINT32        sizeOfStruct;   //[in]  Byte length of this structure.
    BIS_STATUS    returnValue;    //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE  appHandle;    //[in]  From BIS_Initialize( ).
    BIS_DATA      credentials;    //[in]  Verification signed manifest.
    BIS_DATA      dataObject;     //[in]  Data object to verify.
    BIS_DATA      sectionName;    //[in]  Name of credential section to use.
    BIS_DATA      authorityCertificate;   //[in]  Certificate for
                                          //credentials.
    BIS_BOOLEAN   isVerified;     //[out] Result of verification.
}
BIS_VOWC_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_VOWC_PARMS          params;
BIS_APPLICATION_HANDLE  appHandle;
BIS_DATA                credentials;
BIS_DATA                dataObject;
BIS_DATA                sectionName;
BIS_DATA                authorityCertificate;
. . .
// Set "in" parameter values
params.sizeOfStruct         = sizeof(params);
params.appHandle            = appHandle;
params.credentials          = credentials;
params.dataObject           = dataObject;
params.sectionName          = sectionName;
params.authorityCertificate = authorityCertificate;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_VerifyObjectWithCredential,  // opCode
    (void *) & params,                 // pParamBundle
    BIS_TRUE);                         // checkFlag
```

The function selected with the BISOP_VerifyObjectWithCredential operation code verifies the integrity and authorization of the indicated data object according to the indicated credentials and authority certificate.

Both an integrity check and an authorization check are performed. The rules for a successful integrity check are:

- Verify the credentials - The credentials parameter is a valid Signed Manifest, with a single signer. The signer's identity is included in the credential as a certificate.
- Verify the data object - The Manifest must contain a section with the name as specified by the sectionName parameter, with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the data specified by the dataObject parameter of this function.

The authorization check is optional. It is performed only if the authorityCertificate.data parameter is other than BIS_NULL. If it is other than BIS_NULL, the rules for a successful authorization check are:

- The authorityCertificate parameter is a valid digital certificate. There is no requirement regarding the signer (issuer) of this certificate.

Copyright © 1998, Intel Corporation. All rights reserved.

- The public key certified by the signer's certificate must match the public key in the authorityCertificate. The match must be direct, that is, the signature authority cannot be delegated along a certificate chain.

If all of the integrity and authorization check rules are met, the function returns with returnValue zero isVerified is BIS_TRUE. Otherwise, it returns with returnValue as a non-zero specific error code and isVerified is BIS_FALSE.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_VerifyObjectWithCredential.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_VOWC_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_VOWC_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

credentials (input) - A Signed Manifest containing verification information for the indicated data object. The Manifest signature itself must meet the requirements described above. The required syntax of the Signed Manifest is described in the Manifest Syntax section below.

dataObject (input) - An in-memory copy of the raw data object to be verified.

sectionName (input) - A string giving the section name in the manifest holding the verification information (in other words, hash value) that corresponds to dataObject.

authorityCertificate (input) - A digital certificate whose public key must match the signer's public key which is found in the credentials. This parameter is optional (authorityCertificate.data may be BIS_NULL).

isVerified (output) - The function writes BIS_TRUE if the verification was successful. Otherwise, the function writes BIS_FALSE.

Manifest Syntax

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer's information file, and a signature block file. These three parts along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses should appear exactly as shown. Also note that manifest files and signer's information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single "space" character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Copyright © 1998, Intel Corporation. All rights reserved.

Manifest File Example

The manifest file must include a section referring to a memory-type data object with the reserved name as shown in the example below. This data object is the Data Object to be verified. An example manifest file is shown below.

```
Manifest-Version: 2.0
ManifestPersistentId: (base64 representation of a unique GUID)

Name: (a memory-type data object name)
Digest-Algorithms: SHA-1
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 data object)
```

A line-by-line description of this manifest file is as follows.

`Manifest-Version: 2.0`

This is a standard header line that all signed manifests have. It must appear exactly as shown.

`ManifestPersistentId: (base64 representation of a unique GUID)`

The left-hand string should appear exactly as shown. The right-hand string should be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base64 encoded. Base64 is a simple encoding scheme for representing binary values that uses only printing characters. Base64 encoding is described in [BASE64].

`Name: (a memory-type data object name)`

This identifies the section that carries the integrity data for the target Data Object. The right-hand string must obey the syntax for memory-type references, that is, it is of the form “memory:SomeUniqueName”. The “memory:” part of this string must appear exactly. The “SomeUniqueName” part is chosen by the caller. It must be unique within the section names in this manifest file. The entire “memory:SomeUniqueName” string must match exactly the corresponding string in the signer’s information file described below. Furthermore, this entire string must match the value given for the SectionName parameter to this function. Note that the target Data Object cannot be found directly from this manifest. A caller verifying the Data Object integrity must load the Data Object into memory and specify its memory location explicitly to this verification function through the DataObject parameter.

`Digest-Algorithms: SHA-1`

This enumerates the digest algorithms for which integrity data is included for the data object. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm should be “SHA-1”. For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm should be “MD5”. Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

```
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 data object)
```

Gives the corresponding digest value for the data object. The value is base64 encoded.

Signer’s Information File Example

The signer’s information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer’s information file carries the integrity data for the corresponding section in the manifest file. An example signer’s information file is shown below.

