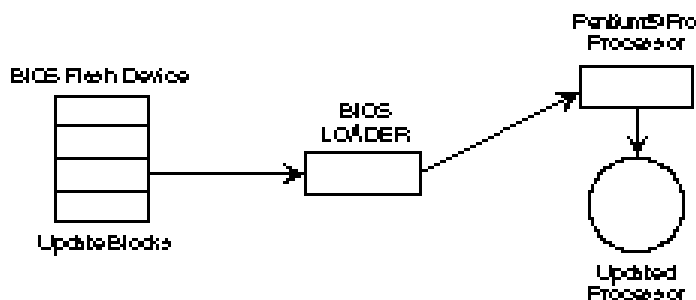
1
Pentium® Pro Processor BIOS Update Feature

The Pentium® Pro processor has the capability to correct specific errata through the loading of an Intel-supplied data block. This chapter describes the underlying mechanisms the BIOS needs in order to utilize this feature during system initialization. It also describes a specification that provides for incorporating future releases of the update into a system BIOS.
Intel considers the combination of a particular silicon revision and BIOS update as the equivalent stepping of the processor. Intel completes a full-stepping level validation and testing for new releases of BIOS updates.
An update loader integrated within the BIOS uses data from the update to correct specific errata. The BIOS is responsible for loading the update on all processors during system initialization, as shown in Figure 8 -1.

Figure 8-1. Integrating Processor Specific Updates



1.1 BIOS Update

A BIOS Update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. This section describes the update and the structure of its data format.
Each BIOS Update is tailored for a particular stepping of the Pentium Pro processor. The data within the update is encrypted by Intel and is designed such that it is rejected by any stepping of the processor other than its intended recipient. Thus, a given BIOS Update is associated with a particular family, model, and stepping of the processor as returned by the CPUID instruction. The encryption scheme also guards against tampering of the update data and provides a means for determining the authenticity of any given BIOS update.
The BIOS Update is a data block that is exactly 2048 bytes in length. The initial 48 bytes of the update contain a header with information used to maintain the update. The update header and its reserved fields are interpreted by software based upon the Header Version. The initial version of the header is `00000001h`. Figure 8 -2 shows the format of the BIOS Update data block, and Table 8 -1 explains each of the fields.

Figure 8-2. BIOS Update Data Block

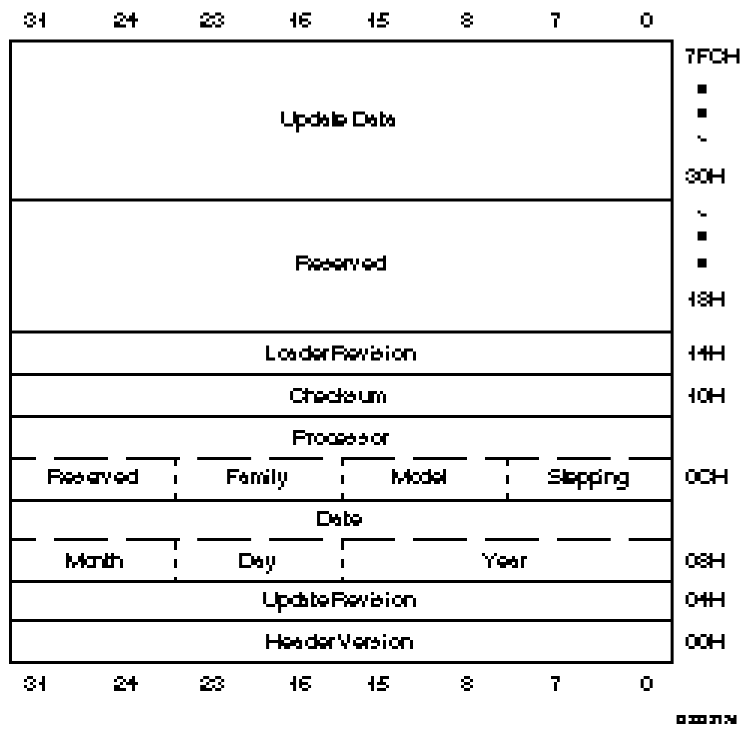

31 24 23 16 15 8 7 0
7FCH
Update Data
30H
Reserved
18H
Loader Revision
14H
Checksum
10H
Processor
Reserved
Family
Model
Stepping
0CH
Date
Month
Day
Year
08H
Update Revision
04H
Header Version
00H
31 24 23 16 15 8 7 0

Table 8-1. BIOS Update Header Data

Field Name	Offset (in bytes)	Length (in bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that it is loaded successfully by the processor. The value in this field cannot be used for processor stepping identification alone.
Date	8	4	Date of the update creation in binary format: mmddyyyy, month, day year (e.g., 07/18/95 as `07181995h`).
Processor	12	4	Family, model, and stepping of processor that requires this particular update revision (e.g., `00000611h`). Each BIOS update is designed specifically for a given family, model, and stepping of the processor. The BIOS uses the Processor field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when summation of the 512 double words of the update results in the value zero.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is `00000001h`.
Reserved	24	24	Reserved Fields for future expansion.
Update Data	48	2000	Update data.

1.2Update Loader

This section describes the update loader used to load an update into the Pentium Pro processor. It also discusses the requirements placed upon the BIOS to ensure proper loading of an update.
The update loader contains the minimal instructions needed to load an update. The specific instruction sequence required to load an update is associated with the Loader Revision field contained within the update header. The revision of the update loader is expected to change very infrequently, potentially only when new processor models are introduced.

