AP-485

APPLICATION NOTE

Intel Processor Identification With the CPUID Instruction

Order Number: 241618-004

Revision	Revision History	Date
-001	Original Issue.	05/93
-002	Modified Table 2. Intel486™ and Pentium® Processor Signatures	10/93
-003	Updated to accommodate new processor versions. Program examples modified for ease of use, section added discussing BIOS recognition for OverDrive® processors, and feature flag information updated.	09/94
-004	Updated with Pentium Pro and OverDrive processors information. Modified Table 2 and 8. Inserted Tables 5, 6, 7. Inserted section 3.4.	10/95

Additional copies of this document or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641
or call 1-800-879-4683

1.0 INTRODUCTION

As the Intel Architecture evolves, with the addition of new generations and models of processors (8086, 8088, Intel 286, Intel386 ™, Intel486™, Pentium® processors, and Pentium® Pro processors), it is essential that Intel provides an increasingly sophisticated means with which software can identify the features available on each processor. This identification mechanism has evolved in conjunction with the Intel Architecture as follows:

- Originally, Intel published code sequences that could detect minor implementation differences to identify processor generations.
- Later, with the advent of the Intel386 processor, Intel implemented processor signature identification, which provided the processor family, model, and stepping numbers to software at reset.
- As the Intel Architecture evolved, Intel extended the processor signature identification into the CPUID instruction. The CPUID instruction not only provides the processor signature, but also provides information about the features supported by and implemented on the Intel processor.

The evolution of processor identification was necessary because as the Intel Architecture proliferates, the computing market must be able to tune processor functionality across processor generations and models that have differing sets of features. Anticipating that this trend will continue with future processor generations, the Intel Architecture implementation of the CPUID instruction is extensible.

This Application Note explains how to use the CPUID instruction in software applications, BIOS implementations, and tools. By taking advantage of the CPUID instruction, software developers can create software applications and tools that can execute compatibly across the widest range of Intel processor generations and models, past, present, and future.

1.1 Update Support

You can obtain new Intel processor signature and feature bits information from the user's manual, programmer's reference manual or appropriate documentation for a processor. In addition, you can receive updated versions of the programming examples included in this application note; contact your Intel representative for more information.

2.0 DETECTING THE CPUID INSTRUCTION

Intel provides a straightforward method for detecting whether the CPUID instruction is available. This method uses the ID flag in bit 21 of the EFLAGS register. If software can change the value of this flag, the CPUID instruction is available. The program examples at the end of this Application Note show how you use the PUSHFD instruction to read and the POPFD instruction to change the value of the ID flag.

3.0 OUTPUTS OF THE CPUID INSTRUCTION

Figure 1 summarizes the outputs of the CPUID instruction.

The CPUID instruction can be executed multiple times, and each execution can have a different parameter value in the EAX register. The output depends on the value in the EAX register, as specified in Table 1. If you want the CPUID instruction to determine the highest acceptable value in the EAX register, the program should set the EAX register parameter value to 0. Always use a parameter value that is less than or equal to this highest returned value.

3.1 Vendor-ID String

If the EAX register contains a v alue of 0, the CPUID instruction returns the vendor identification string in the EBX, EDX, and ECX registers. These registers contain the ASCII string `GenuineIntel` .

While any imitator of the Intel Architecture can provide the CPUID instruction, no imitator can legitimately claim that its part is a genuine Intel part. So the presence of the `GenuineIntel` string is an assurance that the CPUID instruction and the processor signature are implemented as described in this document.

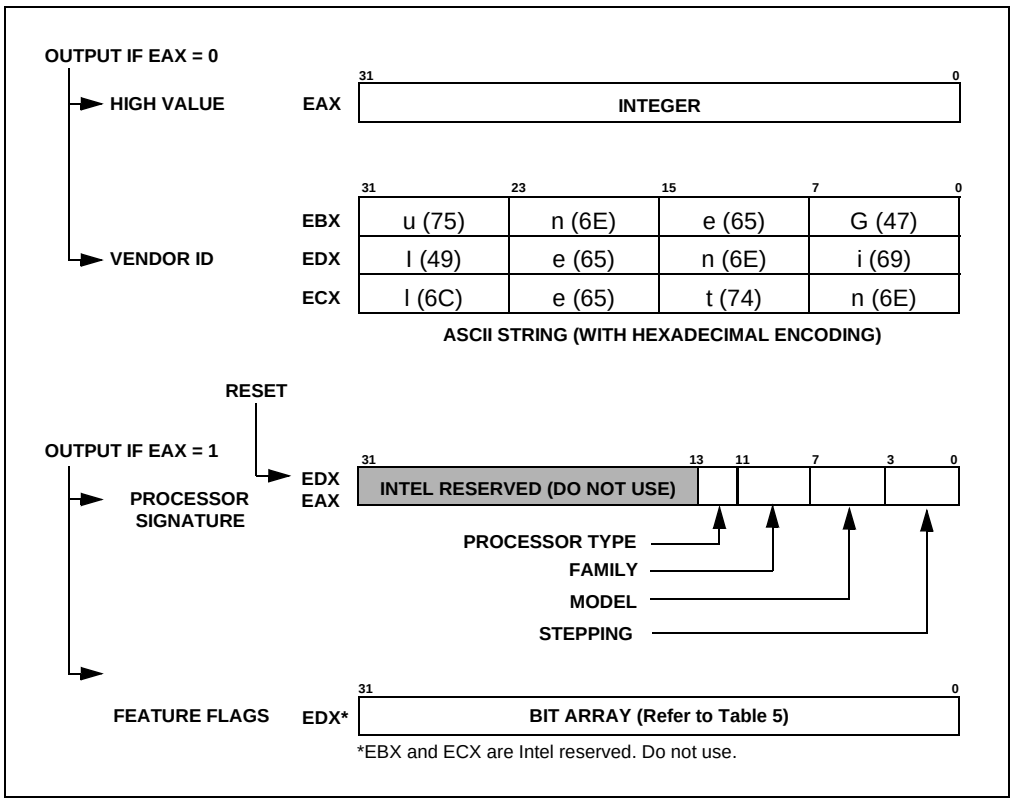


Figure 1. CPUID Instruction Outputs

Table 1. Effects of EAX Contents on CPUID Instruction Output

Parameter	Outputs of CPUID
EAX = 0	EAX ← Highest value recognized
	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 cache parameters
3 ≤ EAX ≤ highest value	Intel reserved
EAX > highest value	EAX:EBX:ECX:EDX ← Undefined (Do not use.)

Table 2. Processor Type

Bit Position	Value	Description
13,12	00	Original OEM processor
	01	OverDrive® Processor
	10	Dual processor
	11	Intel reserved. (Do not use.)

3.2 Processor Signature

Beginning with the Intel386 processor family, the processor can return a signature at reset. With processors that implement the CPUID instruction, they return the processor signature when they either run the CPUID instruction, or at reset. Figure 1 shows the format of the signature for the Intel486 and Pentium processor families. Table 3 shows the values currently defined for these processors. (The high-order 18 bits are undefined and re served.)

The processor type, specified in bits 12 and 13, indicates whether the processor is an original OEM processor, an OverDrive ® processor, or a dual processo r (capable of being used in a dual processor system). Table 2 shows the processor type valu es returned in bits 12 and 13 of the EAX register.

The family, specified in bits 8 through 11, indicates whether the processor belongs to the Intel386, Intel486, Pentium or Pentium Pro family of processors.

The model number, specified in bits 4 though 7, indicates the processor's family model number, while the stepping number in bits 0 through 3 indicates the revision number of that model.

Older ver sions of Intel486 SX, Intel486 DX and IntelDX2 processors do not support the CPUID instruction, so they can only return the processor signature at reset. Refer to Table 3 to determine which processors support the CPUID instructio n.

Figure 2 sho ws the format of the processor signature for Intel386 processors, which are different from other processors. Table 4 shows the values currently defined for these processors .

Table 3. Intel486™, Pentium® Processor, and Pentium® Pro Processor Signatures

Type	Family	Model	Stepping	Description
00	0100	0000 and 0001	xxxx	Intel486™ DX Processors[1]
00	0100	0010	xxxx	Intel486™ SX Processors[1]
00	0100	0011	xxxx	Intel487™ Processors[1]
00	0100	0011	xxxx	IntelDX2™ Processors[1]
00	0100	0011	xxxx	IntelDX2™ OverDrive® Processors
00	0100	0100	xxxx	Intel486™ SL Processor[1]
00	0100	0101	xxxx	IntelSX2™ Processors
00	0100	0111	xxxx	Write-Back Enhanced IntelDX2™ Processors
00	0100	1000	xxxx	IntelDX4™ Processors
00, 01	0100	1000	xxxx	IntelDX4™ OverDrive® Processors
00	0101	0001	xxxx[2]	Pentium® Processors (510\60, 567\66)
00	0101	0010	xxxx[2]	Pentium® Processors (735\90, 815\100, 1110\133)
01	0101	0010	xxxx[2]	Pentium® OverDrive® Processor for Pentium® Processor (510\60, 567\66)
01	0101	0010	xxxx[2]	Pentium® OverDrive® Processor for Pentium® Processor (680\75, 735\90, 815\100, 1000\120, 1100\133)
01	0101	0011	xxxx[2]	Pentium® OverDrive® Processors for Intel486™ CPU-based systems
01	0101	0100	xxxx[2]	Reserved for a future OverDrive® processor for Pentium Processor (680\75, 735\90, 815\100, 1000\120, 1110\133)
01	0101	0101	xxxx[2]	Pentium® OverDrive® Processor for IntelDX4™ Processor
00	0110	0001	xxxx[2]	Pentium® Pro Processor
01	0110	0011	xxxx	Reserved for a future OverDrive® Processor for Pentium® Pro Processor

1. This processor does not implement the CPUID instruction.
2. Refer to the Pentium® Processor Specifications Update (Order number: 242480), or the Pentium® Pro Specifications Update (Order number: TBD) for the latest list of stepping numbers.

