P6 Family Processor Microcode Update Feature Review

First Draft

Jesus Molina, William Arbaugh

College Park, December 2000

ABSTRACT

The P6 family processors have the capability to correct specific errata through the loading of an Intel−supplied data block. This feature is highly obscure and undocumented. This document tries to explains how the microcode updates are composed, what errata they can fix and how they can be signed or encrypted.

1.Introduction

The microcode update feature is described in chapter 8 section 10 of [3]. The microcode updates are composed of 2048 bytes of data, 48 bytes of which are the header and the remaining 2000 are the microcode itself. For avoiding any confusion in this paper we will call the 2048 bytes the microcode update or just the microcode, the first 48 bytes the header and the next 2000 bytes the data or the update data. This update data is intended to fix a deviation from the specifications of a processor (also called errata by Intel. We will use the term bug and errata in this document to refer to this deviations). Further explanation of the composition of the header and the data will be held in section 3 of this document. The microcode is uploaded using a software provided by Intel and flashes on reboot, that is, you have to upload it again every time you reboot the computer.

Although this feature is described by Intel, the description of the microcode updates feature is vague in stating the erratas that the microcode can solve. Also, the microcodes are signed (as we will describe more in depth in section 7, and as stated in Chapter 8.10 of [3]) but as this is stated in [3] it is not said how this encryption/signature is performed.

Alexander Wolfe in his article in eetimes [12] and later in another article in byte.com [11] stated that "knowledge sources report that the data block is mapped directly (or, more precisely, after decryption) to the microcode itself. That is, the decrypted data block contains specific microinstructions. The microinstructions, which are the lowest−level primitives within a microprocessor, control the specific actions opening and closing of gates and the sequence of actions that make up an instruction". In this document we are not covering the microinstruction themselves. However, it seems obvious that the microinstructions are the last level between the processor and the user. And if the microcodes can patch bugs inside the processor, it also seems that adding bugs will be likely. But as the microinstructions inside the microcode are completely obscure, and the updating space is so tiny, creating a complicated algorithm seems really difficult.

This document is divided in to seven sections including this introduction. In section 2 we will describe the MSR registers and the CPUID instruction, a background necessary to understand how the microcode driver works. In section 3 we will describe the basic structure of the microcode updates. In section 4 we will describe the driver, as described in [3], and the changes done in this basic structure to perform the tests. In section three we will discuss the microcode structure. In section 5 we will state some errata solved by the microcodes. In section 6 we will talk about the tests we have done, and what could be the possible encryption systems used by Intel to sign the microcode updates. In section 7 we will enumerate all the future work that needs to be done to profile better this feature and also we will give the conclusion and the location of the drivers and documents related to the project.

Finally, all the work in this document has been done using the drivers and microcode provided by Tigran Aviazan and Simon Erzen.[15] The kernel used was Linux 2.4.0−test10.

2.Background

2.1 CPUID instruction

The CPUID instruction was introduced by Intel in the Pentium® processor. Since then it became the official method of identification for Pentium® class and newer chips. It is also supported by newer 486−class chips made by Intel, AMD, UMC and Cyrix, and newer NexGen CPUs.
Intel provides a straightforward method for determining whether the processor's internal architecture is able to execute the CPUID instruction. This method uses the ID flag in bit 21 of the EFLAGS register. If the value of this flag can be changed by software, the CPUID instruction is executable.
Before executing CPUID, you have to store a value in the EAX register. This will give CPUID instruction the mode of operation. Table 2.1 shows the modes of operation of CPUID, and the results.

Table 2.1. Effects of EAX Contents on CPUID Instruction Output

Parameter	Outputs of CPUID
EAX=0	EAX ← Highest value recognized by CPUID instruction
	EBX:EDX:ECX ← Vendor identification string
EAX=1	EAX ← Processor signature
	EDX ← Feature flags
	EBX:ECX ← Intel reserved (Do not use.)
EAX=2	EAX:EBX:ECX:EDX ← Processor configuration parameters
3≤EAX≤Highest Value	Intel reserved
EAX> highest value	EAX:EBX:ECX:EDX ← Undefined (Do not use.)

In the P6 family, the Vendor ID string will be GenuineIntel.
The processor signature is encoded as follow:

EAX register

Reserved	Type	Family	Model	Stepping
32 14	12	8	4	0

The values returned by the CPUID instructions are encoded as table 2.2 show for the P6 family. All the values are in decimal. The first value is the Family (P6 family), the second value is the model and the last value is the stepping. The microcodes follow the same encoding as the one returned by CPUID.

