DOS Programming Guide with DJGPP - by freevanx

使用 DJGPP 编写 DOS 程序

freevanx

作者： 本文由 freevanx（freevanx@gmail.com）起草，拥有版权。未经允许，不得转载。含有内部资料或者技术的部分文章内容，只开放给使用本公司 BIOS 的开发者参考。含有本文链接的地址可在下面找到：
http://docs.google.com/View?docID=0AXM7WqiAoyr_ZDk3dnI4el8zOTNobmt0a3BkOQ&a

2010/11/01	V1.2 Add dos interrupt sample, update the PCIE read sample
2009/6/14	V1.1 Add PCIe & CMOS functions
2009/6/12	V1.0 draft version

参考文档

DJGPP: http://www.delorie.com/djgpp/
GCC : http://gcc.gnu.org/
Glibc : http://www.gnu.org/software/libc/
DOS Extender: http://en.wikipedia.org/wiki/DOS_extender
GNU Make : http://www.gnu.org/software/make/
本文代码 : http://docs.google.com/View?id=d97vr8z_53c7k6bdfs

前言

写这篇文章之前，看到网络上很多同行都寻找编写 DOS 程序的工具，借此机会向大家介绍这样一款既免费无责任，又非常好用的工具。由于 BIOS 开发的特殊性，DOS 作为古老的 OS 之一，其非常简单的特点很适合一些测试工具的运行，大部分情况下，可以简化测试流程，缩短测试时间，所以在量产测试中，DOS 下测试是一个非常重要且简单的测试过程。当然在 EFI BIOS 下，可以将测试程序写成 EFI Shell APP。

现在开发传统 DOS 下的程序，常用的是 Borland 的 TC 或者 BCC，或者是 MS 的 VC1.52，TC 只能写 16 位程序，如果要用到 32 位数据的话，就非常麻烦，不但要用嵌汇编，还要用机器码来实现 32 位的操作。其他的软件，即使可以使用 32 位的数据，当面临下面一种情况时，同样也需要很麻烦操作： 由于 DOS 系统运行在 real mode 下，所以一般的程序不能访问大于 1M 的地址，但是在开发一些程序的时候需要访问 1M 以上直至 4GB 的地址，这个时候怎么办？传统的 DOS 程序会另外使用一个叫做 DOS Extender 的程序，用来提供 protect mode 的服务，其中最有名的当属 DOS4GW 这个程序，但是这个软件是商业软件，开发早停，收费不止，而且这个程序笨重难用，开源爱好者们开发了一款 open source 的软件来代替他，名叫 DOS/32 Advanced DOS Extender，借助 DOS Extender，DOS 下的程序可以访问高达 4GB 的地址空间，解决了不能访问高地址的问题，不过解决的同时带来了另一个问题，就是 DOS Extender 提供的是一系列 library 形式的 function，程序中需要调用这些 function 才能达到目的。

今天要介绍的 DJGPP，同时解决了以上两个问题，首先 DJGPP 中可以使用 32 位数据，所以不用自己写机器码，其次 DJGPP 编译的 DOS 程序，默认运行在 protect mode 下，所以可以访问 4GB 的地址空间，当然 DJGPP 也需要一个 DOS Extender，不过 DJGPP 不要你额外写代码，因为编译器把 Extender 绑定在一起，这样你只需要关注你自己的代码就可以了，当然这样也有一个坏处，就是不管你写什么样的程序，你总是需要这样一个 Extender，也就是说，你分发的程序至少有两个文件，其中一个是 Extender 的可执行文件。

介绍

DJGPP 是基于 GCC 的一个 DOS porting，所以接受 GCC 语法，GCC 设定，如果你熟悉 Linux GCC 的使用或者使用过 Windows 下的 Mingw，那么你可以很容易的使用 DJGPP 来编译你的程序，即使你以前对 GCC 一无所知，你也可以相信，follow freevanx 的介绍，你可以轻易的学会如何使用 DJGPP 并提高你的开发效率。

