Runtime Microcode Update

Technical Paper

June 2023

Revision1.0

Document Number: 356111-001US

Contents

Revision History

Document Number	Revision Number	Description	Date
356111-001US	1.0	Initial release.	June 2023

1 Introduction

Intel microcode updates provide the ability to update the processor firmware after manufacturing. Typically, a microcode update (MCU) is loaded by the platform firmware during the boot process (e.g., BIOS) and/or during the boot phase of the operating system. However, due to evolving usage models, it has become increasingly common to need to load microcode updates while the system is fully operational, including while end user workloads and virtual machines may be running.

This document describes architectural enhancements and a software methodology to facilitate the efficient loading of microcode updates during runtime. The enhancements described are intended to be backward compatible such that existing OS MCU drivers should continue to work as is, while newer drivers may be developed to take advantage of and benefit from the enhancements.

The architectural enhancements described here are targeted for future products releases. They are preliminary and subject to change. The exact definitions and product intercepts are also subject to change until they are finalized.

2 Runtime Microcode Update

Loading a microcode update during runtime requires coordination across all logical processors within the platform.

Loading a microcode update may cause changes to CPU capabilities or functionality that may be visible to running software. This is true whenever a microcode update is loaded, but it is particularly important when loading an update during runtime, while virtual machines and/or user mode applications may be running. Examples may include, but are not limited to:

- Changes to CPUID or other enumeration bits (e.g., IA32_ARCH_CAPABILITIES).
- Changes to Architectural Model Specific Registers (MSRs) or individual bits within Architectural MSRs.

When loading a newer update over an older update, Intel intends that any such changes to capabilities or functionality will be architecturally compatible with any running software. However, if an older update is loaded over a newer update (i.e., rollback), then software must ensure, prior to loading the update, that any functionality not present in the older update is no longer in use by any running software. Failure to do so may cause running software to malfunction.

2.1 Microcode Update Enumeration and Status

The IA32_MCU_ENUMERATION MSR is a read-only register that provides information about the microcode update capabilities of the processor. The IA32_MCU_STATUS MSR is a read-only register that provides status information for the most recent attempt to load a microcode update. The IA32_MCU_ENUMERATION and IA32_MCU_STATUS MSRs are present if both of the following are true:

- CPUID.(EAX=07H, ECX=0):EDX[29] == 1, and
- IA32_ARCH_CAPABILITIES[16] == 1.

CPUID.(EAX=07H, ECX=0):EDX[29] indicates the presence of IA32_ARCH_CAPABILITIES (MSR 10AH), and IA32_ARCH_CAPABILITIES[MCU_ENUMERATION] (bit 16) indicates the presence of IA32_MCU_ENUMERATION (MSR 7BH) and IA32_MCU_STATUS (MSR 7CH).

The format of the IA32_MCU_ENUMERATION MSR is described below. Later sections provide details on how these bits should be used.

IA32_MCU_ENUMERATION (MSR 7BH) – Read-only – Package Scoped		
Bit Field	Name	Description
0	UNIFORM_MCU_AVAIL	When set to 1, uniform microcode update is available, and UNIFORM_MCU_SCOPE (bits [10:8]) indicates the scope of writes to IA32_BIOS_UPDT_TRIG. When set to 0, uniform microcode update is not available, and writes to IA32_BIOS_UPDT_TRIG are core scoped.
1	UNIFORM_MCU_CONFIG_REQD	When set to 1, indicates that configuration is required to ensure that all MCU components are updated on WRMSR 79H, and UNIFORM_MCU_CONFIG_COMPLETE (bit 2) should be checked to determine whether the necessary configuration has been completed. When set to 0, indicates that no configuration is required, and UNIFORM_MCU_CONFIG_COMPLETE should be ignored.
2	UNIFORM_MCU_CONFIG_COMPLETE	If UNIFORM_MCU_CONFIG_REQD (bit 1) is 0, then this bit should be ignored. If UNIFORM_MCU_CONFIG_REQD is 1, then this bit indicates whether all necessary configuration has been completed to ensure that all MCU components will be updated on WRMSR 79H.
3	ARCH_ROLLBACK_SVN_COMMIT	When set to 1, indicates support for the MCU deferred SVN architecture, SVN reporting architecture, and MCU rollback architecture. See section 4 for details.
4	MCU_STAGING	When set to 1, indicates that the microcode update staging capability is supported by the processor. When supported, use of the MCU staging capability is recommended to reduce the latency of the IA32_BIOS_UPDT_TRIG operation. See section 5 for details.
5	TEE_SVN_SUPPORT	When set to 1, indicates that the processor supports the IA32_TEE_SVN_STATUS MSR.
7:6	Reserved	Reserved for future use.
15:8	UNIFORM_MCU_SCOPE	Indicates the current* uniform microcode update scope: 0x02: Core scoped 0x80: Package Scoped 0xC0: Platform Scoped All others: Reserved for future use * The value of this field reflects the state of platform configuration and may change as the configuration changes during the boot process. Once configuration is complete, it is not expected to change during runtime.
63:16	Reserved	Reserved for future use.

The format of the IA32_MCU_STATUS MSR is described below. Refer to the section "Completing the Update" for details on how these bits should be used.

IA32_MCU_STATUS (MSR 7CH) – Read-only – Core Scoped		
Bit Field	Name	Description
0	MCU_PARTIAL_UPDATE	When set to 1, indicates that the most recent write to IA32_BIOS_UPDT_TRIG resulted in a partial update. This means that microcode update components were only partially updated after some portion of the MCU had already been committed and the Revision ID had been updated.
1	AUTH_FAIL_ON_MCU_COMPONENT	When set to 1, indicates that an authentication failure occurred on some portion of the MCU after another portion of the MCU had already been committed and the Revision ID had already been updated on the most recent write to IA32_BIOS_UPDT_TRIG.
2	Reserved	Reserved for future use.
3	POST_BIOS_MCU	When set to 1, indicates that an update was successfully loaded via IA32_BIOS_UPDT_TRIG after bit 0 of MSR_BIOS_DONE (MSR 151H) was set to 1. See section 4.2.1 for the usage of this bit.
63:4	Reserved	Reserved for future use.

2.2 Preparing to Load the Update

On systems that enumerate Uniform Microcode Update, model-specific platform configuration may be necessary to enable full support for runtime microcode updates. Platform firmware (e.g. BIOS) would typically perform this configuration during the boot process before launching the OS. If the necessary configuration is not done, then runtime updates may not properly load all components of the update. The UNIFORM_MCU_CONFIG_REQD field (bit 1) of IA32_MCU_ENUMERATION indicates whether the CPU requires any such configuration. If this bit is set, then OS software should consult the UNIFORM_MCU_CONFIG_COMPLETE field (bit 2) to determine whether the necessary configuration has been completed.

UNIFORM_MCU_ CONFIG_REQD	UNIFORM_MCU_ CONFIG_COMPLETE	Description
0	-	No configuration is required. Runtime updates are enabled.
1	0	Necessary configuration has not been correctly completed. Runtime updates may not load all MCU components.
1	1	Necessary configuration has been correctly completed, and runtime updates are enabled.