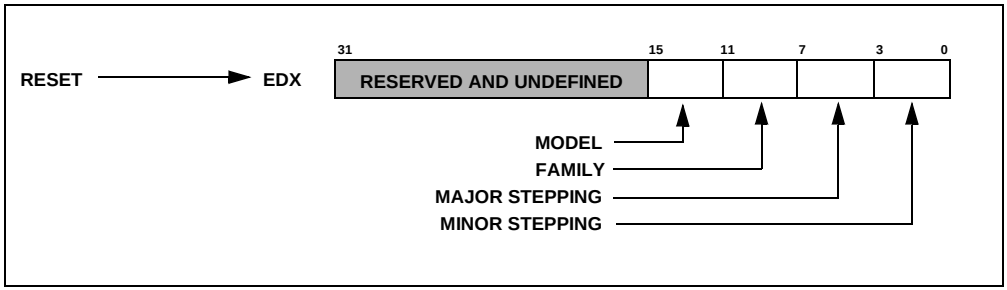


Figure 2. Processor Signature Format on Intel386 ™ Processors

Table 4. Intel386™ Processor Signatures

Model	Family	Major Stepping	Minor Stepping[1]	Description
0000	0011	0000	xxxx	Intel386™ DX Processor
0010	0011	0000	xxxx	Intel386™ SX Processor
0010	0011	0000	xxxx	Intel386™ CX Processor
0010	0011	0000	xxxx	Intel386™ EX Processor
0100	0011	0000 and 0001	xxxx	Intel386™ SL Processor
0000	0011	0100	xxxx	RAPIDCAD™ Processor

1. Intel releases information about minor stepping numbers as needed.

3.3 Feature Flags

When the EAX register contains a value of 1, the CPUID instruction loads the EDX register with the feature flags. The feature flags indicate which features the processor supports. However, in future feature flags, a value of one may indicate a feature is not present . Table 5 lists the currently defined feature flag values. For future processors, refer to the programmer's reference manual, user's manual, or the appropriate documentation for the latest feature flag values.

Use the feature flags in your applications to determine which processor features are supported. By using the CPUID feature flags to predetermine processor features, your software can detect and a void incompatibilities.

Table 5. Feature Flag Values

Bit	Name	Description When Flag = 1	Comments
0	FPU	Floating-point unit on-chip	The processor contains an FPU that supports the Intel 387 floating-point instruction set.
1	VME	Virtual Mode Extension	The processor supports extensions to virtual-8086 mode.
2		Reserved	(see note)[1]
3	PSE	Page Size Extension	The processor supports 4-Mbyte pages.
4	TSC	Time Stamp Counter	The RDTSC instruction is supported including the CR4.TSD bit for access/privilege control.
5	MSR	Model Specific Registers	Model Specific Registers are implemented with the RDMSR, WRMSR instructions.
6	PAE	Physical Address Extension	Physical addresses greater than 32 bits are supported.
7	MCE	Machine Check Exception	Machine Check Exception, Exception 18, and the CR4.MCE enable bit are supported.
8	CX8	CMPXCHG8 Instruction Supported	The compare and exchange 8 bytes instruction is supported.
9	APIC	Local APIC Supported	The processor contains a local APIC.
10-11		Reserved	(see note)[1]
12	MTRR	Memory Type Range Registers	The Processor supports the Memory Type Range Registers specifically the MTRR_CAP register.
13	PGE	Page Global Enable	The global bit in the PDE's and PTE's and the CR4.PGE enable bit are supported.
14	MCA	Machine Check Architecture	The Machine Check Architecture is supported, specifically the MCG_CAP register.
15	CMOV	Conditional Move Instruction Supported	The conditional move instructions are supported.
16–31		Reserved	(see note)[1]

1. Processor specific features are outside the scope of the CPUID Application Note and are not detailed in this publication. This information is available in the respective processor developer guides, or is available with the appropriate non-disclosure agreement in place. Contact Intel Corporation for details.

3.4 Cache Size and Format Information

When the EAX register contains a value of 2, the CPUID instruction loads the EAX, EBX, ECX and EDX registers with descriptors that indicate the processor's cache characteristics. The lower 8 bits of the EAX register (AL) contain a value that identifies the number of times the CPUID has to be executed to obtain a complete image of the processors' caching systems. For example, the Pentium Pro processor returns a value of 1 in the lower 8 bits of the EAX register.

The remainder of the EAX register, and the EBX, ECX, and EDX registers, contain valid 8 bit descriptors. Table 6 shows that valid 8 bit descriptors may be identified as such because their MSB is set to 0. To decode decriptors, move sequentially from the most significant byte of the register down through the least significant byte of the register. Table 7 lists the current descriptor values and their respective cache characteristics. This list will be extended in future as necessary.

Table 6. Descriptor Formats

Register MSB	Descriptor Type	Description
1	Reserved	Reserved for future use
0	8 bit descriptors	Descriptors point to a parameter table to identify cache characteristics. There is no data if the value is 0.

Table 7. Descriptor Decode Values

Descriptor Value	Cache Description
0x00	null
0x01	instruction TLB, 4K pages, 4 way set associative, 64 entries
0x02	instruction TLB, 4M pages, 4 way set associative, 4 entries
0x03	data TLB, 4K pages, 4 way set associative, 64 entries
0x04	data TLB, 4M pages, 4 way set associative, 8 entries
0x06	instruction cache, 8K, 4 way set associative, 32 byte line size
0x0A	data cache, 8K, 2 way set associative, 32 byte line size
0x41	unified cache, 32 byte cache line, 4 way set associative,128K
0x42	unified cache, 32 byte cache line, 4 way set associative, 256K
0x43	unified cache, 32 byte cache line, 4 way set associative, 512K

3.5 Example

The initial member of the Pentium Pro processor family returns the values in Table 8 .

Table 8. Pentium® Pro Processor CPUID (EAX=2) Return Values

Register	Value
EAX	0x03020101
EBX	0
ECX	0
EDX	0x06040A42

As the least significant byte (AL) of the EAX register indicates in Table 8, the CPUID instruction only needs to be executed once to obtain full details of the Pentium Pro processor cache structure. Table 8 also shows the MSB of the EAX register is 0, which indicates the upper 8 bits constitute an 8 bit descriptor. The remaining register values in Table 8 show that the Pentium Pro processor has the following cache characteristics:

- a data TLB that maps 4K pages, is 4 way set associative, and has 64 entries.
- an instruction TLB that maps 4M pages, is 4 way set associative, and has 4 entries.
- an instruction TLB that maps 4K pages, is 4 way set associative, and has 64 entries.
- an instruction cache that is 8K, is 4 way set associative, and has a 32 byte line size.
- a data TLB that maps 4M pages, is 4 way set associative, and has 8 entries.
- a data cache that is 8K, is 2 way set associative, and has a 32 byte line size.
- a unified cache that is 256K, is 4 way set associative, and has a 32 byte line size.

4.0 USAGE GUIDELINES

This document presents Intel-recommended feature-detection methods. Software should not try to identify features by exploiting programming tricks, undocumented features, or otherwise deviating from the guidelines presented in this Application Note.

The following guid elines are intended to help programmers maintain the widest range of compatibility for the ir software.

- Do not depend on the absence of an invalid opcode trap on the CPUID opcode to detect CPUID. Do not depend on the absence of an invalid opcode trap on the PUSHFD opcode to detect a 32-bit processor. Test the ID flag, as described in Section 2.0 and shown in Section 5.0 .
- Do not assu me that a given family or model has any specific feature. For example, do not assume the family value 5 (Pentium processor) means there is a floating-point unit on-chip. Use the feature flags for this determination.
- Do not assume processors with higher family or model numbers have all the features of a processor with a lower family or model number. For example, a processor with a family value of 6 (Pentium Pro processor) does not necessarily have all the features of a processor with a family value of 5.
- Do not assume that the features in the OverDrive processors are the same as those in the OEM version of the processor. Internal caches and instruction execution might vary.
- Do not use undocumented features of a processor to identify steppings or features. For example, the Intel386 processor A-step had bit instructions that were withdrawn with B-step. Some software attempted to execute these instructions and depended on the invalid-opcode exception as a signal that it

was not running on the A-step part. The software failed to work correctly when the Intel486 processor used the same opcodes for different instructions. The software should have used the stepping information in the processor signature.

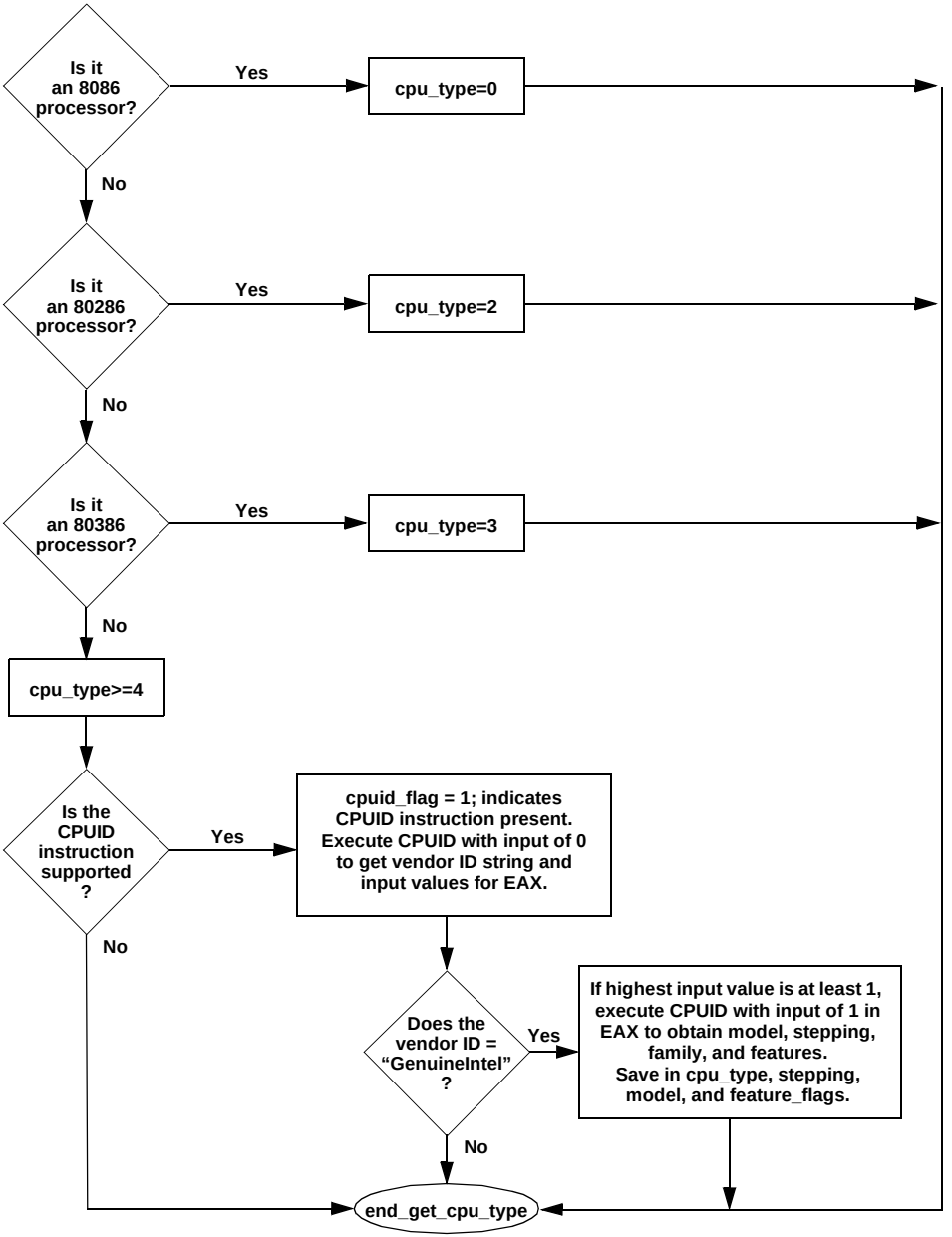
- Do not assume a value of 1 in a feature flag indicates that a given feature is present. For future feature flags, a value of 1 may indicate that the specific feature is not present.
- Test feature flags individually and do not make assumptions about undefined bits. For example, it would be a mistake to test the FPU bit by comparing the feature register to a binary 1 with a compare instruction.
- Do not assume the clock of a given family or model runs at a specific frequency, and do not write clock-dependent code, such as timing loops. For instance, an OverDrive Processor could operate at a higher internal frequency and still report the same family and/or model. Instead, use the system's timers to measure elapsed time. For processors that support the TSC (Time Stamp Counter) functionality, system timers can more directly calibrate the processor core block.
- Processor model-specific registers may differ among processors, including in various models of the Pentium processor. Do not use these registers unless identified for the installed processor. This is particularly important for systems upgradeable with an OverDrive processor. The model specific registers in the OverDrive processor are likely to differ from those in the OEM processor. In general, if an OverDrive processor is detected, model specific registers should not be used.