The code below represents the update loader with a Loader Revision of `00000001h`:

```
mov    ecx,79h             ; 79H in ECX
xor    eax,eax
xor    ebx,ebx
mov    ax,cs               ; Segment of BIOS Update
shl    eax,4
mov    bx,offset Update    ; Offset of BIOS Update
add    eax,ebx             ; Linear Address of Update in EAX
add    eax,48d             ; Offset of the Update Data within the Update
xor    edx,edx             ; Zero in EDX
WRMSR                      ; BIOS Update Trigger
```

1.2.1Update Loading Procedure

The simple loader previously described assumes that Update is the address of a BIOS update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory that is accessible by the processor within its current operating mode (real, protected).

Before the BIOS executes the Update trigger (WRMSR) instruction the following must be true:

- EAX contains the linear address of the start of the Update Data
- EDX contains zero
- ECX contains `79h`

Other requirements to keep in mind are:

- If the update is loaded while the processor is in real mode, then the update data may not cross a segment boundary.
- If the update is loaded while the processor is in real mode, then the update data may not exceed a segment limit.
- If paging is enabled, then the update data must be mapped by pages that are currently present.
- The BIOS update data does not require any particular byte or word boundary alignment.

Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the BIOS update must be reloaded on all processors that observed the reset. The effects of a loaded update are persistent across a processor INIT. No ill effects are caused by loading an update into a processor multiple times.

Timing of Update Loading

There are no specific timing requirements as to when the BIOS must load the update into the processor. It is not necessary to load the update at the time of initial power-on reset or very early during the POST stage. Intel guarantees that the processor successfully executes through POST without having the update loaded. Intel recommends that BIOS vendors provide a setup option to enable/disable update loading. This recommendation implies that update loading must occur after setup.

Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID. The BIOS is responsible for ensuring that this requirement is met, and that the loader is located in a module that is executed by all processors in the system. If a system design permits multiple steppings of Pentium Pro processors to exist concurrently, then the BIOS must verify each individual CPUID against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

Sample Loader

The actual loader implementation in a BIOS contains the basic loader presented in this section and additional supporting code. A sample implementation is supplied in the Pentium Pro processor reference

source code kit. The sample loader in the source code kit verifies the CPUID values of various processors in the system and loads a correct update for each processor stepping that is found in the system.

1.2.2 Update Loader Enhancements

The update loader presented in the previous section is a minimal implementation that can be enhanced to provide additional functionality and features. Some potential enhancements are described below:

- The BIOS can incorporate multiple updates to support multiple steppings of the Pentium Pro processor. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID of the processor that it is running on against the available headers before loading a particular Update.
- A loader can load the update and test the processor to determine if the update was loaded correctly. This can be done as described in the Update Signature and Verification section in this chapter.
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero, and can reject the update.
- A loader can provide sign on messages indicating successful loading of an update.

1.3 Update Signature and Verification

The Pentium Pro processor provides capabilities to verify the authenticity of a particular update and to identify the Update Revision of the currently functioning revision. This section describes these model-specific extensions of the processor that support this feature. The update verification method below assumes that the BIOS only verifies an update that is more recent than the revision currently loaded into the processor.

The CPUID instruction has been enhanced to return a value in a model specific register in addition to its usual register return values. The semantics of the CPUID instruction are not modified except to cause it to deposit an update ID value in the 64-bit model-specific register (MSR) at address `08Bh`. If no update is present in the processor, the value in the MSR remains unmodified. Normally a zero value is preloaded into the MSR before executing the CPUID instruction. If the MSR still contains zero after executing CPUID, this indicates that no update is present.

The Update ID value returned in the EDX register after a RDMSR instruction indicates the revision of the update loaded in the processor. This value, in combination with the normal CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the Update Revision field in the BIOS Update header for verification of the correct BIOS update load. No consecutive updates released for a given stepping of the Pentium Pro processor may share the same signature. Updates for different steppings are differentiated by the CPUID value.

1.3.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the Update Revision of the currently functioning revision. This signature is available any time after the actual update has been loaded, and requesting this signature does not have any negative impact upon any currently loaded update. The procedure for determining this signature is:

```
1. mov ecx, 08Bh       ;Model Specific Register to Read
2. xor eax,eax
3. xor edx,edx
4. WRMSR               ;Load 0 to MSR at 8Bh
5. mov eax,1
6. CPUID
7. mov ecx, 08BH       ;Model Specific Register to Read
8. RDMSR               ;Read Model Specific Register
```

If there is an Update currently active in the processor, its update revision is returned in the EDX register after the RDMSR instruction has completed.

1.3.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, with the following algorithm:

```
Z = Obtain Update Revision from the Update Header to be authenticated;
X = Obtain Current Update Signature from MSR 8Bh;
If (Z > X) Then
    Load Update that is to be authenticated;
    Y = Obtain New Signature from MSR 8Bh;
    If (Z == Y) then Success
    Else Fail
Else Fail
```

The algorithm requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). This authentication procedure relies upon the decryption provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

1.4 Pentium® Pro Processor BIOS Update Specifications

This section describes two interfaces that an application can use to dynamically integrate processor-specific updates into the system BIOS. In this discussion, the application is referred to as the *calling program* or *caller*.

Both the real mode INT 15h call and the alternate protected mode call specifications described here are Intel extensions to an OEM BIOS. These extensions allow an application to read and modify the contents of the BIOS update data in NVRAM. The BIOS update loader, which is part of the system BIOS, cannot be updated by either of the two interfaces. All of the functions defined in either of the two specifications must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode. The protected mode specification provides an interface that is available in either real or protected mode.