Table 2.2 CPUID values for the P6 family

6 − P6 family.

0 − Pentium Pro A step samples

1 − regular Pentium Pro

1 − B0
2 − C0
6 − sA0
7 − sA1
9 − sB1

3 − Pentium II "Klamath" (063x), 233..333 MHz, P6 overdrive with MMX (163x)

2 − tdB0 (1632) − P II overdrive for PPro
3 − C0
4 − C1

4 − P55CT (overdrive for P54)

5 − Pentium II "Deschutes" (266..450 MHz), Pentium II Xeon (400.. MHz) and Celeron w/o L2 cache (266, 300MHz) (Celeron with no L2 cache returns the appropriate info via CPUID 2)

0 − dA0
1 − dA1
2 − dB0
3 − dB1

6 − Celeron "A" "Mendocino" (w/ 128KB full speed L2 cache) or Pentium II PE (256 KB cache)

0 − mA0
5 −
a −

7 − Pentium III "Katmai"

2 − kb0
3 − kC0

8 − Pentium III "Coppermine", Celeron w/SSE

1 − Ca2
3 − Cb0
6 − CC0

a − Pentium III Xeon "Cascades"

f − Pentium 4 ("Willamette") family

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.
For more information about the CPUID instruction, refer to [2], [7],[8],[14].

2.2MSR registers

The P6 family processors and Pentium® processors contain model−specific registers (MSRs). The MSR registers were highly undocumented, but as they were referenced in the Intels Manual as Appendix H, the information these references give made possible to reverse engineer the information about this MSR registers [13]. Finally, Intel released the documentation about the MSR registers in [3], chapters 8.2, 15 and appendix A. There is still some issues about them. For example, registers from 80000000h to FFFFFFFFh were not supposed to be available. But due to an errata, more precisely, errata E29, we can perform WRMSR and RDMSR in invalid addresses[4]. This bug is not intended to be fixed. The workaround is not to use invalid addresses. More information about this bug can be found in [9].
These registers are by definition implementation specific; that is, they are not guaranteed to be supported on future Intel Architecture processors and/or to have the same functions. The MSRs are provided to control a variety of hardware− and software−related features, including Performance−monitoring counters, Debug extensions and miscellaneous services.The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively. This instruction are explained in [2].
The microcode update feature uses two of this registers to perform the updates, as explained in 2.1.
A complete list of the documented MSR registers can be found in [10].

3.Microcode structure

As we have stated in the introduction, the microcode is composed of a header and the data.

In table 3.1 we can see the structure of the microcode. The header, as we can see, is divided in different fields. The update data is shaded. A quick explanation of the fields is in table 3.2.

Table 3.1 Microcode Update Structure

Field				Bytes
HEADER VERSION				***00H***
UPDATE VERSION				***04H***
DATE				***08H***
Month	***Day***	***Year***		
PROCESOR				***0CH***
Reserved	***Family***	***Model***	***Stepping***	
CHECKSUM				***10H***
LOADER REVISION				***14H***
RESERVED				***18H***
DATA				***30H*** ***7FCH***
24	16	8	0	***bit***

Table 3.2 Field descriptors

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.

Field Name	Offset (in bytes)	Length (in bytes)	Description
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.

The fields in the header are checked in the driver. As they are used only for identification and checks done by the driver (and so, we can skip them) we are going to concentrate on the structure of the data.
Our basic start points are the 55 microcodes provided by Intel., that we can divide in 5 microcode updates for Pentium® Pro, 20 for Pentium® II (Klamatch, Deschutes), 8 for Pentium® Celeron, 20 for Pentium® III (Coppermine, Katmai) and 2 for Pentium® III Xeon Cascades. Comparing the structure of the data, that is, comparing each microcode with the others, we get the following results. The comparison is done byte by byte. When we say that a microcode data does not share any data with the others, we are not including random collisions. The average number of collisions is 1 in every 256 bytes. So in a microcode there is an average of about 8 collisions.

−The 4 Pentium® Pro update data is different from all the other microcodes, but between them they share the last 1136 bytes. The Pentium® Pro stepping B0 (0611 CPUID encoding) does not share any bytes with the other microcodes.

−The 42 Pentium® II and Pentium® III update data share the last 1056 Bytes. They differ completely from the Celeron and Pentium® Pro microcodes.