5.0 PROPER IDENTIFICATION SEQUENCE

The `cpuid3a.asm` program example demonstrates the correct use of the CPUID instruction. (See Example 1.) It also shows how to identify earlier processor generations that do not implement the processor signature or CPUID instruction. (See Figure 3.) This program example contains the following two procedures:

- **`get_cpu_type`** identifies the processor type. Figure 4 illustrates the flow of this procedure.
- **`get_fpu_type`** determines the type of floating-point unit (FPU) or math coprocessor (MCP).

This procedure has been tested with 8086, 80286, Intel386, Intel486, Pentium, and Pentium Pro processors. This program example is written in assembly language and is suitable for inclusion in a run-time library, or as system calls in operating systems.



. **Figure 3. Flow of Processor get_cpu_type Procedure**

6.0 USAGE PROGRAM EXAMPLE

The `cpuid3b.asm` and `cpuid3b.c` program examples demonstrate applications that call get_cpu_type and get_fpu_type procedures and interpret the returned information. The results, which are displayed on the monitor, identify the installed processor and features. The `cpuid3b.asm` example is written in assembly language and demonstrates an application that displays the returned information in the DOS environment. The `cpuid3b.c` example is written in the C language (See examples 2 and 3.) Figure 4 presents an overview of the relationship between the three program examples.

The `cpuidsta.cpp` and `cpuidsta.h` examples are 32-bit static libraries in C++, and includes Assembly Language code wherever necessary. They were built and tested in Microsoft Visual C++ versions 2.0 and 2.1.

The code supports:

- 80386 processors and above. (It does not work with processors before the 80386.)
- family identification
- feature identification
- cache parameter extraction

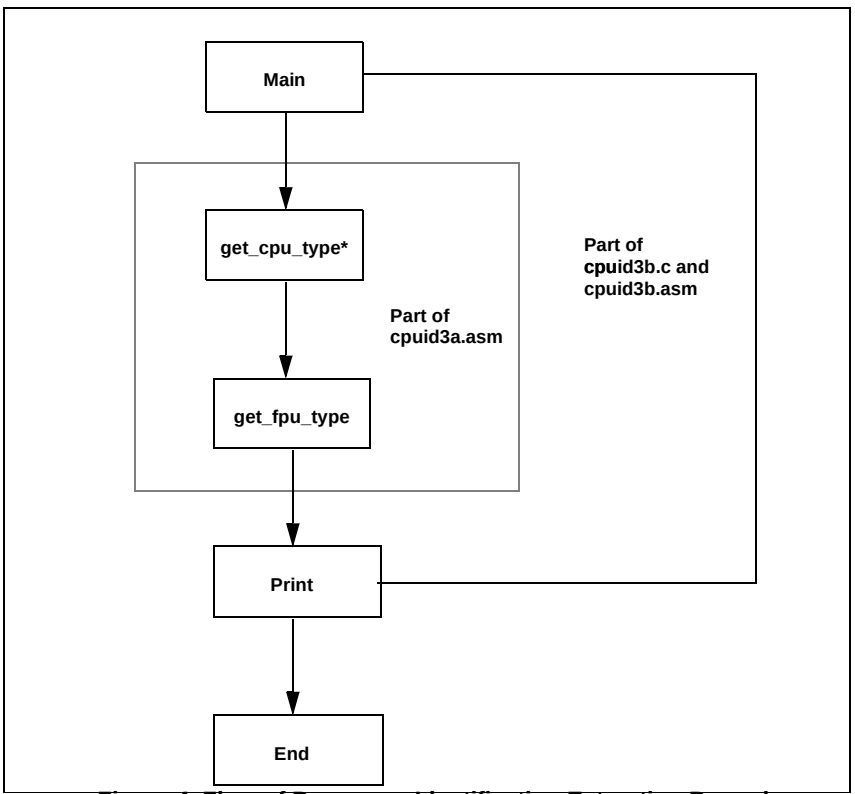
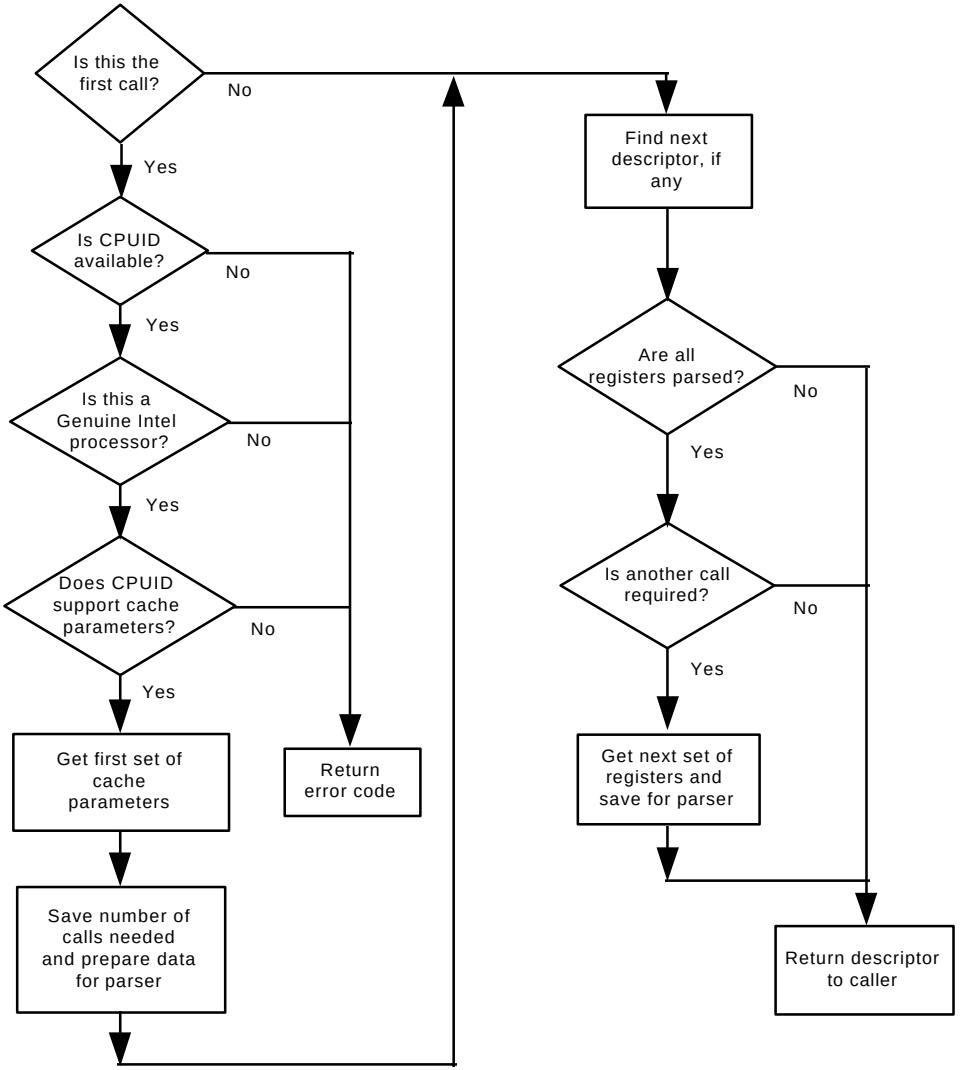