1.4.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15h, AX=0D042h, BL=0h) it must implement all of the subfunctions defined in the INT 15h, AX= 0D042h specification. If a BIOS implements the protected mode interface signature and associated parameters, it must implement all of the subfunctions defined in the protected mode interface specification. There are no optional functions. The BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value `0FFFFFFFFh` indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a 2048 byte region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit is referred to as an *update block*. The BIOS for a single processor system need only provide one update block to store the BIOS update data. The BIOS for a multiple processor capable system needs to provide one update block for each unique processor stepping supported by the OEM's system. The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing update blocks that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis. As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

The following algorithm describes the steps performed during BIOS initialization used to load the updates into the processor(s). It assumes that the BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS and that the

update block contains a correct checksum. It also assumes that the BIOS ensures that at most one update exists for each processor stepping and that older update revisions are not allowed to overwrite more recent ones. These requirements are checked by the BIOS during the execution of the write update function of this interface.

1. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor field in the header matches the family, model, and stepping of the current processor.
2. For each processor in the system {

```
Determine Family, Model and Stepping via CPUID instruction for Genuine Intel processors;
for (i=Update Block 0, i< Update Num; i++) {
    If (UpdateHeader.Processor == Family, Model and Stepping of this processor) {
            Load the Update into the Processor from UpdateHeader.UpdateData;
            /* optionally verify that update was correctly loaded into the processor */
            /* Go on to next processor */
            Break;
            }
    /* If no match is found, then the current processor does not require an update OR */
    /* an update for that processor has not been loaded into NVRAM */
}
```

When performing either the INT 15h, 0D042h functions or protected mode interface functions, the BIOS must assume that the caller has no knowledge about platform specific requirements. It is the responsibility of the BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data via the Write Update subfunction, the BIOS must maintain implementation specific data requirements, such as the update of NVRAM checksum. The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

The BIOS must implement a mechanism whereby the update loading at initialization time can be disabled without destroying the update. Possible ways of implementing this function are either through CMOS bits or through bits located in the NVRAM device. Regardless of which mechanism allows for the enabling and disabling, it is recommended that the user be provided a method for determining the state of the update, such as via the BIOS setup routine.

1.4.2 Responsibilities of the Calling Program

This section of the document lists the responsibilities of the calling program using either of the two interface specifications to load BIOS update(s) into BIOS NVRAM.

The calling program must call the INT 15h, 0D042h functions from a pure real mode program and must be executing on a system that is running in pure real mode. The caller may call the protected mode interface from either pure real mode or protected mode. The caller must issue the Presence Test function (sub function 0) and verify the signature and return codes of that function. It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.

The calling program must read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS refuses to overwrite a newer update with an older version. The Update Header contains information about version and processor specifics for the calling program to make an intelligent decision about load/noload.

There can be no ambiguous updates. The BIOS refuses to allow multiple updates for the same CPUID to exist at the same time. The BIOS also refuses to load an update for a processor that does not exist in the system.

The calling application must implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written. The following pseudo-code represents a calling program.

INT 15 D042 Calling Program Pseudo-code

```
//
//  We must be in real mode
//
If the system is not in Real mode
then Exit
//
//  Detect the presence of Genuine Intel processor(s) that can be updated (Family,Model)
//
If no Intel processors exist that can be updated
    then Exit
//
//  Detect the presence of the Intel BIOS Update Extensions
//
If the BIOS fails the PresenceTest
then Exit
//
// If the APIC is enabled, see if any other processors are out there
//
Read APICBaseMSR
If APIC enabled {
    Send Broadcast Message to all processors except self via APIC;
    Have all processors execute CPUID and record Family, Model, Stepping
    If all processors are not updatable
        then Exit
    }
//
// Determine the number of unique update slots needed for this system
//
NumSlots = 0;
For each processor {
    If ((this is a unique processor stepping) and
    (we have an update in the database for this processor)) {
        Checksum the update from the database;
        If Checksum fails
             then Exit;
        Increment NumSlots;
        }
    }
//
// Do we have enough update slots for all CPUs?
//
If there are more unique processor steppings that update slots provided by the BIOS
    then Exit

//
// Do we need any update slots at all?  If not, then we're all done
//
If (NumSlots == 0)
    then Exit

//
//  Record updates for processors in NVRAM.
//
For (I=0; i<NumSlots; I++) {
        //
```

```
        //   Load each Update
        //
    Issue the WriteUpdate function

    If (STORAGE_FULL) returned {
        Display Error -- BIOS is not managing NVRAM appropriately
        exit
        }
    If (INVALID_REVISION) returned {
        Display Message: More recent update already loaded in NVRAM for this stepping
        continue;
        }

    If any other error returned {
        Display Diagnostic
        exit
        }
        //
        //   Verify the update was loaded correctly
        //
    Issue the ReadUpdate function

    If an error occurred {
        Display Diagnostic
        exit
        }
        //
        //   Compare the Update read to that written
        //
        if (Update read != Update written) {
        Display Diagnostic
        exit
        }
    }
//
//  Enable Update Loading, and inform user
//
Issue the ControlUpdate function with Task=Enable.
```

1.4.3BIOS Update Functions

Table 8 -2 defines the current Pentium Pro Processor BIOS Update Functions.

