A Quick and Easy Way to Use CPUID at Run-time

CONTENTS:

1.0. Introduction

This purpose of this paper is to explain a quick, easy, and effective method of using structured exception handling to determine the features of a processor at run-time in order to execute code specifically for that processor. This paper gives an overview of structured exception handling, explains the syntax, and includes code that sets flags when Pentium® II, Pentium® Pro, Pentium®, and Pentium® processors with MMX^{TM} technology are detected. The code should be easily modified to detect features of future processors when they become available.

Here we use structured exception handling for C with the Microsoft compiler in Microsoft Developer Studio as part of MSVC++. Structured exception handling is also supported in C++, with more flexibility; and is probably supported in other compilers and development environments as well.

For further information about identifying Intel processors, consult the following: the CPUID instruction in Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference and the document "Intel Application Note 485 - Intel Processor Identification with the CPUID Instruction".

2.0. Overview of Structured Exception Handling

An exception is an event that disrupts a process. Both hardware and software can report exceptions; but for our case, we are interested in hardware exceptions such as the execution of illegal instructions. Software exceptions occur in special situations in Windows 95 and Windows NT and can be user defined by calling Microsoft's RaiseException function.

From Microsoft's Visual C++ Books Online:
"There are actually two kinds of “exception handlers”:
· Exception handlers, which can respond to or dismiss the exception
· Termination handlers, which are called when an exception causes termination inside a block of code

These two types of handlers are distinct, yet they are closely related through a process called “unwinding the stack.” When an exception occurs, Windows 95 and Windows NT look for the most recently installed exception handler that is currently active. The handler can do one of three things:
· Pass control to other handlers (fail to recognize the exception).
· Recognize but dismiss the exception.
· Recognize and handle the exception.

The exception handler that recognizes the exception may not be in the function that was running when the exception occurred. In some cases it may be in a function much higher on the stack. The currently running function, as well as all functions on the stack frame, are terminated. During this process, the stack is “unwound”: local variables of terminated functions, unless they are static, are cleared from the stack.
As it unwinds the stack, the operating system calls any termination handlers you’ve written for each function. Use of a termination handler gives you a chance to clean up resources that otherwise would remain open due to abnormal termination. If you’ve entered a critical section, you can exit in the termination handler. If the program is going to shut down, you can perform other housekeeping tasks such as closing and removing temporary files."

We use exception handlers for the example presented in this paper.

3.0. Syntax of Exception Handlers

From Microsoft's Visual C++ Books Online:
"The structure for C exception handlers is shown in the following syntax:

```
__try {
statement-block-1
}
__except ( filter ) {
```

statement-block-2
}
The statements in statement-block-1 are executed unconditionally. During execution of statement-block-1, the exception handler defined by filter and statement-block-2 is active (it becomes the current exception handler).
If an exception occurs during execution of statement-block-1, including any function called directly or indirectly, the system gives control to the current exception handler — unless a handler with higher precedence takes control.

When an exception handler takes control, the system first evaluates the filter. One of the powerful features of structured exception handling is that although filter is evaluated out of normal program sequence (often during execution of another function), filter can refer to local variables within its scope just as any C expression. After filter is evaluated, the next action depends on the value returned.

The values of filter with their descriptions are given below:

EXCEPTION_CONTINUE_SEARCH (0) Passes control to exception handler with next highest precedence. The handler has declined to recognize the exception.
EXCEPTION_CONTINUE_EXECUTION(–1) Dismisses exception, and continues execution at the location where the exception was raised.
EXCEPTION_EXECUTE_HANDLER (1) Handles exception by executing statements in statement-block-2. Execution then falls through to the end of this statement block.

If the value of filter is EXCEPTION_EXECUTE_HANDLER, execution does not resume where the exception was raised, but falls through to the end of statement-block-2 after it is executed. All blocks and function calls nested inside statement-block-1 are terminated before statement-block-2 is entered."

We use EXCEPTION_EXECUTE_HANDLER for the example presented in this paper.

4.0. Determining Processor Type, Family, Model, Stepping, and Features

In this example, we want to set flags based on what processor is detected at run-time in order to execute code specific to that processor and achieve the highest performance. We also want to discover if a processor has MMX technology in order to execute code with MMX instructions on either a Pentium II processor or a Pentium processor with MMX technology. The functions that accomplish these tasks are called GetProcessorType (to get the processor version and features), CheckMMXTechnology (to check for MMX technology and ensure MMX instructions can be executed), and EnumerateProcessorType (to set a flag indicating the processor detected).