If UNIFORM_MCU_CONFIG_REQD is 1 and UNIFORM_MCU_CONFIG_COMPLETE is 0, then the necessary configuration has not been completed, and the update may not load all components of the MCU. In this scenario, Intel recommends that microcode updates not be loaded at runtime.

To load an update during runtime, software must synchronize all logical processors within the system (perform a rendezvous) to load the update in a coordinated manner.

Prior to loading the update, software may wish to snapshot any enumeration values of interest to detect any changes caused by the update. Examples of such values may include CPUID leaves and sub-leaves, architectural enumeration MSRs (IA32_ARCH_CAPABILITIES, IA32_CORE_CAPABILITIES, IA32_MCG_CAP, etc.), VMX capability MSRs, CAPID registers, etc. The exact set of values of interest may be OS/VMM specific.

2.3 Uniform Microcode Update

On some processor configurations, it is sufficient to load the update on only one logical processer per package or per platform. This is known as Uniform Microcode Update. On other processor configurations, at least one logical processor per physical core must load the update. Uniform Microcode Update is supported if IA32_MCU_ENUMERATION is present and UNIFORM_MCU_AVAIL (bit 0) is set to 1.

If Uniform Microcode Update is not supported, then writes to IA32_BIOS_UPDT_TRIG are core scoped. If Uniform Microcode Update is supported, then the UNIFORM_MCU_SCOPE field (bits [10:8]) of IA32_MCU_ENUMERATION indicate whether writes to IA32_BIOS_UPDT_TRIG are core, package, or platform scoped.

Non-Uniform Microcode Update. On processors that do not enumerate Uniform Microcode Update, or on processors that enumerate Uniform Microcode Update with core scope, writes to IA32_BIOS_UPDT_TRIG are core scoped, and at least one logical processor per core must load the update.

Package Scoped Uniform Microcode Update. On processors that enumerate Package Scoped Uniform Microcode Update, writes to IA32_BIOS_UPDT_TRIG are package scoped, and it is sufficient for one logical processor per package to load the update. In this case, loading a microcode update on any logical processor will cause the same update to automatically be loaded on all logical processors within the CPU package.

Platform Scoped Uniform Microcode Update. On processors that enumerate Platform Scoped Uniform Microcode Update, writes to IA32_BIOS_UPDT_TRIG are platform scoped, and it is sufficient for one logical processor per platform to load the update. In this case, loading a microcode update on any logical processor will cause the same update to automatically be loaded on all logical processors across all CPUs in the system.

NOTE

Some processors may support Uniform Microcode Update even if it is not enumerated, or may support Uniform Microcode Update at a wider scope than what is enumerated. This is known as model-specific Uniform Microcode Update and may occur when certain features are enabled. For example, when Intel® Software Guard Extensions (Intel® SGX) is enabled, the CPU will enable platform scoped Uniform Microcode Update even if it is not enumerated or is enumerated with a narrower scope. In these cases, the model-specific behavior will never have narrower scope than what is enumerated. Software may choose to ignore this model-specific behavior and rely solely on the architectural enumeration (or lack thereof) when loading the update. It is always allowed to load the update on more logical processors than necessary, though this may result in unnecessary additional latency.

The following table summarizes the enumeration of the various configurations:

CPUID.(EAX=07H, ECX=0):EDX[29]	IA32_ARCH_CAPABILITIES[16]	IA32_MCU_ENUMERATION		Description
		Bit 0	Bits [15:8]	
0	-	-	-	Core Scoped
1	0	-	-	Core Scoped
1	1	0	-	Core Scoped
1	1	1	0x02	Core Scoped
1	1	1	0x80	Package Scoped
1	1	1	0xC0	Platform Scoped

The following pseudo-code demonstrates how to determine the scope of IA32_BIOS_UPDT_TRIG:

```
Determine_Scope() {
    if (CPUID.(EAX=7, ECX=0).EDX[29] == 0)
        return CORE_SCOPED

    if (IA32_ARCH_CAPABILITIES[16] == 0)
        return CORE_SCOPED

    if (IA32_MCU_ENUMERATION[0] == 0)
        return CORE_SCOPED

    if (IA32_MCU_ENUMERATION[15:8] == 0xC0)
        return PLATFORM_SCOPED

    if (IA32_MCU_ENUMERATION[15:8] == 0x80)
        return PACKAGE_SCOPED

    return CORE_SCOPED
}
```

2.4 ILP Determination

Depending on the scope of the IA32_BIOS_UPDT_TRIG MSR, one or more logical processors must be designated to load the microcode update. Logical processors that explicitly load the update are known as Initiating Logical Processors (ILPs)[1]. Logical processors that do not explicitly load the update are known as Receiving Logical Processors (RLPs).

- On processors that enumerate platform scoped Uniform Microcode Update, there must be at least one ILP for the platform.
- On processors that enumerate package scoped Uniform Microcode Update, there must be at least one ILP per package.
- On processors that do not enumerate Uniform Microcode Update, or processors that enumerate Uniform Microcode Update but do not indicate Platform or Package scope, there must be at least one ILP per core.

In all cases, loading the update on more logical processors than necessary (including loading it on all logical processors) is allowed and will be functional. However, it may not be efficient, and could result in longer latencies.

The following pseudo-code demonstrates how to determine whether a given logical processor is an ILP:

```
Is_ILP() {
    scope = Determine_Scope()

    if (scope == PLATFORM_SCOPED) {
        if (platform leader)
            return True
        else
            return False
    } else if (scope == PACKAGE_SCOPED) {
        if (package leader)
            return True
        else
            return False
    } else {
        if (core leader)
            return True
        else
            return False
    }
}
```

[1] In this context, the terms ILP and RLP are unrelated to the same or similar terms used for other capabilities, such as the Safer Mode Extensions (SMX) that support an Intel® Trusted Execution Technology (Intel® TXT) platform.

2.5 Loading the Update

Once all logical processors have been designated as either an ILP or RLP, then software should synchronize all threads. After synchronization, all RLPs must wait in a spin loop while all ILPs write to IA32_BIOS_UPDT_TRIG to load the update. On processor configurations with more than one ILP, all ILPs may load the update in parallel to minimize the overall time required to load the update. The RLP spin loop must consist of only simple instructions; `PAUSE` or `LFENCE` may be used to throttle the loop, but `MWAIT` and `HLT` must not be used.

After all ILPs have written to IA32_BIOS_UPDT_TRIG, all logical processors must verify that the new update signature matches the update revision from the microcode update header. When the CPUID instruction is executed, the CPU populates the IA32_BIOS_SIGN_ID MSR (MSR 0x8B) with the update signature of the currently loaded microcode update. Therefore, to determine the signature of the currently loaded microcode update, software must clear IA32_BIOS_SIGN_ID to 0, then execute the CPUID instruction, then read the IA32_BIOS_SIGN_ID MSR.