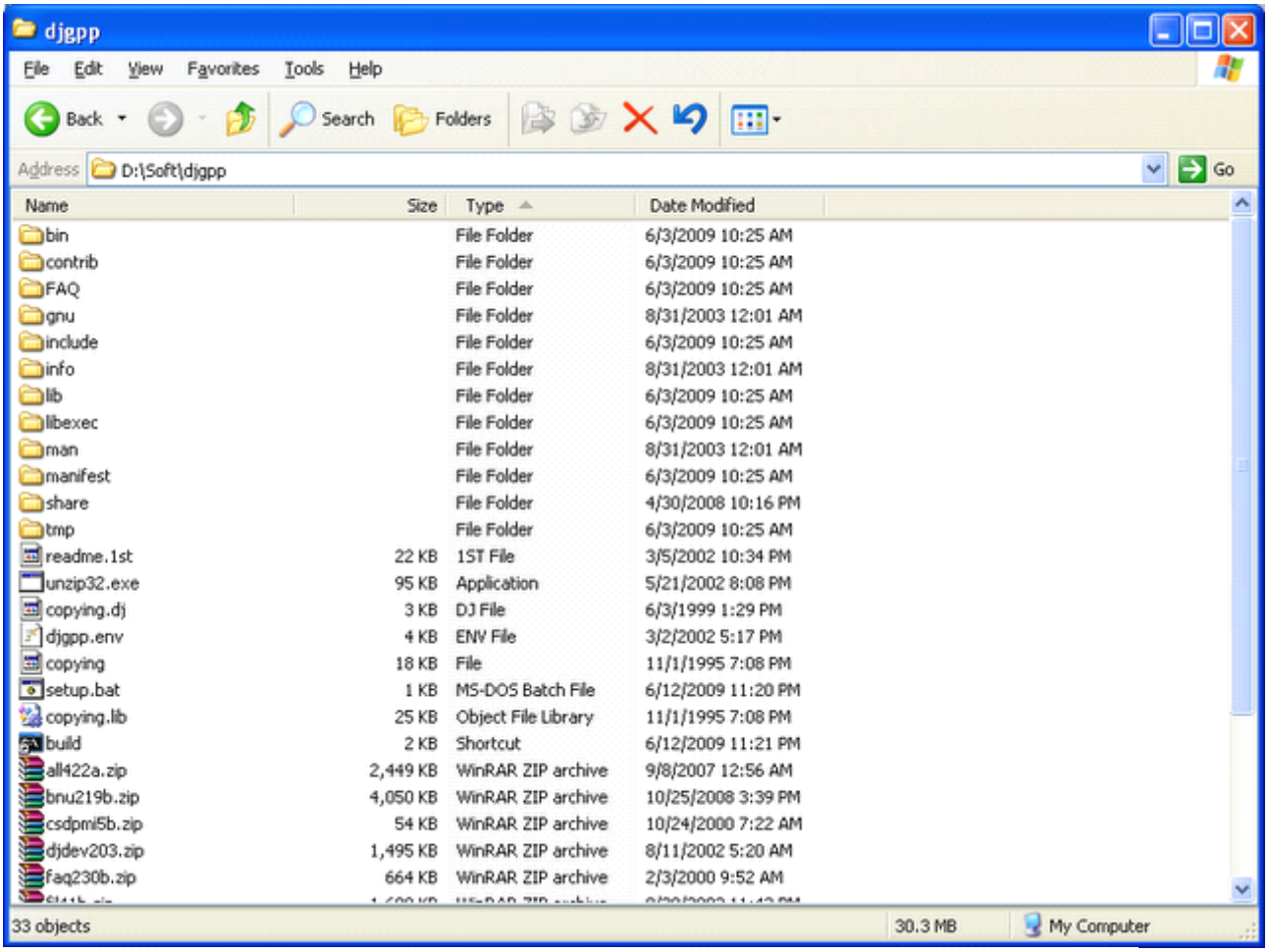
关于 DOS Extender 的携带。所有使用 DJGPP 编译的 DOS 程序，都需要配合支持 DPMI 接口的 DOS Extender 使用，DJGPP 默认编译过的程序使用 CSDPMI 这个程序，所以你在 DOS 下使用的时候需要同时携带 cwsdpmi.exe 使用。你可以从 DJGPP 站点中选择一个 ftp 下载 csdpmi 的软件包，放在 ftp v2misc 目录下的 csdpmi5b.zip。也可以直接从这里下载 http://djgpp.linux-mirror.org/v2misc/。除了 CSDPMI 之外，还有另外一个 DPMI Service 程序可供选择，即 pmode，如果想是程序运行更加高效，可以使用 pmode，使用方法请参考 pmode13b.zip 中的文档。