−The 8 Celeron updates differ completely from all the other microcode update data.

We will discuss these results in section six.

4.Driver Specifications

The driver, as stated in [3], works as follow.

- Check the header. This involves choosing the right microcode for your processor/stepping, reject the microcode if it does not match with your processor stepping, reject the microcode if you are trying to update a microcode with revision number lower than the uploaded microcode, reject the microcode if the checksum is not correct (as we have seen in section two, the checksum is just a check of the sum of the microcode, that should be zero).
- Initialize the registers to call WRMSR
    - EAX contains the linear address of the start of the Update Data
    - EDX contains zero
    - ECX contains 79h
- Call WRMSR (We have updated the microcode data)
- Initialize the registers to call WRMSR
    - EAX contains zero
    - EDX contains zero
    - ECX contains 8bh
- Call WRMSR (We have cleared the data in the MSR register 8bh)
- Initialize the registers to call CPUID
    - EAX contains 1
- CPUID
- Initialize the registers to call RDMSR
    - ECX contains 8bh
- Call RDMSR

If no microcode update has been loaded, the MSR remains unchanged (as we have cleared it to 0 before using CPUID we can test if the updates have been loaded). If the update has succeeded the low 32 bits of this register contains the standard model/stepping information, while the high 32 bits contains the microcode update ID.

The data is provided by Intel in text based files, so we must "translate" these files to a binary before uploading the microcode.
For performing our tests we have changed the driver slightly. Our driver does not perform any check on the header, and relies completely in the check of the RDMSR in 8bh after the call to CPUID to know wether the microcode has been uploaded or not. This way we do not have to worry about changing fields in the header when trying to upload a microcode that belongs to other model/stepping or forged microcodes.

5.Errata solved by the microcodes.

A interesting question and a key to studying the microcodes is what errata (bugs) in the processor are solved by uploading the microcode; in other words, what bug is solved by each microcode. Intel does not give any kind of information on this matter.
Intel releases documents named Processor Specification Updates for each of their processors. These documents [4],[5],[6] contain the changes Intel made from their specifications, and all the bugs appeared are listed (or at least, that Intel want to document) with a description of the problem, the implication this problem could have in the processor, the workaround for this problem and the actual status of this problem (**Fix** if it is intended to be fixed, **Fixed** if it has been fixed in some way, **NoFix** if there are no plans to fix the bug).

In some erratas Intel states, "It is possible for BIOS code to contain a workaround for this errata". At first sight this seems to be the erratas fixed by the microcodes.
Surprisingly, there are more microcodes updates than erratas. For example, in the Intel Pentium® III Processor Specification Update, Released Novemeber 2000, Errata E34, E35, E44, E46, E54, E56, E58, E64 are stated as having been fixed by a BIOS workaround. That is, 8 erratas. The erratas refering to the Coppermine processor, stepping Cb0 (CPUID 0683) are E44, E46, E54, E58, E64 and there are at least 5 microcodes. However, for the Coppermine processor stepping Ca2 (CPUID 0681) we have 4 erratas stated as solvable with a BIOS workaround (E44, E46, E58, E64) and we can find at least 5 microcodes. Also, for the Coppermine processor stepping CC0 (CPUID 0686) we find only one errata, E46, and there are at least three different microcodes (Released almost on the same date, one on 05/04/00 and two on 05/05/00). Why the need for three different microcodes for solving only one errata? The answer seems to be that either some of the known bugs in the processor are not documented by Intel or some errata are solved with the microcode update and are not stated as solvable with a BIOS workaround in the specifications updates. Or maybe both. In fact, E19 and E20 for the Pentium® II processor are solved using a microcode update, but only E20 is stated as solvable with a BIOS workaround.
Finally, the way that Intel handles this Bios workarounds is not clear. In the Pentium® III processor erratas E34, E35, E54 are marked as "Fixed", while erratas E56, E44, E46, E58, E64 are marked as "NoFix". It is not clear if Intel assume the microcode updates are a valid Fix for the erratas.
A complete listing of the erratas solved using a BIOS workaround and microcodes that can probably solve the errata can be found in the webpage of this project, refer to section 7.

6.Microcode Encryption System Analysis

Analyzing the encryption system of the microcodes is comparable to trying to understand a complete random message written in a foreign language.
However, we have some hints that can be exploited.