The following pseudo-code demonstrates the sequence to read the current update revision:

```
Get_Current_MCU_Revision() {
    wrmsr(IA32_BIOS_SIGN_ID, 0)
    cpuid(1)
    return rdmsr(IA32_BIOS_SIGN_ID) >> 32
}
```

The following pseudo-code demonstrates the complete sequence to load the update:

```
Load_MCU(update_ptr) {
    global uint32 ILP_count = 0

    bool ILP = Is_ILP()
    if (ILP)
        atomic_increment(ILP_count)

    sync_logical_processors()

    if (ILP) {
        wrmsr(IA32_BIOS_UPDT_TRIG, update_ptr)
        atomic_decrement(ILP_count)
    }

    while (ILP_count != 0)
        pause

    if (Get_Current_MCU_Revision() == expected_rev)
        return Success
    else
        return Fail
}
```

2.6 Completing the Update

On processors that enumerate the IA32_MCU_STATUS MSR, software should confirm that the entire update was successfully loaded. If both MCU_PARTIAL_UPDATE (bit 0) and AUTH_FAIL_ON_MCU_COMPONENT (bit 1) are 0, then the entire update was loaded successfully. If MCU_PARTIAL_UPDATE (bit 0) is set to 1, then one or more MCU components failed to load. If AUTH_FAIL_ON_MCU_COMPONENT (bit 1) is also set to 1, then one or more components of the MCU failed authentication. If AUTH_FAIL_ON_MCU_COMPONENT is 0, then the partial load was not related to authentication.

MCU_PARTIAL_UPDATE	AUTH_FAIL_ON_MCU_COMPONENT	Description
0	-	Success.
1	0	Partial update due to configuration error.
1	1	Partial update due to authentication failure.

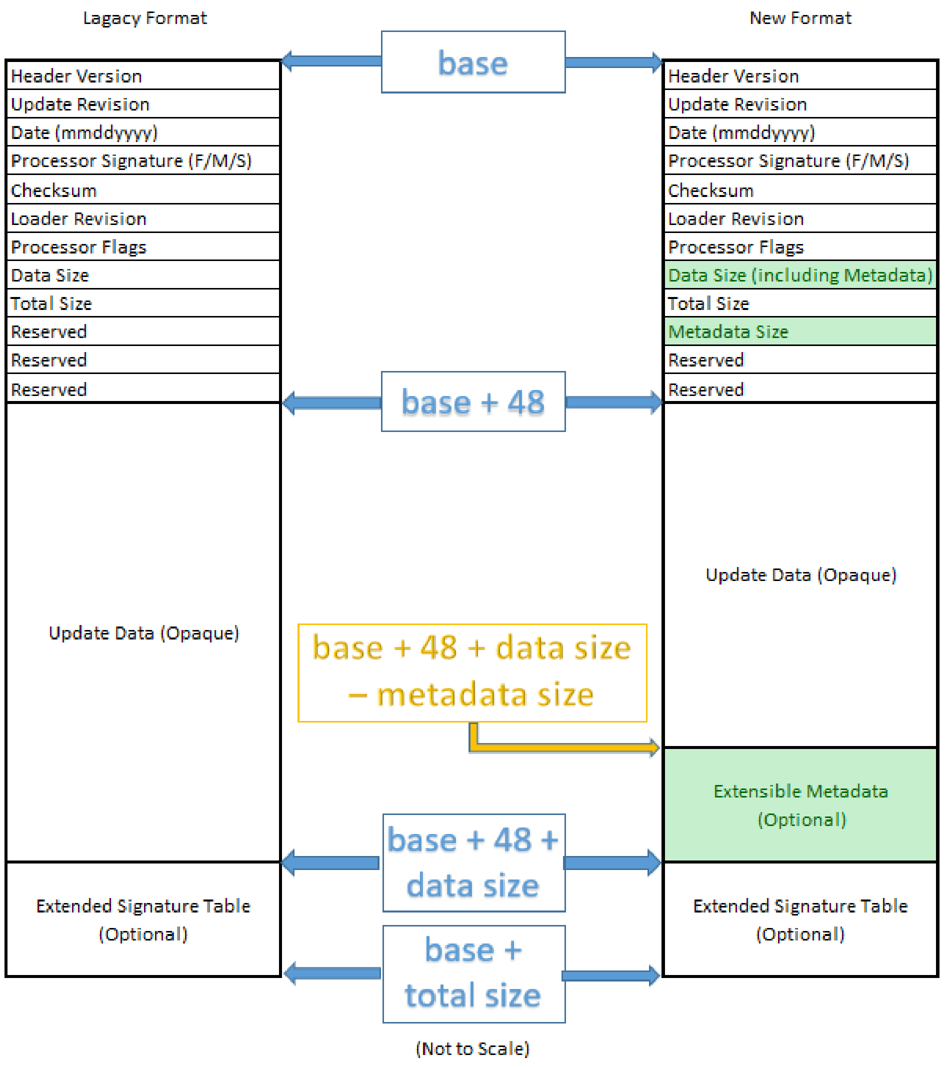
After loading the update and verifying the update signature and status, software may wish to compare the CPUID and other feature enumeration information against the earlier snapshot to detect any changes caused by the update.

After loading the update, software may also need to perform additional follow up actions, such as SGX TCB recovery, or activation or configuration of capabilities added via the update. Which logical processors need to perform these actions may vary depending on the specific actions required and is independent from the scope of the Uniform Update.

3 Extended MCU Metadata

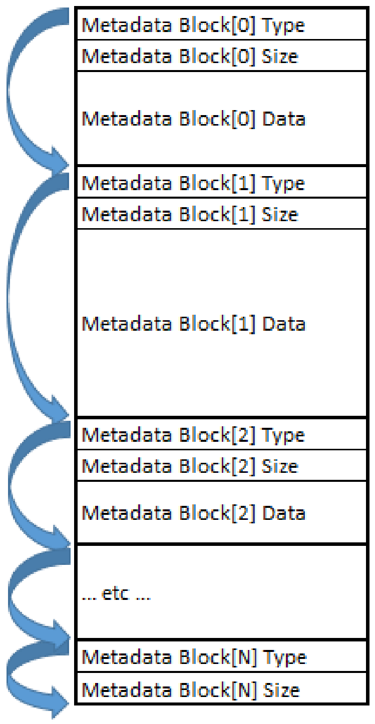
The existing microcode update binary format is described in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Section 10.11.1. It includes a 48-byte header, an encrypted microcode update data block, and an optional extended signature table.

To support enhanced usage models for microcode updates such as runtime rollback, additional metadata must be carried within the microcode update binary file itself. This section describes a mechanism to add additional metadata into the microcode update binary image that can be used by enlightened software while maintaining backward compatibility with existing legacy software.



The previously reserved DWORD at byte offset 36 of the microcode update header is repurposed as the Metadata Size field. If the value of this field is 0, then the microcode update image does not contain any additional metadata. If the value of this field is non-zero, then additional metadata is present, and it begins at an offset of (Data Size + 48 – Metadata Size). This relationship maintains the previous definitions of both the Data Size and Total Size fields of the microcode update header to preserve backward compatibility with existing legacy software that is not aware of the additional metadata.

The additional metadata, if present, is organized as a list of one or more metadata blocks. Each metadata block consists of a 32-bit Type field, a 32-bit Size field, and zero or more DWORDs of data. The Type field indicates the type of metadata contained in this block. The Size field indicates of the size in bytes of this metadata block, including the type and size fields themselves, and will always be a multiple of four bytes. Software can iterate over the list of blocks by adding the size of each block to its starting offset to find the starting offset of the next block.