关于 Ring0。在与 BIOS 有关的 DOS 程序中，很多软件都需要运行在 Ring0 下，也就是说可能很多软件都或多或少需要依赖于特权指令，DJGPP 的 DOS Extender csdpmi 中，如果使用默认的 CWSDPMI.exe 的话，默认是运行在 protect mode 的 user mode，即 Ring3，如果你需

要运行在 Ring0 下，请使用 csdpmi 包中的 CWSDPR0.EXE，将这个软件重命名成 CWSDPMI.EXE 随你的软件分发就可以了。
也可以使用 CWSDPMI 将程序和 DOS Extender 绑在一起，这样在软件分发时只需要分发一个文件就可以了，详情请参考 CWSDPMI 的文档。

安装

DJGPP 的主站：http://www.delorie.com/djgpp/
使用 DJGPP 提供的 **Zip Picker** 选择你要下载的包,下载后请选择解压到当然文件夹，然后就可以看到像下面这样的内容。（见下图 ）



注意： Zip Picker 选择给出的软件并不包含 CSDPMI，所以要单独下载这个包

设置环境。DJGPP 需要一些简单的设置，需要将编译器的路径添加到 PATH 里面，并且建立一个名叫 DJGPP 的环境变量指向 djgpp.env 文件。以下是我使用的一个 batch file 的内容

```
@REM ================================================
@REM    file setup.bat
@REM   seting build environment for DJGPP
@REM   create by freevanx
@REM ================================================
@set ROOT=%CD%
@set PATH=%PATH%;%ROOT%\bin;
@set DJGPP=%ROOT%\djgpp.env
@echo "DJGPP build environment is ready!"
@cmd.exe
```

然后，我们可以测试一下我们的环境是否 OK，双击刚才建立的 setup.bat，然后输入下面命令：

```
D:\Soft\djgpp>setup
"DJGPP build environment is ready!"

D:\Soft\djgpp>cd bin

D:\Soft\djgpp\bin>gcc
gcc.exe: no input files

D:\Soft\djgpp\bin>gcc --version
gcc.exe (GCC) 4.3.2
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

D:\Soft\djgpp\bin>
```

说明环境建立完毕。可以进行编译了。。

第一个程序： **Hello World**

DJGPP 是 GCC 的一个 porting，所以编译器是 gcc，不必对 gcc 感到畏惧，使用你常用的 C 语法就可以了，建立一个 C 文件输入以下内容

```
#include <stdio.h>

int main(void)
{
  printf("Hello, world!\n");
  return 0;
}
```

编译链接：

```
D:\Soft\djgpp\bin>gcc hello.c -o hello.exe
```

在 DOS 下运行： C:\ > hello

```
Hello, World!
```

当然，如果程序比较大，有很多个源文件，可以使用 makefile, DJGPP 中包含的 make

是属于 GNU Make，功能非常强大，你可以写一个非常简单的 makefile 完成很复杂的任务。

进阶

在稍微熟悉了 gcc 以后，我们继续下面的程序，下面将提供一些在 BIOS 开发中常用的函数，例如 IO Memory==来辅助大家开发。

必要的头文件　以下是一个头文件示例，要包含哪些头文件，取决于你要写的程序

```
/*++
************************************************
*             Copyright (c) 2008~2009 freevanx.  All rights reserved.
*
*                    freevanx@gmail.com
*
*    This file is distributed under BSD liscense
*    File:
*    Description:    remember to use cwsdpr0.exe instead of cwsdpmi.exe
*
--************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/nearptr.h>
#include <unistd.h>

#define    QIO_PCI_ADDR_PORT    0xCF8
#define    QIO_PCI_DATA_PORT    0xCFC

typedef    unsigned char    UCHAR, UINT8;
typedef    unsigned short    USHORT, UINT16;
typedef    unsigned long    ULONG, UINT32;
typedef    unsigned int    UINT;
```