In both GetProcessorType and CheckMMXTechnology, we set up for and try to execute the CPUID instruction to get the processor version and features. In GetProcessorType, we are only interested in the processor type, family, model, and stepping; so we save the version info returned in the eax register. In CheckMMXTechnology, we are only interested in the processor features; so we save the version info returned in the edx register. If the CPUID instruction can execute, the function will return the requested information about the processor; if not, it will return FALSE. In CheckMMXTechnology, we also verify that an MMX instruction can be executed if the processor has MMX technology. We do this because there is a possibility that floating-point emulation could be on which would not allow MMX instructions to be executed.

The CPUID instruction is broken down into its assembly opcodes using "_asm" and "_emit". "_asm" is used to include assembly code within a C or C++ module; and "_emit" is used to define a single immediate byte (similar to DB in MASM).

```
// Globals:

BOOL gIsPentiumProcessor = FALSE;

BOOL gIsPentiumProProcessor = FALSE;

BOOL gIsPentiumIIProcessor = FALSE;

BOOL gHasMMXTechnology = FALSE;


DWORD gProcessorType;


// -------------------------------------------------------------------------

// return values for GetProcessorType

//      Type (bits 13-12), Family (bits 11-8), Model (bits 7-4), Stepping (bits 3-0)

//

//      T = 00, F = 0101, M = 0001, for Pentium Processors (60, 66 MHz)

//      T = 00, F = 0101, M = 0010, for Pentium Processors (75, 90, 100, 120, 133,

//      T = 00, F = 0101, M = 0100, for Pentium Processors with MMX technology

//

//      T = 00, F = 0110, M = 0001, for Pentium Pro Processor

//      T = 00, F = 0110, M = 0011, for Pentium II Processor

//

//      T = 00 for original OEM processor

//      T = 01 for Intel OverDrive Processor

//      T = 10 for dual processor

//      T = 11 is reserved


DWORD GetProcessorType(void)

{

        DWORD retval;
        __try {
                _asm {
                        mov eax, 1              // set up CPUID to return processor
                                                //      0 = vendor string, 1 = vers
                        CPUID                   // code bytes = 0fh,  0a2h
                        and eax, 03fffh         // type, family, model, stepping re
                        mov retval, eax
                }
        } __except(EXCEPTION_EXECUTE_HANDLER) {retval = 0;}

        return retval;
}
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
BOOL CheckMMXTechnology(void)
{
        BOOL retval = TRUE;
        DWORD RegEDX;

        __try {
                _asm {
                        mov eax, 1      // set up CPUID to return processor version
                                        //      0 = vendor string, 1 = version info
                        CPUID           // code bytes = 0fh,  0a2h
                        mov RegEDX, edx // features returned in edx
                }
        } __except(EXCEPTION_EXECUTE_HANDLER) { retval = FALSE; }

        if (retval == FALSE)
                return FALSE;           // processor does not support CPUID
        if (RegEDX & 0x800000)          // bit 23 is set for MMX technology
        {
           __try { _asm emms }          // try executing the MMX instruction "emms"
           __except(EXCEPTION_EXECUTE_HANDLER) { retval = FALSE; }
        }

        else
                return FALSE;           // processor supports CPUID but does not su

        // if retval == 0 here, it means the processor has MMX technology but
        // floating-point emulation is on; so MMX technology is unavailable

        return retval;
}
// ----------------------------------------------------------------------------
```

Possibly placed in "WinMain":

```
gProcessorType = GetProcessorType();
gHasMMXTechnology = CheckMMXTechnology();

// ----------------------------------------------------------------------------
void EnumerateProcessorType(void)
{
        DWORD type, family, model, stepping;

        type     = (gProcessorType>>12) & 0x3;
        family   = (gProcessorType>>8)  & 0xf;
        model    = (gProcessorType>>4)  & 0xf;
        stepping =  gProcessorType      & 0xf;

        if (family == 5)
```

```
                gIsPentiumProcessor = TRUE;

        else if (family == 6 && model == 1)

                gIsPentiumProProcessor = TRUE;

        else if (family == 6 && model == 3)

                gIsPentiumIIProcessor = TRUE;

}

// ---------------------------------------------------------------------------
```

5.0. Conclusion

This paper explained a quick, easy, and effective method for determining the features of a processor at run-time in order to execute code specifically for that processor. The paper presented an overview of structured exception handling, explained the syntax, and included code that sets flags when Pentium II, Pentium Pro, Pentium, and Pentium processors with MMX technology are detected. The code should be easily modified to detect features of future processors when they become available.