Figure 4. Flow of Processor Identification Extraction Procedures



. **Figure 5. Flow of Processor get_Processor_Cache_Description Procedure**

Example 1Descriptor check. Processor Identification Extraction Procedure

```
;       Filename:       cpuid3a.asm
;       Copyright 1993 - 95 by Intel Corp.
;
;       This program has been developed by Intel Corporation.  You
;       have Intel's permission to incorporate this source code into
;       your product, royalty free.  Intel has intellectual property
;       rights which it may assert if another manufacturer's processor
;       mis-identifies itself as being "GenuineIntel" when the CPUID
;       instruction is executed.
;
;       Intel specifically disclaims all warranties, express or
;       implied, and all liability, including consequential and other
;       indirect damages, for the use of this code, including
;       liability for infringement of any proprietary rights, and
;       including the warranties of merchantability and fitness for a
;       particular purpose.  Intel does not assume any responsibility
;       for any errors which may appear in this code nor any
;       responsibility to update it.
;
;       This code contains two procedures:
;       _get_cpu_type: Identifies processor type in _cpu_type:
;               0=8086/8088 processor
;               2=Intel 286 processor
;               3=Intel386(TM) family processor
;               4=Intel486(TM) family processor
;               5=Pentium(R) family processor
;
;       _get_fpu_type: Identifies FPU type in _fpu_type:
;               0=FPU not present
;               1=FPU present
;               2=287 present (only if _cpu_type=3)
;               3=387 present (only if _cpu_type=3)
;
;       This program has been tested with the MASM assembler.
;       This code correctly detects the current Intel 8086/8088,
;       80286, 80386, 80486, and Pentium(R) processors in the
;       real-address mode.
;
;       To assemble this code with TASM, add the JUMPS directive.
;       jumps                   ; Uncomment this line for TASM


        TITLE   cpuid2
        DOSSEG
        .model  small
```

```
CPU_ID  MACRO
        db      0fh             ; Hardcoded CPUID instruction
        db      0a2h
ENDM

        .data
        public  _cpu_type
        public  _fpu_type
        public  _cpuid_flag
        public  _intel_CPU
        public  _vendor_id
        public  _cpu_signature
        public  _features_ecx
        public  _features_edx
        public  _features_ebx
_cpu_type       db      0
_fpu_type       db      0
_cpuid_flag     db      0
_intel_CPU      db      0
_vendor_id      db      "------------"
intel_id        db      "GenuineIntel"
_cpu_signature  dd      0
_features_ecx   dd      0
_features_edx   dd      0
_features_ebx   dd      0
fp_status       dw      0

        .code
        .8086

;*************************************************************************

        public  _get_cpu_type
_get_cpu_type   proc

;       This procedure determines the type of processor in a system
;       and sets the _cpu_type variable with the appropriate
;       value.  If the CPUID instruction is available, it is used
;       to determine more specific details about the processor.
;       All registers are used by this procedure, none are preserved.
;       To avoid AC faults, the AM bit in CR0 must not be set.

;       Intel 8086 processor check
;       Bits 12-15 of the FLAGS register are always set on the
;       8086 processor.

check_8086:
        pushf                   ; push original FLAGS
        pop     ax              ; get original FLAGS
        mov     cx, ax          ; save original FLAGS
```

```
        and     ax, 0fffh       ; clear bits 12-15 in FLAGS
        push    ax              ; save new FLAGS value on stack
        popf                    ; replace current FLAGS value
        pushf                   ; get new FLAGS
        pop     ax              ; store new FLAGS in AX
        and     ax, 0f000h      ; if bits 12-15 are set, then
        cmp     ax, 0f000h      ;   processor is an 8086/8088
        mov     _cpu_type, 0    ; turn on 8086/8088 flag
        je      end_cpu_type    ; jump if processor is 8086/8088

;       Intel 286 processor check
;       Bits 12-15 of the FLAGS register are always clear on the
;       Intel 286 processor in real-address mode.

        .286
check_80286:
        or      cx, 0f000h      ; try to set bits 12-15
        push    cx              ; save new FLAGS value on stack
        popf                    ; replace current FLAGS value
        pushf                   ; get new FLAGS
        pop     ax              ; store new FLAGS in AX
        and     ax, 0f000h      ; if bits 12-15 are clear
        mov     _cpu_type, 2    ; processor=80286, turn on 80286 flag
        jz      end_cpu_type    ; if no bits set, processor is 80286

;       Intel386 processor check
;       The AC bit, bit #18, is a new bit introduced in the EFLAGS
;       register on the Intel486 processor to generate alignment
;       faults.
;       This bit cannot be set on the Intel386 processor.

        .386                    ; it is safe to use 386 instructions
check_80386:
        pushfd                  ; push original EFLAGS
        pop     eax             ; get original EFLAGS
        mov     ecx, eax        ; save original EFLAGS
        xor     eax, 40000h     ; flip AC bit in EFLAGS
        push    eax             ; save new EFLAGS value on stack
        popfd                   ; replace current EFLAGS value
        pushfd                  ; get new EFLAGS
        pop     eax             ; store new EFLAGS in EAX
        xor     eax, ecx        ; can't toggle AC bit, processor=80386
        mov     _cpu_type, 3    ; turn on 80386 processor flag
        jz      end_cpu_type    ; jump if 80386 processor

        push    ecx
        popfd                   ; restore AC bit in EFLAGS first

;       Intel486 processor check
;       Checking for ability to set/clear ID flag (Bit 21) in EFLAGS
```

```
;       which indicates the presence of a processor with the CPUID
;       instruction.

        .486
check_80486:
        mov     _cpu_type, 4    ; turn on 80486 processor flag
        mov     eax, ecx        ; get original EFLAGS
        xor     eax, 200000h    ; flip ID bit in EFLAGS
        push    eax             ; save new EFLAGS value on stack
        popfd                   ; replace current EFLAGS value
        pushfd                  ; get new EFLAGS
        pop     eax             ; store new EFLAGS in EAX
        xor     eax, ecx        ; can't toggle ID bit,
        je      end_cpu_type    ; processor=80486

;       Execute CPUID instruction to determine vendor, family,
;       model, stepping and features.  For the purpose of this
;       code, only the initial set of CPUID information is saved.

        mov     _cpuid_flag, 1  ; flag indicating use of CPUID inst.
        push    ebx             ; save registers
        push    esi
        push    edi
        mov     eax, 0          ; set up for CPUID instruction
        CPU_ID                  ; get and save vendor ID

        mov     dword ptr _vendor_id, ebx
        mov     dword ptr _vendor_id[+4], edx
        mov     dword ptr _vendor_id[+8], ecx

        mov     si, ds
        mov     es, si

        mov     si, offset _vendor_id
        mov     di, offset intel_id
        mov     cx, 12          ; should be length intel_id
        cld                     ; set direction flag
        repe    cmpsb           ; compare vendor ID to "GenuineIntel"
        jne     end_cpuid_type  ; if not equal, not an Intel processor

        mov     _intel_CPU, 1   ; indicate an Intel processor
        cmp     eax, 1          ; make sure 1 is valid input for CPUID
        jl      end_cpuid_type  ; if not, jump to end
        mov     eax, 1
        CPU_ID                  ; get family/model/stepping/features
        mov     _cpu_signature, eax
        mov     _features_ebx, ebx
        mov     _features_edx, edx
        mov     _features_ecx, ecx
```

```
        shr     eax, 8          ; isolate family
        and     eax, 0fh
        mov     _cpu_type, al   ; set _cpu_type with family

end_cpuid_type:
        pop     edi             ; restore registers
        pop     esi
        pop     ebx

        .8086
end_cpu_type:
        ret
_get_cpu_type   endp

;*************************************************************************

        public  _get_fpu_type
_get_fpu_type   proc

;       This procedure determines the type of FPU in a system
;       and sets the _fpu_type variable with the appropriate value.
;       All registers are used by this procedure, none are preserved.

;       Coprocessor check
;       The algorithm is to determine whether the floating-point
;       status and control words are present.  If not, no
;       coprocessor exists.  If the status and control words can
;       be saved, the correct coprocessor is then determined
;       depending on the processor type.  The Intel386 processor can
;       work with either an Intel287 NDP or an Intel387 NDP.
;       The infinity of the coprocessor must be checked to determine
;       the correct coprocessor type.

        fninit                  ; reset FP status word
        mov     fp_status, 5a5ah; initialize temp word to non-zero
        fnstsw  fp_status       ; save FP status word
        mov     ax, fp_status   ; check FP status word
        cmp     al, 0           ; was correct status written
        mov     _fpu_type, 0    ; no FPU present
        jne     end_fpu_type

check_control_word:
        fnstcw  fp_status       ; save FP control word
        mov     ax, fp_status   ; check FP control word
        and     ax, 103fh       ; selected parts to examine
        cmp     ax, 3fh         ; was control word correct
        mov     _fpu_type, 0
        jne     end_fpu_type    ; incorrect control word, no FPU
        mov     _fpu_type, 1
```

```
;       80287/80387 check for the Intel386 processor

check_infinity:
        cmp     _cpu_type, 3
        jne     end_fpu_type
        fld1                    ; must use default control from FNINIT
        fldz                    ; form infinity
        fdiv                    ; 8087/Intel287 NDP say +inf = -inf
        fld     st              ; form negative infinity
        fchs                    ; Intel387 NDP says +inf <> -inf
        fcompp                  ; see if they are the same
        fstsw   fp_status       ; look at status from FCOMPP
        mov     ax, fp_status
        mov     _fpu_type, 2    ; store Intel287 NDP for FPU type
        sahf                    ; see if infinities matched
        jz      end_fpu_type    ; jump if 8087 or Intel287 is present
        mov     _fpu_type, 3    ; store Intel387 NDP for FPU type
end_fpu_type:
        ret
_get_fpu_type   endp

        end
```