The following types of metadata are defined:

- **Type 0: End of Metadata.** The last metadata block in the list will always be of Type 0: End of Metadata. This can be used by software to detect the end of the additional metadata. The Size field of the End of Metadata block will always indicate 8 bytes: 4 bytes for the type field and 4 bytes for the size field itself; the End of Metadata block does not have any additional data.

Type	00000000h
Size (bytes)	00000008h
No Additional Data	

- **Type 2: Rollback Metadata.** This metadata block contains information about the runtime rollback capabilities of the microcode update. Refer to section 4.3 for the details of this metadata type.

All other Type encodings are reserved for future use. To ensure future compatibility, software must ignore any metadata types that it does not recognize.

The following pseudo-code demonstrates how to locate a specific metadata block type within the microcode update image:

```
Find_Metadata_Type(Update_Base, Type) {
    if (Update_Base.Metadata_Size == 0)
        return NULL

    Metadata_Base =
        Update_Base + 48 + Update_Base.Data_Size – Update_Base.Metadata_Size

    while (Metadata_Base.Type != END_OF_METADATA) {
        if (Metadata_Base.Type == Type)
               return Metadata_Base
        } else {
                Metadata_Base += Metadata_Base.Size
        }
    }
    return NULL
}
```

4 Microcode Update Rollback

In general, a microcode update should only be loaded if it is newer (i.e., has a higher revision) than what is already loaded. However, under some usage scenarios, it may be desirable to revert, or roll back, the microcode update to a previous revision.

MCU Rollback is the act of requesting a runtime update to an older MCU with a lower revision while a newer MCU with a higher revision is currently loaded on the CPU. The intent of rollback is to revert the system, without a reboot, to a state resembling (or matching) the state prior to a WRMSR 79H with a new MCU.

For example:

1. WRMSR 79H with MCU revision X.
2. MCU loaded successfully and CPU resumes running software with no issues.
3. WRMSR 79H with MCU revision Y (Y > X).
4. MCU loaded successfully and CPU resumes running software.
5. System instability observed after loading MCU Y and we want to revert to the state of the system as it was after (2).
6. WRMSR 79H with MCU revision X – this is the rollback action.

This section describes extensions to the microcode update format and the CPU architecture to enumerate and enable certain specific supported rollback configurations.

- Sections 4.1 and 4.2 describe extensions to the processor architecture for software to enable rollback capabilities and enumerate supported rollback configurations.

- Section 4.3 describes extended MCU metadata within the microcode update binary to enumerate supported rollback configurations and limitations for a given update.

On products that support the rollback architecture described here, Intel will support and validate specific rollback targets for each new microcode update. Intel does not guarantee or support rolling back to any arbitrary older microcode update that is not enumerated by this architecture.

The rollback architecture described in this section is enumerated by IA32_MCU_ENUMERATION bit 3.

4.1 MCU SVN and Deferred SVN Commit

4.1.1 MCU SVN

Microcode updates have a Secure Version Number (SVN) to manage rollback which would preclude users from rolling back MCU past a given point. Once an MCU is loaded, its SVN explicitly prevents loading any (generally older) MCU with a lower SVN.

This is intended to prevent a Ring 0 attacker from being able to roll back certain updates to expose exploitable security issues.

4.1.2 Deferred MCU SVN Commit

Without architectural rollback support, MCU SVN is committed once MCU is loaded via WRMSR 79H.

This architecture adds an option to defer SVN commit to a later point after WRMSR 79H. This will allow validating the system, decide on rollback, and if everything is fine to commit the MCU SVN via a new MSR write.

We are adding two new MSRs, one to decide on the SVN commit mode and one to identify the need for manual commit and perform the commit.

IA32_MCU_SVN_CONFIG - Allow to configure handling of MCU SVN, there are two cases:

1. Auto-commit – MCU SVN will be committed as part of WRMSR 79H, this will prevent rolling back to MCU with lower SVN.

2. Manual-commit – MCU SVN will not be updated via WRMSR 79H[2], rolling back to previous MCU (same SVN as the current CPU SVN) is allowed after WRMSR 79H and before WRMSR IA32_MCU_SVN_COMMIT.

IA32_MCU_SVN_CONFIG (MSR 7A0) – Read/Write – Package Scoped		
Bit Field	Name	Description
0	LOCK	Update of MSR not allowed once this bit is set, WRMSR will result into #GP.
1	DEFER_SVN	0: Auto-Commit (Legacy SVN handling – default) 1: Manual Commit
63:2	Reserved	Reserved for future use.

IA32_MCU_COMMIT_SVN (MSR 7A1) – Read/Write – Core Scoped		
Bit Field	Name	Description
0	COMMIT_SVN	On Read reports if there is pending SVN to commit On write: “1” - Commit SVN “0” - Ignored
63:1	Reserved	Reserved for future use.

So now we can have two phases for software to decide on MCU SVN handling.

Configuration phase:

Software will configure the IA32_MCU_CONFIG MSR to either manual or auto commit; the default value will be auto commit if the MSR was not configured.

It is expected that systems that do not intend to support rollback will lock the MSR with auto commit.

[2] See exception with the MINIMUM MCU SVN chapter.

Systems that intend to support rollback may choose to lock the MSR after choosing between manual and auto commit.

Software must not change the configuration from manual to auto commit while any core reports COMMIT_SVN as 1 (i.e., an SVN update is pending).

Configuring the IA32_MCU_CONFIG MSR is expected to happen after the WRMSR 79H performed by the BIOS (IA32(MCU_STATUS[POST_BIOS_MCU] == 1). Configuring the MSR to Auto-commit before that can lead into BIOS failure.

Runtime update phase:

When software loads a new MCU with new MCU SVN (MCU_SVN will be part of the MCU header, see chapter 4.3), it cannot rollback to a previous MCU with lower SVN (Last loaded MCU SVN) if auto commit was chosen in the configuration phase.

It can rollback to lower MCU SVN when manual commit was chosen in the configuration phase.

Expectation is that after verifying that the new MCU does not cause any system instability, SW performs WRMSR to IA32_MCU_COMMIT MSR with EAX=1 to commit the SVN.

There is no enforcement for software to commit the MCU SVN, but not doing so will prevent loading a subsequent newer MCU with higher MCU SVN (See software rules chapter) and will not guarantee that the CPU has the required security level.

Example of software flow with manual commit:

1. WRMSR 79H with MCU revision X with MCU_SVN A.
2. MCU loaded successfully and CPU resume to run software, no issues.
3. WRMSR IA32_MCU_SVN_COMMIT with EAX=1 – MCU_SVN A committed.
4. WRMSR 79H with MCU revision Y (Y > X) and MCU_SVN B (B > A).
5. MCU loaded successfully and CPU resume to run software.
6. System instability observed as a result of loading MCU Y, we want to revert to same situation in (2).
7. WRMSR 79H with MCU revision X and MCU_SVN A.
    a. This action will succeed as we did not yet commit MCU_SVN B.

4.1.3 SVN Reporting

CPU will report the MCU SVN and the value pending in case of manual commit and commit is pending in IA32_MCU_SVN_INFO.