Table 8-2. BIOS Update Functions

BIOS Update Function	Function Number	Description	Required/Optional
Presence test	00h	Returns information about the supported functions.	Required
Write BIOS update data	01h	Writes one of the BIOS Update data areas (slots).	Required
BIOS update control	02h	Globally controls the loading of BIOS Updates.	Required
Read BIOS update data	03h	Reads one of the BIOS Update data areas (slots).	Required

1.4.4 INT 15h-based Interface

The application that implements the INT 15h-based interface is available on the reference BIOS diskette. The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, the functions return with Carry Cleared and AH contains the returned status. The general return codes and other constant definitions are listed later in this chapter.

The OEM Error (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, the OEM Error must be set to SUCCESS. The OEM Error field is undefined if AH contains either SUCCESS (00) or NOT_IMPLEMENTED (86h). In all other cases it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections give the details of the functions provided by the INT15h-based interface.

Function 00h – Presence Test

This function verifies that the BIOS has implemented the required BIOS update functions. Table 8 -3 lists the parameters and return codes for the function.

Table 8-3. Parameters for the Presence Test

Input:

Register	Name	Value
AX	Function Code	0D042h
BL	Subfunction	00h - Presence Test

Output:

Register	Name	Description
CF	Carry Flag	Carry Set - Failure - AH Contains Status. Carry Clear - All return values are valid.
AH	Return Code	
AL	OEM Error	Additional OEM Information.
EBX	Signature Part 1	'INTE' - Part one of the signature.
ECX	Signature Part 2	‘LPEP’- Part two of the signature.
EDX	Loader Version	Version number of the BIOS Update Loader.
SI	Update Cnt	Number of Update Blocks the system can record in NVRAM.

Return Codes:

Code	Description
SUCCESS	Function completed successfully.
NOT_IMPLEMENTED	Function not implemented.
	See Table 8 -8 for code definitions.

Description:

In order to assure that the BIOS function is present, the caller must verify the Carry Flag, the Return Code, and the 64-bit signature. Each update block is exactly 2048 bytes in length. UpdateCnt reflects the number of update blocks available for storage within non-volatile RAM. UpdateCnt must return with a value greater than or equal to the number of unique processor steppings currently installed within the system.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.   The source code for this loader program is provided to OEMs and BIOS vendors by Intel.

Function 01h – Write BIOS Update Data

This function integrates a new BIOS update into the BIOS storage device. Table 8 -4 lists the parameters and return codes for the function.

Table 8-4. Parameters for the Write Update Data Function

Input:

Register	Name	Value
AX	Function Code	0D042h
BL	Subfunction	01h - Write Update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length.
CX	Scratch Pad1	Real Mode Segment address of 64 kilobytes of RAM Block.
DX	Scratch Pad2	Real Mode Segment address of 64 kilobytes of RAM Block.
SI	Scratch Pad3	Read Mode Segment address of 64 kilobytes of RAM Block.

SS:SP	Stack pointer	32 kilobytes of Stack Minimum.

Output:

CF	Carry Flag	Carry Set - Failure - AH contains Status Carry Clear - All return values are valid.
AH	Return Code	Status of the Call.
AL	OEM Error	Additional OEM Information.

Table 8-10. Parameters for the Write Update Data Function (continued)

Return Codes:

SUCCESS	Function completed successfully.
WRITE_FAILURE	A failure because of the inability to write the storage device.
ERASE_FAILURE	A failure because of the inability to erase the storage device.
READ_FAILURE	A failure because of the inability to read the storage device.
STORAGE_FULL	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
INVALID_HEADER	The Update Header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	The Update does not checksum correctly.
SECURITY_FAILURE	The Update was rejected by the processor.
INVALID_REVISION	The same or more recent revision of the update exists in the storage device.
	See Table 8 -8 for code definitions.

Description:
The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.
Before writing the update block into NVRAM, the BIOS must ensure that the update structure meets the following criteria in the following order:

1. The Update header version must be equal to an Update header version recognized by the BIOS.
2. The Update loader version in the update header must be equal to the Update loader version contained within the BIOS image.
3. The Update block must checksum to zero. This checksum is computed as a 32-bit summation of all 512 double words in the structure, including the header.

The BIOS selects an update block in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