Example 2. Processor Identification Procedure in Assembly Language

```
;       Filename:       cpuid3b.asm
;       Copyright 1993 - 95 by Intel Corp.
;
;       This program has been developed by Intel Corporation.  You
;       have Intel's permission to incorporate this source code into
;       your product, royalty free.  Intel has intellectual property
;       rights which it may assert if another manufacturer's processor
;       mis-identifies itself as being "GenuineIntel" when the CPUID
;       instruction is executed.
;
;       Intel specifically disclaims all warranties, express or
;       implied, and all liability, including consequential and other
;       indirect damages, for the use of this code, including
;       liability for infringement of any proprietary rights, and
;       including the warranties of merchantability and fitness for a
;       particular purpose.  Intel does not assume any responsibility
;       for any errors which may appear in this code nor any
;       responsibility to update it.
;
;       This program contains three parts:
;       Part 1: Identifies processor type in the variable _cpu_type:
;
;       Part 2: Identifies FPU type in the variable _fpu_type:
;
;       Part 3: Prints out the appropriate message.  This part is
;               specific to the DOS environment and uses the DOS
;               system calls to print out the messages.
;
;       This program has been tested with the MASM assembler.
;       If this code is assembled with no options specified and linked
;       with the cpuid2 module, it correctly identifies the
;       current Intel 8086/8088, 80286, 80386, 80486, and Pentium(R)
;       processors in the real-address mode.
;
;       To assemble this code with TASM, add the JUMPS directive.
;       jumps                   ; Uncomment this line for TASM


        TITLE   cpuid20
        DOSSEG
        .model  small
        .stack  100h

        .data
        extrn   _cpu_type: byte
        extrn   _fpu_type: byte
        extrn   _cpuid_flag: byte
```

```
        extrn   _intel_CPU: byte
        extrn   _vendor_id: byte
        extrn   _cpu_signature: dword
        extrn   _features_ecx: dword
        extrn   _features_edx: dword
        extrn   _features_ebx: dword

;       The purpose of this code is to identify the processor and
;       coprocessor that is currently in the system.  The program
;       first determines the processor type.  Then it determines
;       whether a coprocessor exists in the system.  If a
;       coprocessor or integrated coprocessor exists, the program
;       identifies the coprocessor type.  The program then prints
;       the processor and floating point processors present and type.

        .code
        .8086
start:  mov     ax, @data
        mov     ds, ax          ; set segment register
        mov     es, ax          ; set segment register
        and     sp, not 3       ; align stack to avoid AC fault
        call    _get_cpu_type   ; determine processor type
        call    _get_fpu_type
        call    print
        mov     ax, 4c00h       ; terminate program
        int     21h

;*****************************************************************************

        extrn   _get_cpu_type: proc

;*****************************************************************************

        extrn   _get_fpu_type: proc

;*****************************************************************************


FPU_FLAG        equ     0001h
VME_FLAG        equ     0002h
PSE_FLAG        equ     0008h
MCE_FLAG        equ     0080h
CMPXCHG8B_FLAG  equ     0100h
APIC_FLAG       equ     0200h

        .data
id_msg          db      "This system has a$"
cp_error        db      "n unknown processor$"
cp_8086         db      "n 8086/8088 processor$"
cp_286          db      "n 80286 processor$"
```

```
cp_386          db      "n 80386 processor$"

cp_486          db      "n 80486DX, 80486DX2 processor or"
                db      " 80487SX math coprocessor$"
cp_486sx        db      "n 80486SX processor$"

fp_8087         db      " and an 8087 math coprocessor$"
fp_287          db      " and an 80287 math coprocessor$"
fp_387          db      " and an 80387 math coprocessor$"

intel486_msg    db      " Genuine Intel486(TM) processor$"
intel486dx_msg  db      " Genuine Intel486(TM) DX processor$"
intel486sx_msg  db      " Genuine Intel486(TM) SX processor$"
inteldx2_msg    db      " Genuine IntelDX2(TM) processor$"
intelsx2_msg    db      " Genuine IntelSX2(TM) processor$"
inteldx4_msg    db      " Genuine IntelDX4(TM) processor$"
inteldx2wb_msg  db      " Genuine Write-Back Enhanced"
                db      " IntelDX2(TM) processor$"
pentium_msg     db      " Genuine Intel Pentium(R) processor$"
unknown_msg     db      "n unknown Genuine Intel processor$"

; The following 16 entries must stay intact as an array
intel_486_0     dw      offset intel486dx_msg
intel_486_1     dw      offset intel486dx_msg
intel_486_2     dw      offset intel486sx_msg
intel_486_3     dw      offset inteldx2_msg
intel_486_4     dw      offset intel486_msg
intel_486_5     dw      offset intelsx2_msg
intel_486_6     dw      offset intel486_msg
intel_486_7     dw      offset inteldx2wb_msg
intel_486_8     dw      offset inteldx4_msg
intel_486_9     dw      offset intel486_msg
intel_486_a     dw      offset intel486_msg
intel_486_b     dw      offset intel486_msg
intel_486_c     dw      offset intel486_msg
intel_486_d     dw      offset intel486_msg
intel_486_e     dw      offset intel486_msg
intel_486_f     dw      offset intel486_msg
; end of array

family_msg      db      13,10,"Processor Family:  $"
model_msg       db      13,10,"Model:             $"
stepping_msg    db      13,10,"Stepping:          "
cr_lf           db      13,10,"$"

turbo_msg       db      13,10,"The processor is an OverDrive(TM)"
                db      " processor$"
dp_msg          db      13,10,"The processor is the upgrade processor"
                db      " in a dual processor system$"
fpu_msg         db      13,10,"The processor contains an on-chip FPU$"
```

```
mce_msg         db      13,10,"The processor supports Machine Check"
                db      " Exceptions$"
cmp_msg         db      13,10,"The processor supports the CMPXCHG8B"
                db      " instruction$"
vme_msg         db      13,10,"The processor supports Virtual Mode"
                db      " Extensions$"
pse_msg         db      13,10,"The processor supports Page Size"
                db      " Extensions$"
apic_msg        db      13,10,"The processor contains an on-chip"
                db      " APIC$"

not_intel       db      "t least an 80486 processor."
                db      13,10,"It does not contain a Genuine Intel"
                db      " part and as a result, the",13,10,"CPUID"
                db      " detection information cannot be determined"
                db      " at this time.$"

ASC_MSG MACRO   msg
        LOCAL   ascii_done              ; local label
        add     al, 30h
        cmp     al, 39h                 ; is it 0-9?
        jle     ascii_done
        add     al, 07h
ascii_done:
        mov     byte ptr msg[20], al
        mov     dx, offset msg
        mov     ah, 9h
        int     21h
ENDM

        .code
        .8086
print   proc

;       This procedure prints the appropriate cpuid string and
;       numeric processor presence status.  If the CPUID instruction
;       was used, this procedure prints out the CPUID info.
;       All registers are used by this procedure, none are preserved.

        mov     dx, offset id_msg       ; print initial message
        mov     ah, 9h
        int     21h

        cmp     _cpuid_flag, 1          ; if set to 1, processor
                                        ;   supports CPUID instruction
        je      print_cpuid_data        ; print detailed CPUID info

print_86:
        cmp     _cpu_type, 0
        jne     print_286
```

```
        mov     dx, offset cp_8086
        mov     ah, 9h
        int     21h
        cmp     _fpu_type, 0
        je      end_print
        mov     dx, offset fp_8087
        mov     ah, 9h
        int     21h
        jmp     end_print

print_286:
        cmp     _cpu_type, 2
        jne     print_386
        mov     dx, offset cp_286
        mov     ah, 9h
        int     21h
        cmp     _fpu_type, 0
        je      end_print
print_287:
        mov     dx, offset fp_287
        mov     ah, 9h
        int     21h
        jmp     end_print

print_386:
        cmp     _cpu_type, 3
        jne     print_486
        mov     dx, offset cp_386
        mov     ah, 9h
        int     21h
        cmp     _fpu_type, 0
        je      end_print
        cmp     _fpu_type, 2
        je      print_287
        mov     dx, offset fp_387
        mov     ah, 9h
        int     21h
        jmp     end_print

print_486:
        cmp     _cpu_type, 4
        jne     print_unknown           ; Intel processors will have
        mov     dx, offset cp_486sx     ; CPUID instruction
        cmp     _fpu_type, 0
        je      print_486sx
        mov     dx, offset cp_486
print_486sx:
        mov     ah, 9h
        int     21h
        jmp     end_print
```

```
print_unknown:
      mov     dx, offset cp_error
      jmp     print_486sx

print_cpuid_data:
      .486
      cmp     _intel_CPU, 1          ; check for genuine Intel
      jne     not_GenuineIntel       ;   processor
print_486_type:
      cmp     _cpu_type, 4           ; if 4, print 80486 processor
      jne     print_pentium_type
      mov     ax, word ptr _cpu_signature
      shr     ax, 4
      and     eax, 0fh               ; isolate model
      mov     dx, intel_486_0[eax*2]
      jmp     print_common
print_pentium_type:
      cmp     _cpu_type, 5           ; if 5, print Pentium processor
      jne     print_unknown_type
      mov     dx, offset pentium_msg
      jmp     print_common
print_unknown_type:
      mov     dx, offset unknown_msg  ; if neither, print unknown

print_common:
      mov     ah, 9h
      int     21h

; print family, model, and stepping

print_family:
      mov     al, _cpu_type
      ASC_MSG family_msg             ; print family msg

print_model:
      mov     ax, word ptr _cpu_signature
      shr     ax, 4
      and     al, 0fh
      ASC_MSG model_msg              ; print model msg

print_stepping:
      mov     ax, word ptr _cpu_signature
      and     al, 0fh
      ASC_MSG stepping_msg           ; print stepping msg

print_upgrade:
      mov     ax, word ptr _cpu_signature
      test    ax, 1000h              ; check for turbo upgrade
      jz      check_dp
```

```
        mov     dx, offset turbo_msg
        mov     ah, 9h
        int     21h
        jmp     print_features

check_dp:
        test    ax, 2000h               ; check for dual processor
        jz      print_features
        mov     dx, offset dp_msg
        mov     ah, 9h
        int     21h

print_features:
        mov     ax, word ptr _features_edx
        and     ax, FPU_FLAG            ; check for FPU
        jz      check_MCE
        mov     dx, offset fpu_msg
        mov     ah, 9h
        int     21h

check_MCE:
        mov     ax, word ptr _features_edx
        and     ax, MCE_FLAG            ; check for MCE
        jz      check_CMPXCHG8B
        mov     dx, offset mce_msg
        mov     ah, 9h
        int     21h

check_CMPXCHG8B:
        mov     ax, word ptr _features_edx
        and     ax, CMPXCHG8B_FLAG      ; check for CMPXCHG8B
        jz      check_VME
        mov     dx, offset cmp_msg
        mov     ah, 9h
        int     21h

check_VME:
        mov     ax, word ptr _features_edx
        and     ax, VME_FLAG            ; check for VME
        jz      check_PSE
        mov     dx, offset vme_msg
        mov     ah, 9h
        int     21h

check_PSE:
        mov     ax, word ptr _features_edx
        and     ax, PSE_FLAG            ; check for PSE
        jz      check_APIC
        mov     dx, offset pse_msg
        mov     ah, 9h
```

```
        int     21h

check_APIC:
        mov     ax, word ptr _features_edx
        and     ax, APIC_FLAG           ; check for APIC
        jz      end_print
        mov     dx, offset apic_msg
        mov     ah, 9h
        int     21h

        jmp     end_print

not_GenuineIntel:
        mov     dx, offset not_intel
        mov     ah, 9h
        int     21h

end_print:
        mov     dx, offset cr_lf
        mov     ah, 9h
        int     21h
        ret
print   endp

        end     start
```