This MSR is enumerated by same rollback enumeration bit: IA32_MCU_ENUMERATION[3].

IA32_MCU_SVN_INFO (MSR 7A2) – Read only – Core Scoped		
Bit Field	Name	Description
15:0	MCU_SVN	Committed MCU SVN, value can be updated on WRMSR 79H or IA32_MCU_SVN_COMMIT
31:16	PENDING_MCU_SVN	Pending SVN, value updated on WRMSR 79H and will be copied to MCU_SVN field on IA32_MCU_COMMIT_SVN WRMSR.
63:32	Reserved	Reserved for future use.

If IA32_MCU_COMMIT_SVN[COMMIT_SVN] indicates than an SVN update is pending, then the PENDING_MCU_SVN field reports the pending SVN. If an SVN update is not pending, then the value of this field is model specific.

4.1.4 Minimum MCU SVN

With manual commit, software can revert to the last MCU that was loaded in the CPU as the new MCU SVN will not be committed on WRMSR 79H, while with auto-commit this cannot happen when MCU SVN is incremented.

A CPU could have a very old MCU loaded and allowing rollback to such an older MCU may create a security threat to the CPU. While we still want to allow rollback with manual commit, rollback may be limited to an SVN greater than what was in effect with the previous MCU.

To aid with this, MCUs have an additional value – MINIMUM_MCU_SVN. If the SVN of the currently loaded MCU is lower than the MINIMUM_MCU_SVN, then this value will become the new IA32_MCU_SVN_INFO[MCU_SVN] once WRMSR 79H is executed regardless of the chosen MCU commit configuration.

MINIMUM_MCU_SVN will be part of the MCU header as mentioned in chapter 4.3.

Here are examples of few scenarios with manual MCU SVN commit:

Base scenario:

- CPU has MCU_SVN=3, and there is no pending SVN commit
  - IA32_MCU_COMMIT_SVN[MCU_SVN] = 0
  - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [3,3]
- WMRSR 79H with MCU that have MCU_SVN=7 and minimum SVN of 5
  - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
  - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]

Scenario #1 on top of base scenario:

Loading MCU with higher MCU_SVN than the pending MCU SVN, while MCU SVN is pending – no success.

- Base scenario
  - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
  - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]
- WRMSR 79H with MCU that have MCU_SVN=8
- Result: **Silent drop**[3], no update MCU revision ID (MSR 8BH)
  - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
  - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]

Scenario #2 on top of base scenario:

Loading MCU with same pending MCU SVN - success

- Base scenario

[3] Silent Drop – WRMSR 79H completed but MCU was not loaded – MSR 8BH still shows old MCU revision.

    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]
- WRMSR 79H with MCU that have MCU_SVN=7
- Result: **success,** update MCU revision ID (MSR 8BH)
    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]

<u>Scenario #3 on top of base scenario:</u>

Loading MCU with lower MCU_SVN than the pending MCU SVN and higher than the committed MCU SVN – success, pending SVN will be updated.

- Base scenario
    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]
- WRMSR 79H with MCU that have MCU_SVN=6
- Result: **success,** update MCU revision ID (MSR 8BH)
    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [6,5]

<u>Scenario #4 on top of base scenario:</u>

Loading MCU with same committed MCU SVN - success, No pending SVN commit.

- Base scenario
    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]
- WRMSR 79H with MCU that have MCU_SVN=5
- Result: **success,** update MCU revision ID (MSR 8BH)
    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 0
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [5,5]

<u>Scenario #5 on top of base scenario:</u>

Loading MCU with MCU_SVN less than the committed MCU SVN – fail.

- Base scenario
    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]
- WRMSR 79H with MCU that have MCU_SVN=4
- Result: **Silent drop**, no update MCU revision ID (MSR 8BH)
    - IA32_MCU_COMMIT_SVN[MCU_SVN] = 1
    - IA32_MCU_SVN_INFO[PENDING_MCU_SVN, MCU_SVN] = [7,5]

4.1.5 SVN Commit Rules

Software needs to adhere to the following rules for successful MCU load:

- MCU SVN is a barrier to rollback beyond committed SVN.
    - Make sure that WRMSR 79H with MCU_SVN >= IA32_MCU_SVN_INFO[MCU_SVN].
- Must Commit a Deferred SVN update before a subsequent newer MCU with higher SVN can be loaded.

- Make sure that WRMSR 79H with MCU_SVN <= IA32_MCU_SVN_INFO[PENDING_MCU_SVN] when IA32_MCU_COMMIT_SVN[MCU_SVN] = 1.

Software should note that:

- New MCU Update with same SVN as current committed SVN does not require manual commit.
- Even with manual commit, new MCU may update the SVN to minimum MCU SVN.

4.2 Rollback ID Reporting

Each MCU can support set of rollback Ids (up to 16), these Ids are determined when MCU created and reported as part of the MCU header.

Rollback Ids are reported by the CPU via 16 read only MSRs (IA32_ROLLBACK_SIGN_ID_0 to IA32_ROLLBACK_SIGN_ID_15) after successful WRMSR 79H.

Rollback Ids MSRs are enumerated by same rollback enumeration bit: IA32_MCU_ENUMERATION[3].

IA32_ROLLBACK_SIGN_ID_x (MSRs 7B0H .. 7BFH) – Read only – Core Scoped (0 <= x <= 15)		
Bit Field	Name	Description
31:0	ROLLBACK_ID	Supported Rollback ID
47:32	ROLLBACK_MCU_SVN	MCU_SVN correspond to the Rollback ID
63:48	Reserved	Reserved for future use.

These MSRs are updated after successful WRMSR 79H, value of zero in one of the MSRs mean that this MSR does not contain valid rollback ID.

If IA32_ROLLBACK_SIGN_ID_x return “zero” then all IA32_ROLLBACK_SIGN_ID_y MSRs after it will return zero (x < y).

ROLLBACK MCU_SVN that correspond to the rollback ID is reported within the ROLLBACK_SIGN_ID MSR, software can check based on the IA32_MCU_SVN_INFO[MCU_SVN] if rollback to this MCU still supported (follow SVN rules).

4.2.1 Rollback and BIOS MCU

During Boot, MCU will be loaded by uCode reset sequence and later loaded by BIOS after MRC (setting bit #0 of MSR_BIOS_DONE - 151H).

Run time software should not attempt to roll back to an MCU that is older than the MCU loaded by BIOS.

- MCU during BIOS may perform initializations that can’t be undone by a later load of an older MCU.

To aid software, a status bit has been added to indicate the stage post BIOS MCU load and report the minimum rollback ID in a new MSR.

- IA32_MCU_STATUS[3] - POST_BIOS_MCU - When set indicates that a successful WRMSR 79H performed (Revision ID reflected in MSR 8BH) after setting bit #0 of MSR_BIOS_DONE (151H).

  Basically, rollback is not supported until we have bit #3 set in the IA32_MCU_STATUS MSR.