- Microcodes created for Pentium® Pro and microcodes created for Pentium® II and Pentium® III share some data between them. This can be because they share patches for the bugs, or because they uses this piece of code as the loader of the microcode itself. This way this loader will not have to be in the processor. Celeron, however, as it has a different design, will have the loader resident in the processor itself.
- Inside the microcode has to be information about the model/stepping and about the microcode version, as it is returned by the CPUID instruction. This information is provided either by the microcode data or by the encryption system itself. (For example, using a different key for each processor stepping).
- The processor rejects forged microcodes.
- The processor rejects microcode intended for a model/stepping different than the processor model/stepping.
- The newest microcode are likely to include the microinstructions used to patch the bugs that the oldest microcode patched.
- The encryption system should not be a really complicated algorithm, so it will not waste much die space. However this is only an assumption.

The two most likely ways to sign the microcode are

- Encrypt all the microcode or part of it. A CRC value will be inside the encrypted data and will be checked by the processor after decrypting the data.
- Use a signature, like a CRC or a MAC.

Assume Intel Encrypts all the microcodes. The choices are:

- Encrypt all microcodes with the same key.
- Encrypt each microcode depending of the processor stepping, using a different key for each processor/stepping.

If Intel is encrypting the microcodes with the same key, there should be more sharing between the microcodes when we perform the by byte comparison. The only way that this sharing could be avoided is not respecting any order with the microcode instruction. A convolution between two blocks of data will give us several peeks then, reflecting repeated patterns. Also, if they are not encrypting the data, but only using a MAC or a CRC, there has to be some matching between microcodes, and a frequency analysis will give interesting results.
If Intel is using a different key for each model/stepping, there should be some repeated patterns between the microcodes intended for the same model/stepping. We have not yet found any repeated pattern between microcode data intended for the same model/stepping. So another confounder mechanism must be used, possibly using the microcode version (that in this scheme, must be finite) as a confounder. In this scheme, the last part shared by all the microcodes should be definitely a loader.

7.Future work and Conclusions

The work that has been already done is:

- Performing minor changes to the driver released by Tigran Aviazan and Simon Erzen, so it will not check the header.
- Convert the microcode updates in binary files, and sort them by processor/stepping.
- Perform tests on the data.
- Perform rejection tests on the data.
- Search in the Pentium® Specification Updates for the bugs that can be solved by using a BIOS workaround.

The future work need to be done is:

- Frequency analysis on the data.
- Search for repeated patterns between microcodes that are not in order.
- Correlate the microcodes with the erratas.
- Profile the CPU behavior when a microcode is uploaded using the MSR registers.

All the tools and further information can be found of the webpage for the project.

http://www.jesusmolina.com/projects/microcode

or on the mirror site

http://wam.umd.edu/~chus/microcode

8.References

1. IA−32 Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture. Intel Corp
2. IA−32 Intel® Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual. Intel Corp
3. IA−32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide. Intel Corp
4. Pentium® III Processor Specification Update. Intel Corp
5. Pentium® II Processor Specification Update. Intel Corp
6. Pentium® Pro Processor Specification Update. Intel Corp
7. Intel Processor Identification and the CPUID Instruction. Intel Corp, April 1998, AP−485:Application note
8. Intel® Pentium® 4 Processor Identification an the CPUID Instruction. Intel Corp Revision 001, July 2000
9. "Pentium Model−Specific Registers and What They Reveal". Ralf Brown, Revision 1.0, October 11, 1995 http://x86.org/articles/p5msr/pentiummsrs.htm
10. "Model−Specific Registers",Release 60. Ralf Brown, January1999 http://chip.ms.mff.cuni.cz/~pcguts/cpu/msr.txt
11. "Intel Sneaks In Software−Upgrade Feature". Alexander Wolfe, October 2000 http://www.byte.com/column/BYT20001016S0006
12. "Intel preps plan to bust bugs in Pentium MPUs". Alexander Wolfe, EETimes, Issue 960, January 1997
13. "The Pentium collects lots of information about code execution, and now you can get access to it". Terje Mathisen http://www.byte.com/art/9407/sec12/art3.htm
14. "Identification of x86 CPUs with CPUID support", Grzegorz Mazur http://grafi.ii.pw.edu.pl/gbm/x86/cpuid.html
15. "Intel P6 Microcode Update Utility for Linux", Simon Erzen, Tigran Aviazan http://www.urbanmyth.org/microcode/