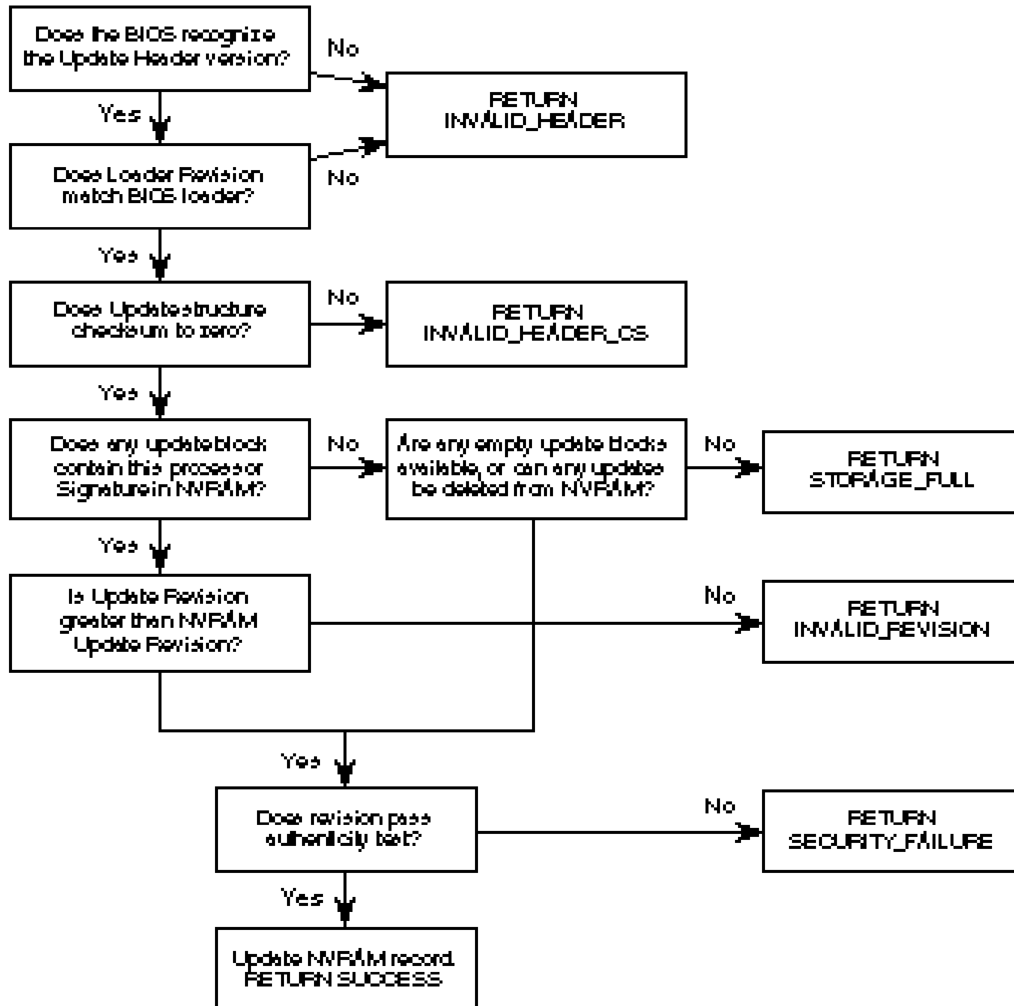
- The Processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM.
- The Update Revision in the proposed update must be greater than the Update Revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite an update block for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.
Finally, before storing the proposed update into NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in the previous section. This includes loading the update into the current processor, executing the CPUID instruction, reading MSR `08Bh`, and comparing a calculated value with the Update Revision in the proposed update header for equality.
When performing the Write Update function, the BIOS must record the entire update, including the header and the update data. When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not

overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping.

Figure 8-4 shows the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new BIOS update.

Figure 8-3. Write OperationFlow Chart



Function 02h – BIOS Update Control

This function enables loading of binary updates into the processor. Table 8 -5 lists the parameters and return codes for the function.

Table 8-5. Parameters for the Control Update Subfunction

Input:		
AX	Function Code	`0D042h`
BL	Subfunction	`02h` - Control Update
BH	Task	See Description.
CX	Scratch Pad1	Real Mode Segment of 64 kilobytes of RAM Block.
DX	Scratch Pad2	Real Mode Segment of 64 kilobytes of RAM Block.
SI	Scratch Pad3	Real Mode Segment of 64 kilobytes of RAM Block.
SS:SP	Stack pointer	32 kilobytes of Stack Minimum
Output:		
CF	Carry Flag	Carry Set - Failure - AH contains Status. Carry Clear - All return values are valid.
AH	Return Code	Status of the Call.
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either Enable or Disable indicator.

Return Codes:	
SUCCESS	Function completed successfully.
READ_FAILURE	A failure because of the inability to read the storage device.
	See Table 8 -8 for code definitions.

Description:

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk. The system BIOS maintains control for disabling the loading of BIOS updates separately via the SETUP or other BIOS dependent mechanism.

The caller specifies the requested operation by placing one of the values from Table 8 -6 in the BH register. After successfully completing this function the BL register contains either the Enable or the Disable designator. Note that if the function fails, the UpdateStatus return value is undefined.

Table 8-6. Mnemonic Values

Mnemonic	Value	Meaning
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot not occur.

Function 03h – Read BIOS Update Data

This function reads a currently installed BIOS update from the BIOS storage into a caller-provided RAM buffer. Table 8 -7 lists the parameters and return codes for the function.

Table 8-7. Parameters for Read BIOS Update Data Function

Input:

Register	Name	Description
AX	Function Code	`0D042h`
BL	Subfunction	`03h` - Read Update
ES:DI	BufferAddress	Real Mode pointer to the Intel Update structure that will be written with the binary data.
ECX	Scratch Pad1	Real Mode Segment address of 64 kilobytes of RAM Block (lower 16 bits).
ECX	Scratch Pad2	Real Mode Segment address of 64 kilobytes of RAM Block (upper 16 bits).
DX	Scratch Pad3	Real Mode Segment address of 64 kilobytes of RAM Block.
SS:SP	Stack pointer	32 kilobytes of Stack Minimum.
SI	Update Number	The index number of the update block to be read. This value is zero based and must be less than the Update Cnt returned from the Presence Test function.

Output:

Register	Name	Description
CF	Carry Flag	Carry Set - Failure - `AH` contains Status. Carry Clear - All return values are valid.
AH	Return Code	Status of the Call.
AL	OEM Error	Additional OEM Information.

Return Codes:

Code	Description
SUCCESS	Function completed successfully.
READ_FAILURE	A failure because of the inability to read the storage device.
UPDATE_NUM_INVALID	Update Number exceeds the maximum number of update blocks implemented by the BIOS. See Table 8 -8 for code definitions.

Description:

The Read function enables the caller to read any update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies exactly 2048 bytes into the location pointed to by ES:DI, with the contents of the Update block represented by Update Number.