```
Signature-Version: 2.0
SignerInformationPersistentId: (base64 representation of a
 unique GUID)
```


Copyright © 1998, Intel Corporation. All rights reserved.

```
SignerInformationName: BIS_VerifiableObjectSignerInfoName

Name: (a memory-type data object name)
Digest-Algorithms: SHA-1
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 corresponding manifest section)
```

A line-by-line description of this signer's information file is as follows.

`Signature-Version: 2.0`

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
SignerInformationPersistentId: (base64 representation of a
 unique GUID)
```

The left-hand string should appear exactly as shown. The right-hand string should be a unique GUID for every signer's information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base64 encoded. Base64 is a simple encoding scheme for representing binary values that uses only printing characters. Base64 encoding is described in [BASE64].

`SignerInformationName: BIS_VerifiableObjectSignerInfoName`

The left-hand string should appear exactly as shown. The right-hand string should appear exactly as shown. This string is defined in manifest_defs.h under the preprocessor symbol VERIFIABLE_OBJECT_SIGINFO_NAME.

`Name: (a memory-type data object name)`

This identifies the section in the signer's information file corresponding to the section with the same name in the manifest file described earlier. The right-hand string must match exactly the corresponding string in the manifest file described above.

`Digest-Algorithms: SHA-1`

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

```
SHA-1-Digest: (base64 representation of a SHA-1 digest of the
 corresponding manifest section)
```

Gives the corresponding digest value for the corresponding manifest section. The value is base64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening “Name:” keyword and continues up to, but not including, the next section's “Name:” keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next “Name:” keyword or end-of-file.

Copyright © 1998, Intel Corporation. All rights reserved.

Signature Block File Example

A signature block file is a raw binary file (not base64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer's information file. There must be a correspondence between the name of the signer's information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer's information file name of "myinfo.SF", the corresponding DSA signature block file name would be "myinfo.DSA".

The format of a signature block file is defined in [PKCS].

3.11 Initialization, Shutdown, and Utility functions

This group of functions consists of miscellaneous support functions used for initialization, shutdown, information, and freeing memory allocated and returned by other functions. The functions in this group are:

BIS_Initialize - Initializes the BIS service, checking that it is compatible with the version requested by the caller. After this call, other BIS functions may be invoked.

BIS_Shutdown - Shuts down the BIS service. After this call, other BIS functions may no longer be invoked.

BIS_Free - Deallocates a BIS_DATA_PTR and associated memory allocated by one of the other BIS functions.

BIS_GetSignatureInfo - Retrieves a list of digital signature algorithm, hash algorithm, and key-lengths that the platform supports.

The functions in this group are described in detail in the next sections.

Copyright © 1998, Intel Corporation. All rights reserved.

3.11.1 BIS_Initialize

```
typedef
struct _BIS_Initialize_PARMS
{
    UINT32                  sizeOfStruct;  //[in]  Byte length of this
                                           //struct.
    BIS_STATUS              returnValue;   //[out] BIS_OK | error code.
    BIS_VERSION             interfaceVersion; //[in/out] version
                                              //needed/available.
    BIS_APPLICATION_HANDLE  appHandle;     //[out] Application handle.
    BIS_DATA                targetAddress  //[in] Address of BIS platform.
}
BIS_INIT_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_INIT_PARMS          params;
BIS_VERSION             interfaceVersion;
. . .
// Set "in" parameter values
params.sizeOfStruct       = sizeof(params);
params.interfaceVersion   = interfaceVersion;
params.targetAddress.data = BIS_NULL;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_Initialize,    // opCode
    (void *) & params,   // pParamBundle
    BIS_TRUE);           // checkFlag
```

The function selected with the BISOP_Initialize operation code initializes the BIS service, checking that it is compatible with the version requested by the caller. After this call, other BIS functions may be invoked.

This function must be the first BIS function invoked by an application. It passes back a handle value that must be used in subsequent BIS functions. The handle must be eventually destroyed by a call to the BIS_Shutdown function, thus ending that handle's lifetime. After the handle is destroyed, BIS functions may no longer be called with that handle value. Thus all other BIS functions may only be called between a pair of BIS_Initialize and BIS_Shutdown functions.

There is no penalty for calling BIS_Initialize multiple times. Each call passes back a distinct handle value. Each distinct handle must be destroyed by a distinct call to BIS_Shutdown. The lifetimes of handles created and destroyed with these functions may be overlapped in any way.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_Initialize.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_INIT_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Copyright © 1998, Intel Corporation. All rights reserved.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_INIT_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

interfaceVersion (input, output) - On input, the caller supplies the major version number of the interface version desired. The minor version number supplied on input is ignored since interface compatibility is determined solely by the major version number. On output, both the major and minor version numbers are updated with the major and minor version numbers of the interface (and underlying implementation). This update is done whether or not the initialization was successful.

appHandle (output) - The function writes the new BIS_APPLICATION_HANDLE if successful, otherwise it writes BIS_NULL. The caller must eventually destroy this handle by calling BIS_Shutdown.

targetAddress (input) - Indicates a network or device address of the BIS platform to connect to. Local-platform BIS implementations require that the caller sets targetAddress.data to BIS_NULL, but otherwise ignores this parameter. BIS Redirector implementations must define their own format and interpretation of this parameter outside the scope of this document. For all implementations, if the targetAddress is an unsupported value, the function fails with the error BIS_NOT_IMPLEMENTED.