访问 IO 端口函数的封装

```
#define __BUILDIO(bwl, bw, type)              \
static inline void out##bwl(unsigned type value, int port)    \
{                              \
   out##bwl##_local(value, port);              \
}                              \
                           \
static inline unsigned type in##bwl(int port)          \
{                              \
   return in##bwl##_local(port);             \
}

#define BUILDIO(bwl, bw, type)                    \
```

```
static inline void out##bwl##_local(unsigned type value, int port)    \
{                                \
   asm volatile("out" #bwl " %" #bw "0, %w1"       \
          : : "a"(value), "Nd"(port));          \
}                                \
                                \
static inline unsigned type in##bwl##_local(int port)          \
{                                \
   unsigned type value;                    \
   asm volatile("in" #bwl " %w1, %" #bw "0"       \
          : "=a"(value) : "Nd"(port));          \
   return value;                        \
}                                \
                                \
                                \
__BUILDIO(bwl, bw, type)                    \
                                \
static inline void outs##bwl(int port, const void *addr, unsigned long count) \
{                                \
   asm volatile("rep; outs" #bwl                \
          : "+S"(addr), "+c"(count) : "d"(port));       \
}                                \
                                \
static inline void ins##bwl(int port, void *addr, unsigned long count)    \
{                                \
   asm volatile("rep; ins" #bwl                \
          : "+D"(addr), "+c"(count) : "d"(port));       \
}

BUILDIO(b, b, char)
BUILDIO(w, w, short)
BUILDIO(l, , int)
```

以上一些宏定义提供下面几个函数：
读 IO Port

```
static inline unsigned char inb(int port);
static inline unsigned short inw(int port);
static inline unsigned int inl(int port);
```

写 IO Port

```
static inline void outb(unsigned char value, int port);
static inline void outw(unsigned short value, int port);
static inline void outl(unsigned long int, int port);
```

访问 Memory 函数的封装：

```
static inline unsigned char readb( unsigned long addr)
{
   unsigned char    ret=0;
    if (__djgpp_nearptr_enable())
    {
      unsigned char *phy_addr = (unsigned char *)
                                  (__djgpp_conventional_base + addr);
       ret = *phy_addr;
      __djgpp_nearptr_disable();
      return ret;
    }
    return 0xFF;
}

static inline unsigned short readw( unsigned long addr)
{
   unsigned short    ret=0;
    if (__djgpp_nearptr_enable())
    {
      unsigned short *phy_addr = (unsigned short *)
                                  (__djgpp_conventional_base + addr);
       ret = *phy_addr;
      __djgpp_nearptr_disable();
      return ret;
    }
    return 0xFF;
}

static inline unsigned long readl( unsigned long addr)
{
   unsigned long    ret=0;
    if (__djgpp_nearptr_enable())
    {
      unsigned long *phy_addr = (unsigned long *)
                                  (__djgpp_conventional_base + addr);
       ret = *phy_addr;
      __djgpp_nearptr_disable();
      return ret;
    }
    return 0xFF;
}

inline void writeb(unsigned char value, unsigned long addr)
{
    if (__djgpp_nearptr_enable())
    {
      unsigned char *phy_addr = (unsigned char *)
                                  (__djgpp_conventional_base + addr);
       *phy_addr = value;
      __djgpp_nearptr_disable();
    }
}
```

```
inline void writew(unsigned short value, unsigned long addr)
{
    if (__djgpp_nearptr_enable())
    {
      unsigned short *phy_addr = (unsigned short *)
                                 (__djgpp_conventional_base + addr);
       *phy_addr = value;
      __djgpp_nearptr_disable();
    }
}

inline void writel(unsigned long value, unsigned long addr)
{
    if (__djgpp_nearptr_enable())
    {
      unsigned long *phy_addr = (unsigned long *)
                                 (__djgpp_conventional_base + addr);
       *phy_addr = value;
      __djgpp_nearptr_disable();
    }
}
```

以上的函数提供下列函数的封装
读 Memory 函数

```
static inline unsigned char readb( unsigned long addr);
static inline unsigned short readw( unsigned long addr);
static inline unsigned long readl( unsigned long addr);
```

写 Memory 函数

```
inline void writeb(unsigned char value, unsigned long addr);
inline void writew(unsigned short value, unsigned long addr);
inline void writel(unsigned long value, unsigned long addr);
```