An update block is considered unused and available for storing a new update if its Header Version contains the value `0FFFFFFFFh` after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than `0FFFFFFFFh`. The BIOS is responsible for translating this information into the header provided by this function.

Return Codes

After the call has been made, the return codes listed in Table 8 -8 are available in the AH Register.

Table 8-8. Error Code Definitions

Return Code	**Value**	**Description**
SUCCESS	00h	Function completed successfully
NOT_IMPLEMENTED	86h	Function not implemented
ERASE_FAILURE	90h	A failure because of the inability to erase the storage device
WRITE_FAILURE	91h	A failure because of the inability to write the storage device
READ_FAILURE	92h	A failure because of the inability to read the storage device
STORAGE_FULL	93h	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system
CPU_NOT_PRESENT	94h	The processor stepping does not currently exist in the system
INVALID_HEADER	95h	The Update Header contains a header or loader version that is not recognized by the BIOS
INVALID_HEADER_CS	96h	The Update does not checksum correctly
SECURITY_FAILURE	97h	The Update was rejected by the processor
INVALID_REVISION	98h	The same or more recent revision of the update exists in the storage device

1.4.5 Protected Mode Interface

As an alternative to the INT 15h-based interface, an application can use the protected mode interface to integrate updates into the system BIOS.

Calling Convention

This section describes the method for system software to determine whether the system supports the BIOS Upgrade Protected Mode Extensions. This installation check indicates whether the system BIOS Extensions are present and the entry point to these BIOS functions.

The installation check must search for a signature of the ASCII string $IBU in system memory, starting from `0F0000h` to `0FFFFFh` at every 16-byte boundary. This signature indicates that the system supports the Pentium Pro Processor BIOS Upgrade Protected Mode Extensions. The signature identifies the start of a structure that specifies the entry point of the BIOS code implementing the support described in this document.

The system software can determine if the structure is valid by performing a **Checksum** operation: Calculate the checksum by adding up *Length* bytes from the top of the structure, including the *Checksum* field, into an 8-bit value. A resulting sum of zero indicates a valid checksum operation.

The entry points specified in this structure are the software interface to the BIOS functions. The structure element that specifies the 16-bit protected mode entry point enables the caller to construct a protected mode selector for calling this support.

Table 8 -9 shows the structure of the Pentium Pro Processor BIOS Upgrade Extensions Installation Structure.

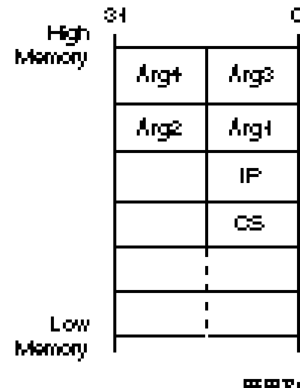
Table 8-9. BIOS Upgrade Extensions Data Fields

Field	Offset	Length	Description
Signature	00h	4 bytes	$IBU (ASCII string) where byte 0='$' (24h), byte 1= byte 1=I (49h), byte 2='B' (42h), and byte 3='U' (55h).
Version	04h	byte	01h for version 1.0. A binary value that implies a level of compliance with version changes of the Pentium Pro Processor BIOS Upgrade specification.
Length	05h	byte	17h, the length of the entire Installation Structure expressed in bytes. The length count starts at the Signature field.
Checksum	06h	byte	Calculated by adding up the number of bytes in the Installation Structure, including the Checksum field, into an 8-bit value. A resulting sum of zero indicates a valid checksum.
Real mode 16-bit offset to entry point	07h	word	Specifies the segment offset of the real mode entry point.
Real mode 16-bit code segment address	09h	word	Value varies.
16-bit protected mode offset to entry point	0Bh	word	Specifies the code segment base address, so that the caller can construct the descriptor from this segment base address before calling this support from protected mode. The offset value is the offset of the entry point. The 16-bit protected mode interface is assumed to be sufficient for 32-bit protected mode callers.
16-bit protected mode code segment base address	0Dh	dword	Value varies.
Real mode 16-bit data segment address	11h	word	Value varies.
16-bit protected mode data segment base address	13h	dword	Value varies.

The caller must also construct data descriptors for the functions that return information in the function arguments that are pointers. The only limitation is that the pointer offset can only point to the first 64 kilobytes of a segment.
If a call is made to these BIOS functions from 32-bit protected mode, the 32-bit stack is used for passing any arguments to the BIOS functions. However, it is important to note that the BIOS functions are not implemented as a full 32-bit protected mode interface. They access arguments on the stack as a 16-bit

stack frame. Therefore, the caller must ensure that the function arguments are pushed onto the stack as 16-bit values and not 32-bit values. The stack parameter passing is illustrated in Figure 8 -4.

Figure 8-4. 16-bit Stack Frame on a 32-bit Stack



The system BIOS can determine whether the stack is a 32-bit stack or a 16-bit stack in 16-bit and 32-bit environments with the load access rights (LAR) byte instruction. The LAR instruction loads the high order double word for the specified descriptor. By loading the access rights for the current stack segment selector, the system BIOS can check the B-bit (Big bit) of the stack segment descriptor, which identifies the stack segment descriptor as either a 16-bit segment (B-bit clear) or a 32-bit segment (B-bit set).
In addition to executing the LAR command to get the entry point stack size, the BIOS code should avoid ADD BP, x type stack operands in runtime service code paths. These operands carry the risk of faulting if the 32-bit stack base happens to be close to the 64-kilobyte boundary. For the 16-bit protected mode interface, it is assumed that the segment limit fields will be set to 64 kilobytes. The code segment must be readable. The current I/O permission bit map must allow access to the I/O ports that the system BIOS needs to perform the function. The current privilege level (CPL) must be less than or equal to the I/O privilege level. This enables the BIOS to use sensitive instructions such as CLI and STI.
The entry point is assumed to have a function prototype of the form

```
int FAR (*entryPoint)(int Function, ...);
```

and follow the standard C calling conventions.
System software interfaces with all of the functions described in this specification by making a far call to this entry point. As noted above, the caller passes a function number and a set of arguments based on the function being called. Each function also includes an argument specifying a data selector that enables the BIOS to access and update variables within the system BIOS memory space. This data selector parameter is required for protected mode callers. The caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure, a limit of 64 kilobytes, and the descriptor must be read/write capable. Real mode callers are required to set this parameter to the real mode 16-bit data segment address specified in the Installation Structure.