IA32_MCU_STATUS (MSR 7CH) – Read-only – Core Scoped		
Bit Field	Name	Description
0	MCU_PARTIAL_UPDATE	When set to 1, indicates that the most recent write to IA32_BIOS_UPDT_TRIG resulted in a partial update. This means that microcode update components were only partially updated after some portion of the MCU had already been committed and the Revision ID had been updated.
1	AUTH_FAIL_ON_MCU_COMPONENT	When set to 1, indicates that an authentication failure occurred on some portion of the MCU after another portion of the MCU had already been committed and the Revision ID had already been updated on the most recent write to IA32_BIOS_UPDT_TRIG.
2	Reserved	Reserved for future use.
3	POST_BIOS_MCU	When set indicates that a successful WRMSR 79H performed (Revision ID reflected in MSR 8BH) after setting bit #0 of MSR_BIOS_DONE (151H)
63:4	Reserved	Reserved for future use.

- IA32_MCU_ROLLBACK_MIN_ID[31:0] - Reports the minimal MCU revision ID that software can rollback to per boot.
  - Exists if IA32_MCU_ENUMERATION[3] set.

IA32_MCU_ROLLBACK_MIN_ID (MSR 7A4H) – Read only – Core Scoped		
Bit Field	Name	Description
31:0	REVISION_ID	Minimal MCU revision ID for rollback
63:32	Reserved	Reserved for future use.

4.3 MCU Header Changes

MCU rollback architecture requires adding metadata into the MCU header, see chapter for extensible MCU metadata for more information on how software can extract the following data from the MCU.

Here is the data structure that added for the MCU rollback architecture:

Type		00000002h
Size (bytes)		0000006Ch
MCU SVN info		DWORD
Rollback ID 0		DWORD
Rollback ID 1		DWORD
Rollback ID 2		DWORD
:		DWORD
:		DWORD
Rollback ID 15		DWORD
Rollback SVN 1	Rollback SVN 0	DWORD
Rollback SVN 3	Rollback SVN 2	DWORD
:	:	DWORD
:	:	DWORD
Rollback SVN 15	Rollback SVN 14	DWORD

MCU SVN info field consists of:

- 15:0 – MCU_SVN
- 31:16 – MINIMUM_MCU_SVN

5 MCU Staging

As the payload size of microcode updates continues to increase, the latency required to load the update via IA32_BIOS_UPDT_TRIG (MSR 79H) has increased. When an update is loaded during runtime, this increased latency may lead to system instability. Microcode Update Staging is a capability to load and verify portions of the microcode update into staging buffers within the processor prior writing IA32_BIOS_UPDT_TRIG. During runtime, the staging can be done as a lower priority task while other processes continue to run normally and will result in significantly lower latency when IA32_BIOS_UPDT_TRIG is written to apply the update.

The MCU Staging capability is enumerated by bit 4 in the IA32_MCU_ENUMERATION MSR. This enumeration bit indicates the presence of the IA32_MCU_STAGING_MBOX_ADDR MSR and the MCU Staging mailbox registers. See section 2.1 for details of the IA32_MCU_ENUMERATION MSR.

5.1 MCU Staging Mailbox Overview

The MCU staging capability provides a mailbox interface for software to load portions of the microcode update into the internal staging buffers. System software uses the mailbox interface to transmit data objects to the staging hardware (a request), and to receive data objects returned from the staging hardware back to system software (a response). Each data object consists of multiple DWORDs that are transmitted through the mailbox interface registers one by one. The following sections describe the data objects, interfaces registers, and protocols in more detail.

The MCU staging mailbox is modeled after the Data Object Exchange (DOE) mechanism defined by the PCIe Specification, but it is not implemented within the PCI configuration space and does not precisely conform to the DOE specification.

5.2 MCU Staging Mailbox Data Object Format

An MCU staging mailbox data object consists of an 8-byte mailbox header that specifies a vendor ID, data object type, object length, and zero or more DWORDs of type-specific data.

Mailbox Data Object Format

Byte 3								Byte 2								Byte 1								Byte 0								DWORD	Section
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Reserved								Data Object Type								Vendor ID																0	Mailbox Header
Reserved														Length																		1	
Type-Specific Data DWORD 0																																2	Type-Specific Data
…																																…	
Type-Specific Data DWORD N																																N+2	

Vendor ID – 16-bit PCI-SIG Vendor ID of the entity that defined the type of this data object.

Data Object Type – 8-bit Type of the data object. The Vendor ID and Data Object Type together uniquely identify the type of the data object and the layout of the type-specific data.

Length – 18-bit length of the entire data object in DWORDs, including the mailbox header. A value of 0 indicates 2^{18} DWORDs (1 MB).

The MCU Staging mailbox supports two data object types: the Discovery object type and the MCU Staging object type.

Discovery Index	Vendor ID	Data Object Type	Description
0	0001H	00H	Discovery
1	8086H	0BH	MCU Staging

5.2.1 Discovery Object

The Discovery data object type provides an interface to identify the data object types supported by the mailbox.

Mailbox Discovery Request					
Byte 3	Byte 2	Byte 1	Byte 0	DWORD	Section
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
Reserved	Data Type (00H)	Vendor ID (0001H)		0	Mailbox Header
Reserved	Length (00003H)			1	
Reserved			Index	2	Data

Index – 8-bit index to query for discovery. Software should start with index 00H and incrementally query each index to discover the supported object types until no additional object types are indicated.

Mailbox Discovery Response					
Byte 3	Byte 2	Byte 1	Byte 0	DWORD	Section
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
Reserved	Data Type (00H)	Vendor ID (0001H)		0	Mailbox Header
Reserved	Length (00003H)			1	
Next Index	Data Object Type	Vendor ID		2	Data

Vendor ID – 16-bit Vendor ID of the enumerated protocol. If the requested index is invalid or out of range, the Vendor ID in the response must be FFFFH.

Data Object Type – 8-bit Data Object Type of the enumerated protocol. If the Vendor ID value in the response is FFFFH, then this field is undefined. The Discovery protocol itself is enumerated at index 0.

Next Index – 8-bit index of the next available protocol for enumeration. If additional protocols are present beyond the requested index, the response returns the index provided in the request incremented by 1. If there are no additional protocols to enumerate, this value will be 00H. If the Vendor ID value in the response is FFFFH, then this field is undefined.

To enumerate the protocols supported by the MCU staging mailbox, software should send a Discovery request with index equal to 00H. If the next index field of the response is non-zero, software should increment the index and send another Discovery request, and so on, until the next index field of the response is zero, indicating no additional protocols to enumerate.

5.2.2 MCU Staging Object

The MCU Staging object provides multiple commands to facilitate staging of the microcode update. Each MCU Staging command object consists of a mailbox header followed an 8-byte MCU Staging command header and zero or more DWORDs of command data. The specified command determines the layout of the command data.

MCU Staging Command Format					
Byte 3	Byte 2	Byte 1	Byte 0	DWORD	Section
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
Reserved	Data Type (0BH)	Vendor ID (8086H)		0	Mailbox Header
Reserved		Length (variable)		1	
Parameters[1]	Parameters[0]	Sub-Command	Command	2	Command Header
Parameters[5]	Parameters[4]	Parameters[3]	Parameters[2]	3	
Data Object DWORD 0				4	Command Data
…				…	
Data Object DWORD N				N+4	

Command – 8-bit index of the MCU Staging command to execute. The MCU Staging object supports the following commands:

Command Index	Description
0	**MCU Staging Discovery** – Returns various parameters of the MCU Staging interface.
1	**Get Staged MCU Revision** – Returns the Revision ID of the currently staged MCU, if available.
2	Reserved for future use. Response is undefined
3	**MCU Load and Verify** – Loads and verifies one page of MCU data into the internal staging buffers.
4-255	Reserved for future use. Reponses are undefined

Sub-Command – The definition of the 8-bit sub-command field depends on the command.