访问 PCI 设备函数的封装

```
UINT pci_bus_read_config_dword(UCHAR bus, UCHAR devfn, UCHAR offset, ULONG
* pvalue)
{
   ULONG    addr = 0x80000000;

   if (pvalue == NULL)
      return 1;       // Fix Here
   offset = offset/4 * 4;
   addr = addr + (bus << 16) + (devfn << 8) + offset;
```

```
   outl(addr, QIO_PCI_ADDR_PORT);
   *pvalue = inl(QIO_PCI_DATA_PORT);
   return 0;
}

UINT pci_bus_read_config_byte(UCHAR bus, UCHAR devfn, UCHAR offset, UCHAR *
pvalue)
{
   ULONG    value =0;
   UINT ret=0;
   UCHAR    longoffset= offset/4 * 4;
   UCHAR    index = offset - longoffset;

   if (pvalue == NULL)
      return 1;       // Fix Here

   ret = pci_bus_read_config_dword(bus, devfn, longoffset, &value);
   *pvalue = (UCHAR) ((value >> (index*8)) & 0xFF);
   return 0;
}
UINT pci_bus_read_config_word(UCHAR bus, UCHAR devfn, UCHAR offset, USHORT
* pvalue)
{
   ULONG    value =0;
   UINT ret=0;
   UCHAR    longoffset= offset/4 * 4;
   UCHAR    index = offset - longoffset;

   if (pvalue == NULL)
      return 1;       // Fix Here

   ret = pci_bus_read_config_dword(bus, devfn, longoffset, &value);
   *pvalue = (USHORT) ((value >> (index*8)) & 0xFFFF);
   if (index >= 3)
   {
      ret = pci_bus_read_config_dword(bus, devfn, longoffset+4, &value);
      *pvalue += (USHORT) ((value  & 0xFF) <<8);
   }

   return 0;
}

UINT pci_bus_write_config_dword(UCHAR bus, UCHAR devfn, UCHAR offset,
ULONG  value)
{
   ULONG    addr = 0x80000000;

   offset = offset/4 * 4;
   addr = addr + (bus << 16) + (devfn << 8) + offset;
   outl(addr, QIO_PCI_ADDR_PORT);
   outl(value, QIO_PCI_DATA_PORT);
   return 0;
}
```

```
UINT pci_bus_write_config_byte(UCHAR bus, UCHAR devfn, UCHAR offset,
UCHAR  value)
{
   ULONG    temp_value =0;
   UINT ret=0;
   UCHAR    longoffset= offset/4 * 4;
   UCHAR    index = offset - longoffset;
   UCHAR* pvalue = (UCHAR*)&temp_value;

   ret = pci_bus_read_config_dword(bus, devfn, longoffset, &temp_value);
   *(pvalue + index ) = value;
   ret = pci_bus_write_config_dword(bus, devfn, longoffset, temp_value);

   return 0;
}
UINT pci_bus_write_config_word(UCHAR bus, UCHAR devfn, UCHAR offset,
USHORT  value)
{
   ULONG    temp_value =0;
   UINT ret=0;
   UCHAR    longoffset= offset/4 * 4;
   UCHAR    index = offset - longoffset;
   UCHAR* pvalue = (UCHAR*)&temp_value;
   if (index <3)
   {
      ret = pci_bus_read_config_dword(bus, devfn, longoffset, &temp_value);
      *(USHORT *)(pvalue + index ) = value;
      ret = pci_bus_write_config_dword(bus, devfn, longoffset, temp_value);
   }
   else
   {
      ret = pci_bus_read_config_dword(bus, devfn, longoffset, &temp_value);
      *(pvalue + index ) =(UCHAR) (value & 0xFF);
      ret = pci_bus_write_config_dword(bus, devfn, longoffset, temp_value);
      ret = pci_bus_read_config_dword(bus, devfn, longoffset+4, &temp_value);
      *pvalue=(UCHAR) (value >> 8);
      ret = pci_bus_write_config_dword(bus, devfn, longoffset+4, temp_value);
   }

   return 0;
}
```

以上函数提供以下 PCI 函数的封装

读 PCI 设备配置空间：

```
UINT pci_bus_read_config_dword(UCHAR bus, UCHAR devfn, UCHAR offset, ULONG
* pvalue);
UINT pci_bus_read_config_byte(UCHAR bus, UCHAR devfn, UCHAR offset, UCHAR *
pvalue);
```

UINT pci_bus_read_config_word(UCHAR bus, UCHAR devfn, UCHAR offset, USHORT * pvalue);

写 PCI 设备的配置空间：

UINT pci_bus_write_config_dword(UCHAR bus, UCHAR devfn, UCHAR offset, ULONG value);
UINT pci_bus_write_config_byte(UCHAR bus, UCHAR devfn, UCHAR offset, UCHAR value);
UINT pci_bus_write_config_word(UCHAR bus, UCHAR devfn, UCHAR offset, USHORT value);