Function 00h – Presence Test

This function verifies that the BIOS has implemented the required BIOS update functions.
Synopsis:

```
short FAR (*entryPoint)(Function, IBUSlotCount, IBUSignature, minWriteSize,
IBULoaderVer, BiosSelector);
short function;                   /* Pentium Pro Processor BIOS Upgrade Function 00h*/
unsigned short IBUSlotCount;/* Number of BIOS Update Data Records storable by BIOS*/
char FAR *IBUSignature;           /* Pointer to the IBUSignature*/
unsigned long minWriteSize;       /* Minimum Buffer Size to Write BIOS Update Data */
```

```
unsigned long IBULoaderVer;    /* BIOS Update Loader Code version number */
unsigned short BiosSelector;   /* BIOS readable/writable selector */
```

Description:

`IBUSlotCount`	Updated by the BIOS call with the total number of BIOS Upgrade Data blocks supported by the BIOS. For example, f the BIOS is capable of storing four BIOS Upgrade Data blocks, a value of 04h is returned by this function.
`IBUSignature`	Updated by the BIOS call to point to the ASCII character string INTELPEP.
`MinWriteSize`	Updated by the BIOS call with the size, in bytes of scratch buffer required by the BIOS to perform the Write BIOS Upgrade Data (01h) function. If the BIOS stores the BIOS Upgrade Data in a block erase device, this value would be the size of the largest block containing BIOS Upgrade Data. If the value of the `minWriteSize` returned by the BIOS is zero, then the BIOS does not require any additional buffer space to perform the Write BIOS Upgrade Data function.
`IBULoaderVer`	Updated by the BIOS call with the version number of the loader code implemented by the BIOS. The initial release of the BIOS Update loader code should cause the return of `00000001h`.
`BiosSelector`	Enables the system BIOS, if necessary, to update system variables contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure, a limit of 64 kilobytes, and the descriptor must be read/write capable. If this function is called from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

Returns:
If successful - SUCCESS, or else the Error Code is returned in AX, the FLAGS and all other registers are preserved. See Table 8 -10 for return codes.

Example:
The following example illustrates how the C style call interface could be made from an assembly language module:

```
push     BiosSelector

pushd    IBULoaderVer
pushd    minWriteSize
push     segment/selector of IBUSignature ; Pointer to BIOS Update Signature
push     offset of IBUSignature
push     IBUSlotCount
push     PRESENCE_TEST
call     FAR PTR entryPoint
add      sp, 18t                    ; Clean up stack
cmp      ax, SUCCESS                ; Function completed successfully?
jne      error
```

Function 01h – Write BIOS Update Data

This function writes the BIOS Update Data contained in the buffer specified by `IBUBuffer` into a storage area determined by the BIOS.

Synopsis:

```
short FAR (*entryPoint)(Function, IBUBuffer, segStructure, BiosSelector);
short function;                /*Pentium Pro Processor BIOS Upgrade Function 01h*/
unsigned char FAR *IBUBuffer;  /* Pointer to buffer containing BIOS Update Data */
unsigned short FAR *segStructure /*Pointer to buffer containing Segment/Selector Info*/
unsigned short BiosSelector;     /* BIOS readable/writable selector */
```

Description:

`segStructure` The segment/selector:offset of a structure containing an array of scratch segments. The sum of the size of all the segments should be at least as large as the size specified by the minWriteSize parameter returned from the Installation Check Function (`00h`). The structure is as follows:

```
DW    Segment/Selector 0,    Limit of Segment/Selector 0
DW    Segment/Selector 1,    Limit of Segment/Selector 1
DW    ......
DW    Segment/Selector n,    Limit of Segment/Selector n
DW    NULL
```

`BiosSelector` Enables the system BIOS, if necessary, to update system variables contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure and a limit of 64 kilobytes. The descriptor must be read/write capable. If this function is called from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

The BIOS is responsible for selecting an appropriate update block within the non-volatile storage for retaining the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authentication of the proposed update before incorporating it into storage. Before writing the update block into NVRAM, the BIOS must ensure that the update structure meets the following criteria in the order listed:

1. The Update header version must be equal to an Update header version recognized by the BIOS.

2. The Update loader version in the Update header must be equal to the Update loader version contained within the BIOS image.
3. The Update block must checksum to zero. This checksum is computed as a 32-bit summation of all 512 double words in the structure, including the header.

The BIOS selects an update block from within non-volatile storage for retaining the candidate update. The BIOS may select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected currently contains an update, then the following additional criteria apply to overwrite it:

1. The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM.
2. The Update Revision in the proposed update must be greater than the Update Revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS may overwrite an update block for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings identified in the MP Specification table to the processor steppings currently in the system.