Example 3. Processor Identification Procedure in the C Language

```
/* Filename:    cpuid3b.c                                   */
/* Copyright 1994 by Intel Corp.                            */
/*                                                          */
/* This program has been developed by Intel Corporation.  You    */
/* have Intel's permission to incorporate this source code into  */
/* your product, royalty free. Intel has intellectual property   */
/* rights which it may assert if another manufacturer's processor*/
/* mis-identifies itself as being "GenuineIntel" when the CPUID  */
/* instruction is executed.                                 */
/*                                                          */
/* Intel specifically disclaims all warranties, express or       */
/* implied, and all liability, including consequential and other */
/* indirect damages, for the use of this code, including         */
/* liability for infringement of any proprietary rights, and     */
/* including the warranties of merchantability and fitness for a */
/* particular purpose.  Intel does not assume any responsibility */
/* for any errors which may appear in this code nor any          */
/* responsibility to update it.                             */
/*                                                          */
/* This program contains three parts:                       */
/* Part 1: Identifies CPU type in the variable _cpu_type:        */
/*                                                          */
/* Part 2: Identifies FPU type in the variable _fpu_type:        */
/*                                                          */
/* Part 3: Prints out the appropriate message.              */
/*                                                          */
/* This program has been tested with the Microsoft C compiler.   */
/* If this code is compiled with no options specified and linked */
/* with the cpuid2 module, it correctly identifies the           */
/* current Intel 8086/8088, 80286, 80386, 80486, and             */
/* Pentium(R) processors in the real-address mode.               */

#define FPU_FLAG        0x0001
#define VME_FLAG        0x0002
#define PSE_FLAG        0x0008
#define MCE_FLAG        0x0080
#define CMPXCHG8B_FLAG  0x0100
#define APIC_FLAG       0x0200

extern char cpu_type;
extern char fpu_type;
extern char cpuid_flag;
extern char intel_CPU;
extern char vendor_id[12];
extern long cpu_signature;
extern long features_ecx;
extern long features_edx;
extern long features_ebx;
```

```
main() {
   get_cpu_type();
   get_fpu_type();
   print();
}
print() {
   printf("This system has a");
   if (cpuid_flag == 0) {
      switch (cpu_type) {
      case 0:
         printf("n 8086/8088 processor");
         if (fpu_type) printf(" and an 8087 math coprocessor");
         break;
      case 2:
         printf("n 80286 processor");
         if (fpu_type) printf(" and an 80287 math coprocessor");
         break;
      case 3:
         printf("n 80386 processor");
         if (fpu_type == 2)
            printf(" and an 80287 math coprocessor");
         else if (fpu_type)
            printf(" and an 80387 math coprocessor");
         break;
      case 4:
         if (fpu_type) printf("n 80486DX, 80486DX2 processor or \
80487SX math coprocessor");
         else printf("n 80486SX processor");
         break;
      default:
         printf("n unknown processor");
      }
   } else {
      /* using cpuid instruction */
      if (intel_CPU) {
         if (cpu_type == 4) {
            switch ((cpu_signature>>4)&0xf) {
            case  0:
            case  1:
               printf(" Genuine Intel486(TM) DX processor");
               break;
            case  2:
               printf(" Genuine Intel486(TM) SX processor");
               break;
            case  3:
               printf(" Genuine IntelDX2(TM) processor");
               break;
            case  4:
               printf(" Genuine Intel486(TM) processor");
               break;
```

```
            case  5:
               printf(" Genuine IntelSX2(TM) processor");
               break;
            case  7:
               printf(" Genuine Write-Back Enhanced \
IntelDX2(TM) processor");
               break;
            case  8:
               printf(" Genuine IntelDX4(TM) processor");
               break;
            default:
               printf(" Genuine Intel486(TM) processor");
            }
         } else if (cpu_type == 5)
            printf(" Genuine Intel Pentium(R) processor");
         else
            printf("n unknown Genuine Intel processor");
         printf("\nProcessor Family: %X", cpu_type);
         printf("\nModel:           %X", (cpu_signature>>4)&0xf);
         printf("\nStepping:        %X\n", cpu_signature&0xf);
         if (cpu_signature & 0x1000)
            printf("\nThe processor is an OverDrive(TM) upgrade
                  \processor");
         else if (cpu_signature & 0x2000)
            printf("\nThe processor is the upgrade processor \
in a dual processor system");
         if (features_edx & FPU_FLAG)
            printf("\nThe processor contains an on-chip FPU");
         if (features_edx & MCE_FLAG)
            printf("\nThe processor supports Machine Check \
Exceptions");
         if (features_edx & CMPXCHG8B_FLAG)
            printf("\nThe processor supports the CMPXCHG8B \
instruction");
         if (features_edx & VME_FLAG)
            printf("\nThe processor supports Virtual Mode \
Extensions");
         if (features_edx & PSE_FLAG)
            printf("\nThe processor supports Page Size \
Extensions");
         if (features_edx & APIC_FLAG)
            printf("\nThe processor contains an on-chip APIC");
       } else {
          printf("t least an 80486 processor.\nIt does not \
contain a Genuine Intel part and as a result, the\nCPUID detection \
information cannot be determined at this time.");
       }
    }
    printf("\n");
}
```

Example 4 Processor Identification C++ Code

```
//////////////////////////////////////////////////////////
//
// Filename:      cpuidsta.h
// Copyright 1995 by Intel Corp.
//
// This program has been developed by Intel Corporation.
// You have Intel's permission to incorporate this code
// into your product, royalty free.  Intel has various
// intellectual property rights which it may assert under
// certain circumstances, such as if another manufacturer's
// processor mis-identifies itself as being "GenuineIntel"
// when the CPUID instruction is executed.
//
// Intel specifically disclaims all warranties, express or
// implied, and all liability, including consequential and
// other indirect damages, for the use of this code,
// including liability for infringement of any proprietary
// rights, and including the warranties of merchantability
// and fitness for a particular purpose.  Intel does not
// assume any responsibility for any errors which may appear
// in this code nor any responsibility to update it.
//
//////////////////////////////////////////////////////////


//
// Header file for the CPUID utility 32-bit static C library
//

#ifndef CPUIDSTA_H
#define CPUIDSTA_H

#include <afxwin.h> // for definition of BOOL

//
// All 32 bit integers are defined on this typedef for
// portability.
//
typedef long int TInt32;

//
// All 16 bit integers are defined on this typedef.
//
typedef short int TInt16;


//
// The following family of typedefs are used to
// interpret the results of the CPUID instruction.
//
typedef struct TCpuidInfoAsRegs {
```

```
                        TInt32      eax;
                        TInt32      ebx;
                        TInt32      edx;
                        TInt32      ecx;
                } TCpuidInfoAsRegs;

typedef struct TCpuidInfoVendorId {
                        TInt32      highestRecognized;
                        char        vendor[12];
                } TCpuidInfoVendorId;

typedef struct TCpuidInfoSignature {
                        TInt32      signature;
                        TInt32      intelReserved1;
                        TInt32      featureFlags;
                        TInt32      intelReserved2;
                } TCpuidInfoSignature;

typedef union {
                TCpuidInfoAsRegs reg;
                        TCpuidInfoVendorId id;
                        TCpuidInfoSignature processor;
                } TCpuidInfo;


//
// These are symbolic definitions for processor family codes.
//
#define GENUINE_INTEL_NO_CPUID  (-2)
#define NON_INTEL_WITH_CPUID  (-1)
#define FAMILY_386                                              ( 3)
#define FAMILY_486                                              ( 4)
#define GENUINE_INTEL_PENTIUM  ( 5)
#define GENUINE_INTEL_PENTIUM_PRO  ( 6)


//
// These masks are used to test processor flag bits
// for cpu identification.
//
#define FLAGS_AC_BIT (0x00040000)
#define FLAGS_ID_BIT (0x00200000)

//
// This define generates the op code for the CPUID
// instruction in inline assembly code.
//
#define CPUID _asm _emit 0x0F _asm _emit 0xA2

//
```

```
// These masks are used to test the feature bits returned
// by the Intel implementation of the CPUID instruction.
//
#define FEATURE_FPU     (0x00000001)
#define FEATURE_VME     (0x00000002)
#define FEATURE_PSE     (0x00000008)
#define FEATURE_TSC     (0x00000010)
#define FEATURE_MSR     (0x00000020)
#define FEATURE_PAE     (0x00000040)
#define FEATURE_MCE     (0x00000080)
#define FEATURE_CX8     (0x00000100)
#define FEATURE_APIC (0x00000200)
#define FEATURE_MTRR (0x00001000)
#define FEATURE_PGE     (0x00002000)
#define FEATURE_MCA     (0x00004000)
#define FEATURE_CMOV (0x00008000)


//
// Numeric Data Processing unit identification constants.
//
#define FPU_NONE                                (0)
#define FPU_287                                         (2)
#define FPU_387                                         (3)
#define FPU_487SX                               (4)
#define FPU_486DX_DX2_487SX (5)
#define FPU_ON_CHIP                             (6)

//////////////////////////////////////////////////////////
//
// Cache descriptors
//
// Defined descriptors: (All others reserved.)
//

// 1. The null descriptor:
#define CD_NULL                                 (0)

// 2. Instruction TLB, 4K pages, 4-way set associative,
//    64 entries.
#define CD_ITLB_P4K_W4_64 (0x01)

// 3. Instruction TLB, 4M pages, 4-way set associative,
//    4 entries.
#define CD_ITLB_P4M_W4_4 (0x02)

// 4. Data TLB, 4K pages, 4-way set associative,
//    64 entries.
#define CD_DTLB_P4K_W4_64 (0x03)
```

```
// 5. Data TLB, 4M pages, 4-way set associative,
//    8 entries.
#define CD_DTLB_P4K_W4_8 (0x04)

// 6. Instruction cache, 32 byte lines, 4-way set
//    associative, 8K
#define CD_IC_L32_W4_8K (0x06)

// 7. Data cache, 32 byte lines, 2-way set
//    associative, 8K
#define CD_DC_L32_W2_8K (0x0A)

// 8. Unified cache, 32 byte lines, 4-way set
//    associative, 128K
#define CD_UC_L32_W4_128K (0x41)

// 9. Unified cache, 32 byte lines, 4-way set
//    associative, 256K
#define CD_UC_L32_W4_256K (0x42)

// 10.Unified cache, 32 byte lines, 4-way set
//    associative, 512K
#define CD_UC_L32_W4_512K (0x43)


//////////////////////////////////////////////////////////
//
// Function prototypes.
//

//
// cpuidInstructionAvailable - determines if CPUID
//                     instruction is available for use.
//
BOOL
cpuidInstructionAvailable();

//
// processorFamily - Returns the family identification
//   code, using the CPUID instruction where possible.
//
TInt32
processorFamily();

//
// rawCPUID - provides a raw, unguarded interface
//                     to the CPUID instruction.
//
TCpuidInfo *
rawCPUID (TInt32 query, TCpuidInfo *info);
```

```
//
// numericCoprocessorType - identifies floating point unit
//
TInt32
numericCoprocessorType();

//
// genuineIntel   - Tests vendor identification, if possible.
//
BOOL
genuineIntel();

//
// processorSignature - Get raw signature word.
//
TInt32
processorSignature();

//
// processorFeatureFlags - Get raw feature word.
//
TInt32
processorFeatureFlags();

//
// processorHasIntel - tests for the presence of
// individual feature bits as specified by a mask.
//
BOOL
processorHasIntel(TInt32 featureMask);

//
// fpuType - Determine type of floating point unit.
//
TInt32
fpuType();

//
// getProcessorCacheDescription - returns cache information
//   from the CPUID instruction.
//
TInt32
getProcessorCacheDescription(BOOL firstCall);

#endif
```