注意：dword 的读写函数都只接受与 4 字节对齐 offset 值，与 PCI 设备配置空间的访问方式相对应，参见 PCI specification

接下来，是一组访问 MSR 的函数，这组函数封装了读写 CPU MSR 寄存器的操作，使用时只需要调用这些函数即可

```
static inline void readmsr(unsigned long ecx, unsigned long* eax, unsigned long* edx)
{
   asm volatile("rdmsr"
            : "=a"(*eax), "=d"(*edx)
            : "c"(ecx) );
}

static inline void writemsr(unsigned long ecx, unsigned long eax, unsigned long edx)
{
   asm volatile("wrmsr"
            :
            : "c"(ecx), "a"(eax), "d"(edx));
}
```

以上两个函数提供下列的 API
读取 CPU MSR 的值
static inline void readmsr(unsigned long ecx, unsigned long* eax, unsigned long* edx)；

写 CPU MSR 的值
static inline void writemsr(unsigned long ecx, unsigned long eax, unsigned long edx) ；

好了，以上提供了 4 组函数，通过这 4 组函数，你可以读写 IO Port，读写 Memory/MMIO, 读写 PCI 配置空间，读写 MSR，那么有了这几组函数，还有多少你做不到的事情呢？Let's go！

应用

接下来我们将使用上面提供的 PCI 函数和 IO 函数，来写一个 DOS 下的延时程序，这个延时程序使用 32 位的计时器，可以精确到毫秒级别，最长可以延时 19.99 分钟。

这里使用的计时器是 ACPI PM1 Timer,这是一个 24 位的计时器，使用一个 3.579545MHz 的时钟，刚才我们说过我们的时钟是 32bit 的，如何使用 24 位时钟实现 32 位时钟呢？看了以下代码相比就明白了吧：

```
ULONG    current_ticket=0;
ULONG    previous_ticket = 0;
ULONG    count=0;

current_ticket = previous_ticket = inl(pmbase +0x08);      // PM1 timer register

while(count < delay_ms)
{
   current_ticket = inl(pmbase +0x08);                    // PM1 timer register
   if (current_ticket >= previous_ticket)
   {
      count += (current_ticket - previous_ticket);
   }
   else
   {
      count += current_ticket;
   }
   previous_ticket = current_ticket;
}
```

count 是计数器的累计，是一个 32bit 的值，用来表示一个 32bit 的计数器，current_ticket 是当前读到 PM1 Timer 的数值，previous_ticket 是上次读到 PM1 Timer 的数值，如果当前的数值比上次独到的数值大，说明 PM1 Timer 在增长并且没有溢出 24bit 这个范围，我们只需要在 count 上加上(current_ticket - previous_ticket)这样一个经过时间的差量就可以了，当前读到的数值较小，说明 PM1 Timer 计数已经溢出 24bit 并且重新开始，那么我们要给 count 加上当前读到的数值。

在这之前，还要解决一个问题，PM1 Timer 的 IO Port 是在 PMBASE 的基础上加 8（参见南桥的 spec），而 PMBASE 是一个不定值，是在 BIOS POST 的时候设置的，常见的值是 PMBASE=0x800 或者 PMBASE=0x400，那如何获取当前系统 PMBASE 的设定呢？我们需要使用上面提供的 PCI 函数读取 LPC Bridge 上 PMBASE 的设置，参见以下代码：

```
inline static USHORT GetPMBase(void)
{
   UINT    ret = 0;
   USHORT    base =0;
   ret = pci_bus_read_config_word(0, 0xF8, 0x40, &base);
   base &= 0xFF80;
   return base;
}
```