Parameters[x] – The definition of the six 8-bit parameter fields depend on the command.

5.2.2.1 MCU Staging Discovery

The MCU Staging Discovery command returns various information about the MCU Staging capabilities. This command does not support any sub-commands or parameters; those fields are reserved and must be zero.

MCU Staging Discovery Request																																	
Byte 3								Byte 2								Byte 1								Byte 0								DWORD	Section
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Reserved								Data Type (0BH)								Vendor ID (8086H)																0	Mailbox Header
Reserved														Length (00004H)																		1	
Reserved																								Command (00H)								2	Command Header
Reserved																																3	

MCU Staging Discovery Response																																	
Byte 3								Byte 2								Byte 1								Byte 0								DWORD	Section
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Reserved								Data Type (0BH)								Vendor ID (8086H)																0	Mailbox Header
Reserved														Length (00005H)																		1	
Reserved																Major Revision								Minor Revision								2	Response Data
Mailbox Size																																3	
Maximum Page Size																																4	

Major/Minor Revision – 16-bit revision of the MCU staging mailbox interface. Expected to be xx.yy for the initial implementation.

Mailbox Size – 32-bit size in bytes of the MCU staging mailbox interface. Will be 4,112 bytes on GNR to accommodate 8 bytes for the mailbox header, 8 bytes for the MCU Staging command header, and up to 4,096 bytes of MCU data.

Maximum Page Size – 32-bit size in bytes of the maximum MCU data payload. Will be 4,096 bytes on GNR. The MCU will be transferred incrementally to the staging buffers in chunks of up to 4,096 bytes each.

5.2.2.2 Get Staged MCU Revision

The Get Staged MCU Revision command returns the Revision ID of the currently staged MCU, if available, and status information about the MCU Staging interface. This command does not support any sub-commands or parameters; those fields are reserved and must be zero.

Get Staged MCU Revision Request

Byte 3								Byte 2								Byte 1								Byte 0								DWORD	Section
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Reserved								Data Type (0BH)								Vendor ID (8086H)																0	Mailbox Header
Reserved														Length (00004H)																		1	
Reserved																								Command (01H)								2	Command Header
Reserved																																3	

Get Staged MCU Revision Response

Byte 3								Byte 2								Byte 1								Byte 0								DWORD	Section
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Reserved								Data Type (0BH)								Vendor ID (8086H)																0	Mailbox Header
Reserved														Length (00005H)																		1	
MCU Revision ID																																2	Response Data
Flags																																3	
Reserved																SVN																4	

MCU Revision ID – 32-bit Revision ID of the currently staged MCU. If an MCU is successfully staged, then this returns the Revision ID of the staged MCU. If no MCU is currently staged or staging is not complete, then this field returns 0.

Flags – 32-bit flags indicating the status of the MCU Staging hardware.

- **Bit 0:** When set to 1, indicates that the MCU Revision ID specified in the response is valid, and that staging has completed successfully.
- **Bit 1:** When set to 1, indicates that MCU staging is currently in progress, i.e., at least one data chunk of MCU data has been received, but staging is not yet complete.
- **Bit 2:** When set to 1, indicates an error has been detected, and the staging process must be restarted.
- Bits 3 through 31 are reserved for future use.

SVN – 16-bit Secure Version Number of the currently staged MCU.

5.2.2.3 MCU Load and Verify

The MCU Load and Verify command is used to stage the MCU data. The data is staged in chunks up to the Maximum Page Size specified by the MCU Staging Discovery response. To begin staging an update, software sends an MCU Load and Verify command with the sub-command set to 01H and no MCU data payload to initialize the staging hardware. The staging hardware responds with the offset of the first chunk to stage. The offset is specified in bytes from the start of the microcode update header. Software then sends MCU Load and Verify commands with the sub-command set to 00H and MCU data up to the Maximum Page Size to stage each subsequent chunk according to the offset specified in each response. Note that the responses may specify chunks out of sequential order, and portions of the update that do not need to be staged may be skipped altogether. The staging hardware solely determines which chunks must be staged and in what order. Failure to stage the chunks as requested by the hardware may cause staging to fail. This command does not support any parameters; those fields are reserved and must be zero.

MCU Load and Verify Request

Byte 3								Byte 2								Byte 1								Byte 0								DWORD	Section
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Reserved								Data Type (0BH)								Vendor ID (8086H)																0	Mailbox Header
Reserved														Length (variable)																		1	
Reserved																Sub-Command								Command (03H)								2	Command Header
Reserved																																3	
MCU Data																																4	Command Data
																																...	
																																N	

Sub-Command – The MCU Load and Verify command supports two sub-commands:

- Sub-command 00H is used for normal operation to load one chunk of update data into the staging buffers.
- Sub-command 01H is used to initialize the MCU staging hardware prior to loading the first chunk of MCU data, and to reset the hardware in the event of any errors. When sending an MCU Load and Verify command with sub-command equal to 01H, software should not include any data with the request.

MCU Data – For sub-command 00H, one chunk of MCU data, up to the Maximum Page Size specified by the MCU Staging Discovery object (4,096 bytes on GNR). No data should be sent with sub-command 01H.

MCU Load and Verify Response

Byte 3								Byte 2								Byte 1								Byte 0								DWORD	Section
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
Reserved								Data Type (0BH)								Vendor ID (8086H)																0	Mailbox Header
Reserved														Length (00004H)																		1	
Next Chunk Offset																																2	Response Data
Flags																																3	

Next Chunk Offset– 32-bit offset of the next chunk to stage. After each chunk of update data is written to the staging buffers, the staging hardware responds with the offset of the next chunk to stage. The offset is specified in bytes from the start of the microcode update header. If no additional chunks are required, this field will return 0xFFFFFFFF. If software provides different chunks than indicated by the response, the update data will eventually fail verification, and the staging hardware will indicate an error via bit 2 of the flags field.

Flags – 32-bit flags indicating the status of the staging hardware after each chunk is processed.

- **Bit 0:** When set to 1, indicates that staging and verification of all required chunks has completed successfully.
- **Bit 1:** When set to 1, indicates that MCU staging is currently in progress, i.e., at least one data chunk of MCU data has been received, but more chunks must be delivered before staging and verification can complete.
- **Bit 2:** When set to 1, indicates an error has been detected, and the staging process must be restarted. Software must issue an MCU Load and Verify command with sub-command equal to 01H to reset the staging hardware and restart staging.
- Bits 3 through 31 are reserved for future use.

5.3 MCU Staging Mailbox Interface Registers

The interface to the staging hardware is provided by a set of four memory mapped I/O (MMIO) registers within the platform physical address space. The IA32_MCU_STAGING_MBOX_ADDR MSR provides the platform physical address of the first mailbox register (MCU_STAGING_CTRL).