Finally, before storing the proposed update into NVRAM, the BIOS must verify the authenticity of the update via the mechanism described previously for update authentication.

When performing the Write Update function the BIOS must record the entire update, including the header and the update data. When writing an update the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping.

Returns:

If successful - SUCCESS, else the Error Code is returned in AX, the FLAGS and all other registers are preserved. See Table 8 -10 for return codes.

Example:
The following example illustrates how the C style call interface could be made from an assembly language module:

```
push        BiosSelector
push        segment/selector of segStructure       ; Pointer to Segment Structure
push        offset of segStructure
push        segment/selector of IBUBuffer    ; Pointer to BIOS Update Data Buffer
push        offset of IBUBuffer
push        WRITE_BIOS_UPDATE_DATA
call        FAR PTR entryPoint
add         sp, 12t                          ; Clean up stack
cmp         ax, SUCCESS                      ; Function completed successfully?
jne         error
```

Function 02h – BIOS Update Control

This function enables or queries the global loading of BIOS Update Data into the processor(s).

Synopsis:

```
short FAR (*entryPoint)(Function, segStructure,  taskControl, BiosSelector);
short function;  /* Pentium Pro Processor BIOS Upgrade Function 02h */
unsigned short FAR *segStructure /* Pointer to buffer containing Segment/Selector
Info */
unsigned short taskControl      /* Task to Perform */
unsigned short BiosSelector;/* BIOS readable/writable selector */
```

Description:

`segStructure` — The segment/selector:offset of a structure containing an array of scratch segments. The sum of the size of all the segments should be at least as large as the size specified by the `minWriteSize` parameter returned from the Presence Test Function (`00h`). The structure is as follows:

```
DW      Segment/Selector 0,           Limit of Segment/Selector 0
DW      Segment/Selector 1,           Limit of Segment/Selector 1

DW  ... ...
DW      Segment/Selector n,           Limit of Segment/Selector n
DW      NULL
```

`taskControl` — Defines the task to be performed. The following tasks are defined:

Task	Value	Description
Enable	`0000h`	Enable the BIOS Update loading at initialization time
Query	`0001h`	Determine the current state of BIOS Update loading without changing its state

`BiosSelector` — Enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure, a limit of 64KB, and the descriptor must be read/write capable. If this function is called from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

Returns:

If successful - SUCCESS, else the Error Code are returned in AX, the FLAGS and all other registers are preserved. See Table 8 -10 for return codes.

Example:
The following example illustrates how the C style call interface could be made from an assembly language module:

```
push        BiosSelector
push        taskControl
push        segment/selector of segStructure  ; Pointer to Segment Structure
push        offset of segStructure
push        BIOS _ UPDATE _CONTROL
call        FAR PTR entryPoint
add         sp, 10t    ; Clean up stack
cmp         ax, SUCCESS; Function completed successfully?
jne         error
```

Function 03h – Read BIOS Update Data

This function reads the BIOS Update Data block indexed by the `IBUIndex` into the buffer specified by `IBUBuffer`.

NOTE. *This function is required for Pentium Pro Processor BIOS Upgrade Support.*

Synopsis:

```
short FAR (*entryPoint)(Function, IBUBuffer, IBUIndex, BiosSelector);
short function;                /* Pentium Pro Processor BIOS Upgrade Function 03h*/
unsigned char FAR *IBUBuffer;/* Pointer to buffer to write BIOS Update Data */
unsigned short IBUIndex        /* Index of BIOS Update Record to Read */
unsigned short BiosSelector;   /* BIOS readable/writable selector */
```

Description:

`IBUBuffer` The segment/selector:offset of the buffer that the BIOS Update Data is to be read into.

`IBUIndex` The index of the BIOS Update Data to be read.

`BiosSelector` Enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit protected mode data segment base address specified in the Installation Structure and a limit of 64 kilobytes. The descriptor must be read/write capable. If this function is called from real mode, `BiosSelector` should be set to the real mode 16-bit data segment address as specified in the Installation Structure. This function is available in real mode and 16-bit protected mode.

Returns:
If successful - SUCCESS, or else the Error Code is returned in AX, the FLAGS and all other registers are preserved. See Table 8 -10 for return codes.

Example:
The following example illustrates how the C style call interface could be made from an assembly language module:

```
push        BiosSelector
push        IBUIndex
push        segment/selector of IBUBuffer  ; Pointer to BIOS Update Data Buffer
```

```
push            offset of IBUBuffer
push            READ_BIOS_UPDATE_DATA
call            FAR PTR entryPoint
add             sp, 10t     ; Clean up stack
cmp             ax, SUCCESS    ; Function completed successfully?
jne             error
```

Return Codes

After the call has been made, the codes listed in Table 8 -10 are available in the AX Register.

Table 8-10. Return Codes

Return Code	Value	Description
SUCCESS	00h	Function completed successfully
NOT_IMPLEMENTED	86h	Function not implemented
ERASE_FAILURE	90h	A failure because of the inability to erase the storage device
WRITE_FAILURE	91h	A failure because of an inability to write the storage device
READ_FAILURE	92h	A failure because of an inability to read the storage device
STORAGE_FULL	93h	The BIOS non-volatile storage area is unable to accommodate the update, because all available update blocks are filled with updates that are needed for processors in the system.
INVALID_HEADER	94h	The Update Header contains a header or loader version that is not recognized by the BIOS
INVALID_HEADER_CS	95h	The Update does not checksum correctly
SECURITY_FAILURE	96h	The Update was rejected by the processor
INVALID_REVISION	97h	1The same or more recent revision of the update exists in the storage device