Example 5 Processor Identification C++ Code

```
//////////////////////////////////////////////////////////
//
// Filename:      cpuidsta.cpp
// Copyright 1995 by Intel Corp.
//
// This program has been developed by Intel Corporation.
// You have Intel's permission to incorporate this code
// into your product, royalty free.  Intel has various
// intellectual property rights which it may assert under
// certain circumstances, such as if another manufacturer's
// processor mis-identifies itself as being "GenuineIntel"
// when the CPUID instruction is executed.
//
// Intel specifically disclaims all warranties, express or
// implied, and all liability, including consequential and
// other indirect damages, for the use of this code,
// including liability for infringement of any proprietary
// rights, and including the warranties of merchantability
// and fitness for a particular purpose.  Intel does not
// assume any responsibility for any errors which may appear
// in this code nor any responsibility to update it.
//
//////////////////////////////////////////////////////////

//
// CPUID utility 32-bit static C library
//

//
// This library assumes that it is running on a 386 or
// later.  Some functions will cause an illegal instruction
// exception if it is not.
//

#include <string.h>
#include "cpuidsta.h"

//
// cpuidInstructionAvailable - determines if CPUID
//                      instruction is available for use.
// returns
//   FALSE (0) : CPUID instruction is not supported
//   TRUE  (1) : CPUID instruction is supported
//
extern
BOOL
cpuidInstructionAvailable()
{
TInt32              available;
```

```
        _asm {
                pushfd                          // Save flags
                pushad                          // .. and registers.
                pushfd                          // Transfer flags
                pop             eax             // ..to the general
    registers.
                mov             ecx,eax // Save a copy of the flags.
                xor             eax,FLAGS_ID_BIT // Toggle the ID bit.
                push    eax                     // Transfer flags
                popfd                           // ..back to flags register
                pushfd                          // ..and now back again
                pop             eax             // ..to general registers.
                // If the ID bit did not stay put, the CPUID
                // ..instruction is not supported.
                xor             eax,ecx // Test against original
                jz              nocpuid // ..and jump if still the same.
                mov             available,TRUE // Indicate CPUID available
                jmp             done    // ..and leave
        nocpuid:
                mov             available,FALSE // ..else indicate CPUID
                                                                        // ..not
    available.
        done:
                popad                           // Restore original flags
                popfd                           // ..and registers before exit.
        }
        return available;
}

//
// processorFamily - Returns the family identification
//   code, using the CPUID instruction where possible.
// return values:
//  -2 : supports CPUID instruction and is Genuine Intel,
//                                  but CPUID implementation does not return family.
//  -1 : supports CPUID instruction but is not Genuine Intel
//   3 : 386
//   4 : 486
//   5 : Pentium(R) Processor
//   6 : Pentium(R) Pro Processor
//  >6 : Genuine Intel, supports CPUID instruction.  Family
//                                  code is returned directly from CPUID.
//
// Note:
//  1. This function does not test for 286 and earlier
//     processor and will cause illegal instruction
//     exceptions on those processors.
//  2. Some, but not all, 486 processor support the CPUID
//     instruction.  This functions uses CPUID where
//     supported.
```

```
//  3. No family code is reported for processors that
//     support CPUID but are not Genuine Intel because
//     the processor signature format is specific to
//     the Intel implementation of CPUID.
//
extern
TInt32
processorFamily()
{
        TInt32      family;
        TInt32      highParameter;
        // The text "GenuineIntel" in the form returned by
        // a Genuine Intel processor with the CPUID instruction.
        TInt32      id0 = 'uneG';
        TInt32      id1 = 'Ieni';
        TInt32      id2 = 'letn';

        _asm {
                pushfd                          // Save flags
                pushad                          // .. and registers.
                pushfd                          // Transfer flags
                pop             eax             // ..to the general
    registers.
                mov             ecx,eax // Save a copy of the flags.

                // 386 Test
                xor             eax,FLAGS_AC_BIT // Toggle the AC bit.
                push    eax                     // Transfer flags
                popfd                           // ..back to flags register
                pushfd                          // ..and now back again
                pop             eax             // ..to general registers.
                // If the AC bit did not stay put, then
                // is 386.
                xor             eax,ecx         // Compare to saved flags.
                mov             family,3        // Set family code
                jz              done            // Return to caller if a
    386.

                // If here, the processor is at least a 486.

                // 486 Test
                mov             eax,ecx         // Get fresh copy of
    flags.
                xor             eax,FLAGS_ID_BIT // Toggle the ID bit.
                push    eax                     // Transfer flags
                popfd                           // ..back to flags register
                pushfd                          // ..and now back again
                pop             eax             // ..to general registers.
                // If the ID bit did not stay put, the CPUID
                // ..instruction is not supported. If CPUID is
```

```
                        // ..not supported, it must be a 486 if we are
                        // ..this far into the routine.
                        xor             eax,ecx     // Test against original flags
                        jnz             hascpuid    // ..and jump if is different.
                        mov             family,4    // Set family code
                        jmp             done                // ..and return to caller.

                hascpuid:
                        xor             eax,eax             // Zero out EAX.
                        CPUID                               // Execute cpuid
    instruction.
                        mov             highParameter,eax // Save highest
                                                                        //
    ..allowable parameter.
                        // Test the vendor identification string.
                        cmp             ebx,id0             // Test 'Genu'.
                        jne             notIntel    // Branch if no match.
                        cmp             edx,id1             // Test 'ineI'
                        jne             notIntel    // Branch if no match.
                        cmp             ecx,id2             // Test 'ntel'.
                        je              isIntel             // Branch if match.
                notIntel:
                        mov             family,-1   // Set family code to non-Intel
                        jmp             done                // ..and exit function
                isIntel:
                        // Make sure processor signature is supported.
                        mov             eax,1               // Parameter for
    signature.
                        cmp             eax,highParameter // Is it supported?
                        jle             hasSig              // Branch if supported.
                        mov             family,-2   // Else return code for Intel
                        jmp             done                // ..but no signature.
                hasSig:
                        // retrieve family code from processor signature.
                        CPUID
                        and             eax,0x0f00  // Extract family code.
                        shr             eax,8               // Align family code
                        mov             family,eax  // ..and set the return value.
                        // Fall into exit code.

                done:
                        popad                       // Restore original flags
                        popfd                       // ..and registers before exit.
                }
                return family;
}


//
// rawCPUID - provides a raw, unguarded interface
```

```
//                      to the CPUID instruction.
// parameters:
//   query : input parameter to the CPUID instruction.
//   info : a POINTER to an information record provided
//                                              by the caller.
// result:
//   The record at *info is filled with the information
//                      returned by the CPUID instruction without modification.
// . A pointer to the record is returned as a result.
//
// Note:
//   This function will cause an illegal instruction
//                      trap on processors that do not support CPUID.
//
extern
TCpuidInfo *
rawCPUID (TInt32 query, TCpuidInfo *info)
{
                    _asm {
                                        pushfd                                                  // Save flags
                                        pushad                                                  // .. and
     registers.

                                        mov             eax,query               // Set CPUID parameter.
                                        CPUID                                                   // Call
     CPUID function.
                                        push    eax                                             // Save for
     the moment.
                                        mov             eax,[info]              // Get pointer to record.
                                        mov             [eax+4],ebx             // Move vendor id string
                                        mov             [eax+8],edx             // ..into the information
                                        mov             [eax+12],ecx // ..record.
                                        pop             ebx                                     // Get back
     highest value
                                        mov             [eax],ebx               // ..and move to the
     record.

                                        popad                                                   // Restore
     original flags
                                        popfd                                                   // ..and
     registers on exit.
                    }
                    return info;
}


//
// numericCoprocessorType - identifies floating point unit
// returns:
//   0 : no floating point unit
```

```
//   2 : 287 floating point unit
//   3 : 387 or higher floating point unit
//
// Note:
//   1. This function assumes that the cpu is at least
//   a 386 and that the numeric processing unit is at
//   least a 287.
//
//   2. A 386 can be used with either a 287 or a 387.
//
extern
TInt32
numericCoprocessorType()
{
        TInt32  fpuType;
        TInt16  fp_status = 0x5a5a; // Initialize non-zero.

        _asm {
                pushfd                                  // Save flags
                pushad                                  // .. and registers.

                fninit                                  // Reset the FP status
    word.
                fnstsw  fp_status  // Save the FP status word.
                mov     ax,fp_status  // Check the FP status word
                cmp     al,0                    // ..for correct status
                mov     fpuType,0  // No FPU present.
                jmp     done

                // check for 287/387
                fld1                            // Must use default control from
                                                        // ..fninit.
                fldz                            // Form an infinity by
                fdiv                            // ..dividing by zero.
                fld             st                      // Form a negative
    infinity from
                fchs                            // ..a copy of the positive inf.
                fcompp                          // See if the positive and the
                fstsw fp_status // ..negative infinity are the same.
                mov     fpuType,2  // Assume a 287 for now.