There is one set of mailbox registers and internal staging buffers per physical processor package. Therefore, the IA32_MCU_STAGING_MBOX_ADDR MSR is package scoped, and in a multi-socket system, will provide a different physical address on each physical package.

IA32_MCU_STAGING_MBOX_ADDR (MSR 7A5H) – Read-only – Package Scoped		
Bit Field	Name	Description
63:0	MCU_STAGING_MBOX_ADDR	Indicates the MMIO platform physical address of the MCU Staging Control register. If the value is 0, then the mailbox has not been configured by platform firmware (BIOS).

The remaining mailbox registers are located at 4-byte offsets from the MCU_STAGING_CTRL register.

Byte Offset	Register	Description
+00h	MCU_STAGING_CTRL	Control Register
+04h	MCU_STAGING_STATUS	Status Register
+08h	MCU_STAGING_WRITE_DATA	Write Data Register
+0Ch	MCU_STAGING_READ_DATA	Read Data Register

5.3.1 MCU Staging Control Register

The MCU Staging Control register (MCU_STAGING_CTRL) is written by software to control the MCU staging interface.

MCU_STAGING_CTRL (offset 0x00) – Read/write – Package Scoped		
Bit Field	Name	Description
0	ABORT	(RW) Writing 1 to this bit causes all data object transfer operations associated with this MCU staging mailbox instance to be aborted. Always read as 0.
28:1	Reserved	Must be 0.
29	MBOX_EN	(RO) When 1, indicates that the mailbox interface is enabled. When 0, indicates that the mailbox interface has not been properly enabled by system firmware (BIOS). Writes to this bit are ignored.
30	MAILBOX_DATA_READY	(RW) During data writes, software sets this bit to indicate that a DWORD has been written to the MCU_STAGING_WRITE_DATA register. The processor clears this bit after consuming the DWORD written to the register. During data reads, the processor sets this bit to indicate that a DWORD is ready to be read from the MCU_STAGING_READ_DATA register. Software must clear this bit after consuming the DWORD read from the register.
31	GO	(RW) Software sets this bit to 1 after writing an entire data object to indicate to the processor to begin processing the object. The processor clears this bit to 0 after it completes processing the object.

5.3.2 MCU Staging Status Register

The MCU Staging Status register (MCU_STAGING_STATUS) is used by software to determine the status of the MCU staging interface.

MCU_STAGING_STATUS (offset 0x04) – Read-only – Package Scoped		
Bit Field	Name	Description
0	BUSY	(RO) When set to 1, indicates that the processor is temporarily unable to receive new data through the MCU_STAGING_WRITE_DATA register.
1	Reserved	Must be 0.
2	ERROR	(RO) When set to 1, indicates that an error has occurred while the processor was handling the last MCU staging mailbox data object.
30:3	Reserved	Must be 0.
31	OBJECT_READY	(RO) When set to 1, indicates that the processor has completed processing the last MCU staging mailbox data object received and a new object is available in response.

5.3.3 MCU Staging Write Data Register

The MCU Staging Write Data register (MCU_STAGING_WRITE_DATA) is written by software to transfer one DWORD of an object to the MCU staging mailbox interface.

MCU_STAGING_WRITE_DATA (offset 0x08) – Read/Write – Package Scoped		
Bit Field	Name	Description
31:0	WRITE_DATA	(RW) Software writes this register to transfer one DWORD of object data into the MCU staging mailbox interface. After writing this register, software must set the MAILBOX_DATA_READY bit in the MCU_STAGING_CTRL register to indicate to the processor that new data is available.

5.3.4 MCU Staging Read Data Register

The MCU Staging Read Data register (MCU_STAGING_READ_DATA) is read by software to transfer one DWORD of an object from the MCU staging mailbox interface.

MCU_STAGING_READ_DATA (offset 0x0C) – Read/Write – Package Scoped		
Bit Field	Name	Description
31:0	READ_DATA	(RW) Software reads this register to transfer one DWORD of object data from the MCU staging mailbox interface. After reading this register, software must clear the MAILBOX_DATA_READY bit in the MCU_STAGING_CTRL register to indicate to the processor that the data has been read.

5.4 MCU Staging Mailbox Message Protocol

System software uses the MCU staging mailbox to load the microcode update into internal staging buffers by sending data object requests to the staging hardware and receiving data object responses from the staging hardware.

During initialization, system software must verify the presence of the mailbox interface via enumeration and determine the maximum supported page size by issuing the MCU Staging Discovery request.

To stage a patch for loading during runtime, software must begin by initializing the staging hardware with an MCU Load and Verify request with sub-command equal to 01H and no data payload. The staging hardware responds with the offset of the first chunk to stage. Software must then issue a sequence of MCU Load and Verify requests with sub-command equal to 00H to incrementally load the microcode update data into the staging buffers in chunks up to the maximum supported page size, beginning from the offset specified in response to the initialization sub-command. After each MCU Load and Verify request, the staging hardware responds with the offset of the next chunk to stage. The offsets specified in response to the MCU Load and Verify commands are specified in bytes from the start of the microcode update header. The staging hardware may not specify chunks in sequential order, and some portions of the update that do not require staging may be skipped altogether. When no additional chunks are required, the staging hardware will respond with an offset of FFFFFFFFH.

After staging all required chunks, system software should issue the Get Staged MCU Revision request to confirm whether the staging was successful.

To send a data object request to the staging hardware, software must first verify that the BUSY bit in MCU_STAGING_STATUS is clear. Then, software sends the entire object one DWORD at a time as described below. After the entire object has been sent (which includes waiting for the MAILBOX_DATA_READY bit to be cleared after sending the last DWORD), software must set the GO bit in the MCU_STAGING_CTRL register to indicate that the object is complete. If the submitted request produces a response, software must then poll the OBJECT_READY bit in the MCU_STAGING_STATUS register until it is set, indicating that the hardware has processed the entire object and is ready to return the response. Software then must read the response one DWORD at a time as described below.

To send one DWORD to the staging hardware, software must write the DWORD into the MCU_STAGING_WRITE_DATA register, then set the MAILBOX_DATA_READY bit in the MCU_STAGING_CTRL register. Software must then poll the MAILBOX_DATA_READY bit in the MCU_STAGING_CTRL register until it is cleared by the staging hardware, indicating that the DWORD has been consumed by the hardware. If software writes another DWORD into the MCU_STAGING_WRITE_DATA register before the MAILBOX_DATA_READY bit is cleared by the hardware, then either DWORD may be dropped and the MCU staging may fail. Note that in some implementations, the MAILBOX_DATA_READY bit may be cleared by hardware immediately after being set by software (software may not observe the bit as set).

To receive one DWORD from the staging hardware, software must poll the MAILBOX_DATA_READY bit in the MCU_STAGING_CTRL register until it is set by the staging hardware, indicating that a DWORD has been populated by the hardware into the MCU_STAGING_READ_DATA register. Software must then read the DWORD from the MCU_STAGING_READ_DATA register and clear the MAILBOX_DATA_READY bit in the MCU_STAGING_CTRL register, indicating that the DWORD has been consumed by the software.