Real Mode Tools & Techniques

Overview

Emulators and specialized compiler toolchains are common when working with embedded style programmed I/O. The x86 line is still in use in embedded applications (e.g., NASA rovers and space station components), and the line has a long history of development, so toolchains continue to be supported, and in some cases even expanded, over time. This document covers tools and techniques used with the x86 line in its real mode (with direct hardware access).

Tools

Emulators and specialized compiler toolchains are common when working with embedded style programmed I/O. Here we discuss the DOSBox emulator and Open Watcom C/C++ toolchains.

DOSBox Emulator

The DOSBox platform is popular for emulation of early x86 processors. It can emulate various CPU models, processor speeds, and peripherals. DOSBox's emulation also features the ability to connect directly to the underlying filesystem, which makes moving data and programs into its "disk space" (***mostly***) effortless. If the host filesystem changes, DOSBox can also rescan the filesystem to detect those changes.

```
Z:\>mount q .\foo-folder
Drive Q is mounted as local directory .\foo-
folder
… Do some stuff on local / host filesystem …
Z:>rescan
Drive cache cleared.

Z:>_
```

Open Watcom C/C++

While some development continues via "end-of-life" compiler suites (e.g., the popular Borland Turbo C++ IDE and compiler), the WATCOM corporation released source for their C/C++ suite. As a result, development of Open Watcom C/C++ continues today, with some support integrated for modern versions of the C and C++ standards. Additionally, Open Watcom is built for several platforms and targets, and it can build from and target DOS real mode. (Its other host platforms – including Windows and Linux – can also

```
W:\samples\cplbexam\complex\friend$ wmake
…
creating a DOS executable

W:\samples\cplbexam\complex\friend$   wcl
*.cpp
…
creating a DOS executable
```

build for the DOS real-mode targets.) Its most critical tools include **wmake** (make system), **wlib** (archiver), and **wcl** (compiler and linker) which correspond roughly to **make**, **ar**, and **gcc/g++** respectively.

Programming Techniques

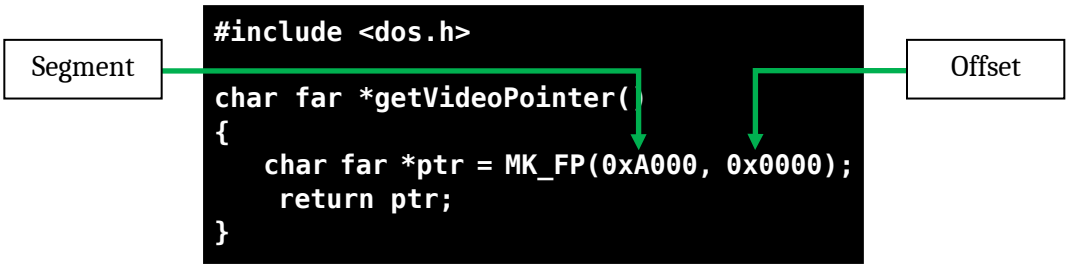
Real mode programming brings with it several important challenges. This section addresses addressing approaches and use of inline assembly.

Segmented Addressing

Real mode uses 16-bit addresses. Normally, such an addressing scheme would limit memory to 64KiB – barely enough even for a typical VGA memory page. To get around this limitation, the x86 processor line uses ***segmentation*** to expand the memory address space. In 16-bit segmented addressing, a memory location is made up of a ***segment*** and an ***offset*** – each 16 bits. The segment is shifted left by 4 bits, then the offset is added – resulting in a 20-bit address space (max of 1 MiB). For example, to access video memory (**`0xA0000`**), a program could use segment **`0xA000`**:

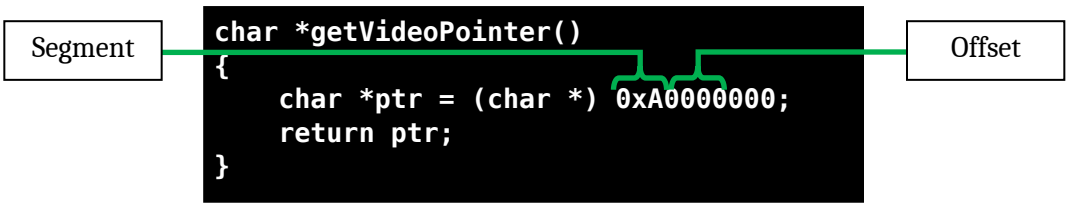
Segment	Offset	Shifted Segment	Target Address
0xA000	0x0010	0xA0000	0xA0010

When a typical real mode program needs to access memory outside of its segment, at a specific address, it must generate a ***far pointer*** that encompasses the segment and offset together:



However, far pointer systems can be cumbersome to work with, and the type system can be confusing and error-prone. An alternative is to use the large memory model, which allows the segment and offset to be stored in a single 32-bit pointer:

To use the large



memory model, append "**`-ml`**" to the compile command (e.g., "**`wcl -ml`**").

Inline Assembly

When working directly with hardware, it is often more efficient and/or more convenient to work directly in assembly for small sections of code. In these cases, we can use **inline assembly**. Most compilers support inline assembly, but unfortunately the syntax and invocation methods vary. Here, we cover the Open Watcom C/C++ inline assembly syntax and invocation.

```
// Inline assembly – country code example
#include <stdint.h>

#define KEYBOARD_HANDLER 0x16

int getKeyboardCountryCode()
{
  uint16_t getCodeFunction = 0x5001;
  int country;

  _asm
  {
    mov ax, getCodeFunction // Get country code
     int KEYBOARD_HANDLER     // Invoke keyboard
handler
    mov [country], bx      // Copy code to variable
  }

  return country;
}
```

This example uses Intel assembly syntax, as is common for real-mode (and generally DOS) compilers. Note that **the operand order is reversed** compared to AT&T (GNU) syntax! Otherwise, the syntax is very similar. In Watcom inline assembly, a developer can use local C/C++ variables in instructions. The ***value*** at an address can be accessed by simply using the variable name (.e.g, **`getCodeFunction`**). A program can get the address of a variable (for writing) by enclosing it in square brackets (e.g., **`[country]`**). As in other sections of your code, you can also use **`#define`** statements that the preprocessor will replace before compilation (e.g., **`KEYBOARD_HANDLER`**).