Assembling 16-Bit Code in a 32-Bit Code Segment for Windows® CE Bootstrap

Application Note

by Dave Tobias

This application note explains how to assemble 16-bit code in a 32-bit code segment for Windows® CE bootstrap. This information is useful for those using a 32-bit microcontroller such as an Élan™SC400 or ÉlanSC410 microcontroller.

The standard build process that comes with the OEM adaptation kit for Windows CE requires that you use the assembler in flat, 32-bit mode. However, the CPU boots into 16-bit mode.

Microsoft® compensates for this discrepancy by keeping the 16-bit code as short as possible and also by using macros to force the assembler into generating instructions for 16-bit mode.

A good understanding of this workaround process is required to successfully add chipset-specific initialization code to the 16-bit code section.

The execution unit of the CPU controls how the CPU processes a given opcode. For a 386 (or later) processor, the execution unit operates in either 16-bit mode or 32-bit mode.

The execution unit considers instruction size as two elements: address size and data size. The execution unit allows individual control over these two elements on an instruction-by-instruction basis.

When in real mode or V8086 mode, both the data and address, which will be manipulated by a particular opcode, default to being processed as 16-bit quantities because descriptors are not used in real or V8086 modes.

When in protect mode, the default instruction size depends on the default (D) bit, which is found in all code segment descriptors, and which must be set up by the systems programmer.

Regardless of whether the default instruction size is 16 bit or 32 bit, the execution unit can always execute instructions using either 16-bit or 32-bit operands and/or addresses. In fact, a given opcode can be executed using any combination of operand size and data size via the use of override prefix bytes. To individually change the default address size and/or default data size of a single instruction, either one or two prefix bytes must precede the instruction.

The two prefix bytes are 66h and 67h:

- 66h changes the size of the instruction operand.
- 67h changes the size of the instruction address.

Separating the size of the operand from the size of the address allows programmers to write instructions such as the following:

```
mov   ax, [ebx]
```

In 16-bit mode, the assembler encodes the preceding instruction as 67 8B 03. This is because the address size is not the default 16 bits, so the assembler adds a 67h address prefix.

In 32-bit mode, the assembler encodes the instruction as 66 8B 03. This is because the operand size is not the default 32 bits, so the assembler adds a 66h operand prefix.

The assembler is designed to generate correct prefixes. The programmer defines the mode in which the assembler is operating by using the USE16 or USE32 keyword in a segment definition.

Note that simply providing the assembler with one of these directives does not control whether the CPU hardware will actually operate in 16-bit mode or 32-bit mode. This hardware setup must be done manually, and is beyond the scope of this application note. Specifying a USE16 or USE32 segment directive simply tells the assembler to generate code *as if* the CPU hardware is in 16-bit mode or 32-bit mode, respectively.

Unfortunately, the assembler does not allow the programmer to switch back and forth between these assumptions in a single segment. NASM, the netwide assembler, does allow this switching via the USE16 and USE32 assembler pseudo-operation codes. (NASM is a free assembler available via the internet.)

The "clean" way to approach this problem if the assembler does not support arbitrary switching between USE16 and USE32 is to generate two segments, a 16-bit segment and a 32-bit segment.

This document contains information on a product under development at Advanced Micro Devices. The information is intended to help you evaluate this product. AMD reserves the right to change or discontinue work on this product without notice.

Publication# **21642** Rev: **A** Amendment/**0**
Issue Date: **November 1997**

However, the linker and/or build process may not always support this approach.

Here are the steps required to modify and integrate working 16-bit code:

1. Use macros to define the prefix bytes:

   ```
   OpPrefix  Macro
       db    66h
   ENDM


   AddrPrefixMacro
       db    67h
   ENDM
   ```

2. Prefix all instructions that use 32-bit operands or addressing with `OpPrefix` and/or `AddrPrefix`, as appropriate. The assembler will not automatically insert the prefixes because it thinks it is in 32-bit mode, even though the CPU is in 16-bit mode.

3. Change all 16-bit instructions into pseudo-32-bit instructions to keep the assembler from inserting unwanted `OpPrefixes`/`AddrPrefixes`.

   For example:

   ```
   out    22h,ax    --->  out   22h,eax
   ```

   Because the assembler thinks the CPU is in 32-bit mode, the assembler will assemble the `out 22h,eax` without a prefix, which the CPU (in 16-bit mode) will correctly interpret as `out 22h,ax`.

4. Important! While performing step 3, watch for instructions that have immediate 16-bit data. If you do something like the following example, you will get very bad results:

   ```
   mov    ax,ConstValue  --->
   mov    eax,ConstValue
   ```

   The assembler thinks it is in 32-bit mode, so it will make the constant in the instruction four bytes. The computer will interpret the last two of those bytes (00h 00h) as the *first* two bytes of the *next* instruction.

   There are several methods for working around this. One method is to create a macro:

   ```
   LOADAX macro value
       db    0B8h
       dw    value
   ENDM
   ```

   Another method is to code the instructions as 32-bit instructions:

   ```
   OpPrefix
   mov    eax,ConstValue
   ```

   Of course, this does not work if you want to preserve the upper half of `eax`. If the upper half of `eax` needs to be preserved, you could do it in two instructions:

   ```
   OpPrefix
   and    eax,0FFFF0000h
   OpPrefix
   or     eax,ConstValue
   ```

5. Check the assembly listing to ensure the following:
   - The instructions assembled as you expected them to.
   - *All* conditional jumps (e.g., `jz`, `jnz`, etc.) are assembled in their short form (single byte offset). If a conditional jump is assembled in its long form, the prefix byte will not be correct. For unconditional jumps, the fact that the offset was extended by two bytes is immaterial (will not affect CPU operation).

6. Strip all calls out of the code (replace with macros), or insert prefix instructions to ensure not only that the size of the offset is correct (the last two bytes of the four-byte offset are not treated as the next instruction), but also that the calls and returns match in usage of the stack.

7. Finally, use the disassembler in DOS debug to disassemble and/or step through a portion of the code to make sure that it is doing what you expect it to do.

Trademarks