See Also

BIS_Shutdown

Copyright © 1998, Intel Corporation. All rights reserved.

3.11.2 BIS_Shutdown

```
typedef
struct _BIS_Shutdown_PARMS
{
    UINT32                  sizeOfStruct; //[in]  Byte length of this
                                          //structure.
    BIS_STATUS              returnValue;  //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE  appHandle;    //[in]  From BIS_Initialize( ).
}
BIS_SHUTDOWN_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_SHUTDOWN_PARMS      params;
BIS_APPLICATION_HANDLE  appHandle;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_Shutdown,        // opCode
    (void *) & params,     // pParamBundle
    BIS_TRUE);             // checkFlag
```

The function selected with the BISOP_Shutdown operation code shuts down the BIS service, invalidating the application handle. After this call, other BIS functions may no longer be invoked using the application handle value.

This function must be paired with a preceding successful call to the BIS_Initialize function. The lifetime of an application handle extends from the time the handle was returned from BIS_Initialize until the time the handle is passed to BIS_Shutdown. If there are other remaining handles whose lifetime is still active, they may still be used in calling BIS functions.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_Shutdown.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_SHUTDOWN_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_SHUTDOWN_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

Copyright © 1998, Intel Corporation. All rights reserved.

See Also

BIS_Initialize

Copyright © 1998, Intel Corporation. All rights reserved.

3.11.3 BIS_Free

```
typedef
struct _BIS_Free_PARMS
{
    UINT32                  sizeOfStruct;   //[in]  Byte length of this
                                            //structure.
    BIS_STATUS              returnValue;    //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE  appHandle;      //[in] From BIS_Initialize( ).
    BIS_DATA_PTR            toFree;         //[in]  BIS_DATA being freed.
}
BIS_FREE_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_FREE_PARMS          params;
BIS_APPLICATION_HANDLE  appHandle;
BIS_DATA_PTR            toFree;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
params.toFree       = toFree;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_Free,           // opCode
    (void *) & params,    // pParamBundle
    BIS_TRUE);            // checkFlag
```

The function selected with the BISOP_Free operation code deallocates a BIS_DATA_PTR and associated memory allocated by one of the other BIS functions.

Callers of other BIS functions that allocate memory in the form of a BIS_DATA_PTR must eventually call this function to deallocate the memory before calling the BIS_Shutdown function for the application handle under which the memory was allocated. Failure to do so causes unspecified results, and the continued correct operation of the BIS service cannot be guaranteed.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_Free.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_FREE_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_FREE_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

Copyright © 1998, Intel Corporation. All rights reserved.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

toFree (input) - A BIS_DATA_PTR and associated memory block to be freed. This BIS_DATA_PTR must have been allocated by one of the other BIS functions.

See Also

Functions that allocate BIS_DATA_PTR:
BIS_GetBootObjectAuthorizationCertificate
BIS_GetBootObjectAuthorizationUpdateToken
BIS_UpdateBootObjectAuthorization
BIS_GetSignatureInfo

Copyright © 1998, Intel Corporation. All rights reserved.

3.11.4 BIS_GetSignatureInfo

```
typedef
struct _BIS_GetSignatureInfo_PARMS
{
    UINT32                   sizeOfStruct;   //[in]  Byte length of this
                                             //structure.
    BIS_STATUS               returnValue;    //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE   appHandle;      //[in]  From BIS_Initialize( ).
    BIS_DATA_PTR             signatureInfo;  //[out] Signature info struct.
}
BIS_GSI_PARMS;
```

Calling Example:

```
UINT8                   integrityOk;
BIS_ENTRY_POINT         entryStructure;
BIS_GSI_PARMS           params;
BIS_APPLICATION_HANDLE  appHandle;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_GetSignatureInfo,   // opCode
    (void *) & params,        // pParamBundle
    BIS_TRUE);                // checkFlag
```

The function selected with the BISOP_GetSignatureInfo operation code retrieves a list of digital signature algorithm, hash algorithm, and key-length combinations that the platform supports. The list is an array of (algorithm id, key length) pairs, where the algorithm id represents the combination of signature algorithm and hash algorithm, and the key length is expressed in bits. The number of array elements can be computed using the Length field of the retrieved BIS_DATA_PTR.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_GetSignatureInfo.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_GSI_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_GSI_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

Copyright © 1998, Intel Corporation. All rights reserved.

signatureInfo (output) - The function writes an allocated BIS_DATA_PTR containing the array of BIS_SIGNATURE_INFO structures representing the supported algorithm and key length combinations. The caller must eventually free the memory allocated by this function using the function BIS_Free.

See Also

BIS_Free

Copyright © 1998, Intel Corporation. All rights reserved.