                mov     ax,fp_status  // Get the FP status
                sahf                            // ..into the flags.
                jz      done            // Jump if is 287.
                mov     fpuType,3  // 387 says +inf != -inf.

        done:
                popad                                   // Restore original flags
                popfd                                   // ..and registers on exit.
        }
        return fpuType;
```

```
}

//
// genuineIntel   - Tests vendor identification, if possible.
//
// returns:
//   FALSE : can not be determined, or
//                                   determined not to be Genuine Intel
//   TRUE  : Genuine Intel processor
//
extern
BOOL
genuineIntel()
{
                TCpuidInfo  info;

                if (cpuidInstructionAvailable())
                {
                                rawCPUID(0,&info);
                                if (strncmp(info.id.vendor, “GenuineIntel”,12) == 0)
                                {
                                                return TRUE;
                                }
                                else
                                {
                                                return FALSE;
                                }
                }
                else
                {
                                return FALSE;
                }
}

//
// processorSignature - Get raw signature word.
// returns:
//   Raw processor signature from CPUID  instruction, or
//    0 if processor signature is not supported, or
//   -1 if not a Genuine Intel processor.
//
// Note:
//   1. If the processor is not a Genuine Intel processor
//   as determined by the function genuineIntel() then
//   the value -1 is returned.  The signature format is
//   specific to the Intel implementation of the CPUID
//   instruction.
//
//   2. Strictly speaking, ALL possible 32-bit values
//   are reserved by Intel as potential processor
```

```
//   signatures.  This means that callers should not
//   rely on the return code of -1 as an error
//   indication.  It is more robust to guard calls to
//   this function with a test of genuineIntel() in the
//   calling routine.
//
//   3. This function requires the CPUID instruction.
//
extern
TInt32
processorSignature()
{
                TCpuidInfo  info;

                // Processor signature is only defined for
                // Genuine Intel processors, so guard with test.
                if (genuineIntel() == 1)
                {
                        rawCPUID(0,&info);
                        if (info.id.highestRecognized >= 1)
                        {
                                rawCPUID(1,&info);
                                return (info.processor.signature);
                        }
                        else
                        {
                                return 0;     // Return no signature.
                        }
                }
                return -1; // Return error code.
}

//
// processorFeatureFlags - Get raw feature word.
// returns:
//   The 32 bit feature flag value, or
//   0, if the CPUID instruction is not supported, or
//   0, if not a Genuine Intel processor, or
//   0, if Genuine Intel without feature flag support.
//
// Notes:
//   1. If the processor is not a Genuine Intel processor
//   as determined by the function genuineIntel() then
//   the value 0 is returned.  Feature flags are
//   specific to the Intel implementation of the CPUID
//   instruction.
//
//   2. This function requires the CPUID instruction.
//
extern
```

```
TInt32
processorFeatureFlags()
{
        TCpuidInfo info;

        if (genuineIntel() == 1)
        {
                rawCPUID(0,&info);
                if (info.id.highestRecognized >= 1)
                {
                        rawCPUID(1,&info);
                        return (info.processor.featureFlags);
                }
        }
        return 0;
}


//
// processorHasIntel - tests for the presence of
// individual feature bits as specified by a mask.
//
// parameters:
//   featureMask - feature bit(s) to test.
//
// result:
//   TRUE  - All features specified in mask are present.
//   FALSE - Feature(s) specified by mask not present
//           or CPUID instruction is not supported
//           or not a Genuine Intel Processor.
//
// Notes:
//   1. If the processor is not a Genuine Intel processor
//   as determined by the function genuineIntel() then
//   the value FALSE is returned.  Feature flags are
//   specific to the Intel implementation of the CPUID
//   instruction.
//
//   2. This function requires the CPUID instruction.
//
// Use:
//   Use this function with the FEATURE_xxx mask
//   definitions found in the header file for
//   easy to read feature test predicates.
//
// Usage example:
//   if (processorHasIntel(FEATURE_APIC)) {...}
//
extern
BOOL
```

```
processorHasIntel(TInt32 featureMask)
{
        TCpuidInfo  info;

        if (genuineIntel() == 1)
        {
                rawCPUID(0,&info);
                if (info.id.highestRecognized >= 1)
                {
                        rawCPUID(1,&info);
                        return ((info.processor.featureFlags
                                                                & featureMask) ==
    featureMask);
                }
                else
                {
                        return FALSE;
                }
        }
        else
        {
                return FALSE;
        }
}

//
// fpuType - Determine type of floating point unit.
//
// returns:
//  FPU_NONE  : if no floating point unit
//  FPU_287           : if 287 floating point unit
//  FPU_387           : if 387 floating point unit
//  FPU_487SX : only if unambiguously an Intel 487SX NDP
//  FPU_486DX_DX2_487SX : if one of 486DX, 486DX2, or
//                                                                  486SX with
    487SX
//  FPU_ON_CHIP : if Genuine Intel processor that supports
//                                                  CPUID and that reports FPU feature
    flag.
//  -1                  : if this routine cannot recognize the FPU
//
extern
TInt32
fpuType()
{
        TInt32      coProc;
        TInt32      family;

        family = processorFamily();
```

```
if (cpuidInstructionAvailable())
{
        if (processorHasIntel(FEATURE_FPU))
        {
                return FPU_ON_CHIP;
        }
        else
        {
                if (family == 4)
                {
                        // Test for 487SX
                        if (numericCoprocessorType() == 0)
                        {
                                return FPU_NONE;
                        }
                        else
                        {
                                return FPU_487SX;
                        }
                }
                else
                {
                        return FPU_NONE;
                }
        }
}
else
{
        // Family is either 3 or 4

        coProc = numericCoprocessorType();

        if (coProc == 0)
        {
                return FPU_NONE;
        }
        else if (coProc == 2)
        {
                // Must be a 386 system with a 287.
                return FPU_287;
        }
        else if (coProc == 3)
        {
                if (family == 4)
                {
                        // Is either a 486DX or 486DX2
                        // or a 486SX with a 487SX.
                        return FPU_486DX_DX2_487SX;
                }
                else
```

```
                                        {
                                                // Must be a 386 with a 387.
                                                return FPU_387;
                                        }
                                }
                        }
                        return -1; // Return an error code if we got here.
}

//
// getProcessorCacheDescription - returns cache information
//   from the CPUID instruction. Each call returns a single
//                    cache descriptor.  Call this function in a loop to
//   collect all available cache information.
//
// parameters:
//   firstCall: If TRUE, starts retrieval of cache
//     information from the beginning.  Set to FALSE
//     on success calls to retrieve the next cache
//     available descriptor.

// returns:
//  -1: This processor does not support cache parameter
//     information via the CPUID instruction.
//   CD_NULL: End of information marker.
//   otherwise: A cache descriptor.
//
// Code for the function getProcessorCacheDescription()
// follows static declarations and the support function
// parseCacheDescriptor().
//

// Static variables used by getProcessorCacheDescription()
static TCpuidInfo cpuidCacheInfo; // info from last call
static TInt32                   callsRemaining; // to CPUID
static TInt32                   bytesRemaining; // from last call
static TInt32                   registerBeingParsed;


// Support function parseCacheDescriptor() used by
// getProcessorCacheDescription()
TInt32
parseCacheDescriptor(void)
{
                        TInt32          descriptor;
                        int                             descriptorLength;

                        descriptor = CD_NULL;
                        while (bytesRemaining > 0 && descriptor == CD_NULL)
                        {
```

```
// Get the next register to parse, when needed.
switch (bytesRemaining){
case 16:
                registerBeingParsed = cpuidCacheInfo.reg.eax;
                break;
case 12:
                registerBeingParsed = cpuidCacheInfo.reg.ebx;
                break;
case 8:
                registerBeingParsed = cpuidCacheInfo.reg.ecx;
                break;
case 4:
                registerBeingParsed = cpuidCacheInfo.reg.edx;
                break;
 default:
                // Already parsing a register.
                break;
}

// Determine number of bytes in the descriptor,
// extract the descriptor, and shift the
// register begin parsed.
if ((registerBeingParsed && 0xC000) == 0xC000)
{
                // 32 bit descriptor
                descriptorLength = 4;
                descriptor = registerBeingParsed;
                // no need to shift
}
else if ((registerBeingParsed && 0xC000) == 0x8000)
{
                // 16 bit descriptor
                descriptorLength = 2;
                descriptor = registerBeingParsed >> 16;
                registerBeingParsed <<= 16;
}
else
{
                // must 8 bit descriptor
                descriptorLength = 1;
                descriptor = registerBeingParsed >> 24;
                registerBeingParsed <<= 8;
}

bytesRemaining -= descriptorLength;

// Do another call to CPUID, if necessary.
if (descriptor == CD_NULL
                && bytesRemaining == 0
                && callsRemaining > 0)
```

```
                {
                        rawCPUID(2, &cpuidCacheInfo);
                        callsRemaining--;
                        bytesRemaining = 16;
                }
        }
        return descriptor;
}

//
// getProcessorCacheDescription()
//
extern
TInt32
getProcessorCacheDescription(BOOL firstCall)
{

        if (firstCall)
        {
                callsRemaining = 0; // Assume not supported.
                bytesRemaining = 0;
                if (genuineIntel())
                {
                        rawCPUID(0, &cpuidCacheInfo);
                        if (cpuidCacheInfo.id.highestRecognized < 2)
                        {
                                // Processor does not support cache
                                // descriptor information.
                                return -1;
                        }
                        else
                        {
                                rawCPUID(2, &cpuidCacheInfo);
                                // AL contains the number of times to call,
                                // including the call we just made.
                                callsRemaining =
                                        (cpuidCacheInfo.reg.eax & 0xFF) -1;
                                // AL from the first call is not a descriptor
                                // so null it out.
                                cpuidCacheInfo.reg.eax &= 0xFFFFFF00;
                                // Indicate to parser that there is data to
                                // process.
                                bytesRemaining = 16;
                                // Return via parsing function.
                                return parseCacheDescriptor();
                        }
                }
                else
                {
                        // No CPUID instruction or is not Genuine Intel
```

```
                    return -1;
                }
            }
            else
            {
                // Return the next descriptor.
                return parseCacheDescriptor();
            }
        }
```