好，大部分问题都解决了，那么我现在贴出源程序的代码，加上上面的头文件和函数定义，就可以

完成我们需要的功能了。

```
inline static USHORT GetPMBase(void)
{
   UINT    ret = 0;
   USHORT    base =0;
   ret = pci_bus_read_config_word(0, 0xF8, 0x40, &base);
   base &= 0xFF80;
   return base;
}


char  gMsg[] =  "lychee Project (c) freevanx All Rights Reserved!\n" \
             "QuikIO hardware access library!\n" \
            "ms delay utility for DOS, max delay input value is 1199879 = 19.99
min\n" \
             "    Build by DJGPP";

int verbose =0;


inline static UINT    mdelay(USHORT pmbase, ULONG ms)
{
   ULONG    current_ticket=0;
   ULONG    previous_ticket = 0;
   ULONG    count=0;

   // PM1 timer add 7159.09 every 2ms
   ULONG    delay_ms = (ms/2 *7159) + ((ms %2)? 7159/2 : 0 );
   static    int    start_flag =0;

   current_ticket = previous_ticket = inl(pmbase +0x08);  // PM1 timer register

   if (verbose)
   {
      printf("Delay start at ticket %u\n", current_ticket);
   }

   while(count < delay_ms)
   {
      current_ticket = inl(pmbase +0x08);         // PM1 timer register
      if (current_ticket >= previous_ticket)
      {
         count += (current_ticket - previous_ticket);
      }
      else
      {
         count += current_ticket;
      }
      previous_ticket = current_ticket;
   }

   if (verbose)
```

```
    {
        printf("Delay end at ticket %u\n", current_ticket);
        printf("Delay should end at count %u\n", delay_ms);
        printf("Delay end at count %u\n", count);
    }

    return 0;
}

int main(int argc, char** argv)
{
    int c =0;
    ULONG    time=0;
    USHORT    pmbase =0;
    int go_flag=0;

    while ((c = getopt(argc, argv, "t:hv")) != -1 )
    {
        switch (c)
        {
        case    't'    :
            time = strtoul(optarg, NULL, 0);
            go_flag=1;
            break;

        case 'v'    :
            verbose =1;
            break;

        case 'h'    :
        case '?'    :
            printf(gMsg);
            break;

        default:
            printf(gMsg);
        }
    }

    pmbase = GetPMBase();
    if (verbose)
    {
        printf("\n Delay %u ms\n", time);
        printf(" Found PMBASE at %04x\n", pmbase);
    }

    if (go_flag)
    {
        mdelay(pmbase, time);
    }
    return 0;
}
```

再解释一下，其中用到的 getopt 函数，是 glibc 特有的函数，MSVC 的 libc 库并没有这个函数，详细的用法可以查看 glibc 的帮助。

事实上，这个延时 Timer 的程序经过了 HW 示波器的检验，在 1ms 的精度上可以看到非常的精确，用肉眼无法发现误差。

本文中用到的所有代码都可以通过：
http://docs.google.com/View?id=d97vr8z_53c7k6bdfs 观看。

应用 2 访问 PCIe 设备配置空间

PCIe 设备拥有 4KB 的配置空间，虽然 PCIe 设备也支持按照 PCI 设备的方式访问其配置空间，不过有一些常用的 Capability 寄存器是在大于 256 的配置空间里，这样按照访问 PCI 设备的方式便无法访问这些寄存器。PCIe spec 定义了一种扩展的访问方式，其将 PCIe 设备的配置空间映射到 MMIO，通过 MMIO 去访问 PCIe 设备的配置空间。详细的访问方式请参考 PCIe spec，在访问 PCIe 设备之前，首先要知道其地址空间所 Map MMIO 的 base，即 PCIe 配置空间的基地址，这个地址通常由 BIOS 配置 chipset 决定，不同的 chipset，可能其设置的方式有不同，现在以 Intel 一种 MCH 为例，其 PCIe base 的设置是在 PCI 设备 bus 0， device 16 function 0 register 0x64 的 bit 12~23 决定，我们通过读取这个 HECBASE，得到 PCIe configuration space 的基地址。当然，在现行的很多 BIOS 中，PCIe 配置空间的基地址一般都设定在 0xE0000000 这里，如果以一般的情况考虑，可以直接使用这个地址，我们使用 GetPCIeBase 函数获得 PCIe 的基地址。参见下面的代码。

```
static inline unsigned char breadb(unsigned long addr)
{
       void * virtualAddress = NULL;
       unsigned long      pageaddr = 0;
       unsigned long      size = 0x1000;
       unsigned long      offset = 0;
       unsigned char value = 0;

       pageaddr = addr & 0xFFFFF000;
       offset = addr & 0x00000FFF;

       if (__djgpp_map_physical_memory(virtualAddress, size,
              pageaddr) == 0)
       {
              value = *((unsigned char *)((unsigned long)virtualAddress +
offset));
              return value;
       }
     return 0xFF;
}



static inline unsigned short breadw(unsigned long addr)
{
       void * virtualAddress = NULL;
       unsigned long      pageaddr = 0;
       unsigned long      size = 0x1000;
       unsigned long      offset = 0;
       unsigned short value = 0;

       pageaddr = addr & 0xFFFFF000;
```

```
	offset = addr & 0x00000FFF;

	if (__djgpp_map_physical_memory(virtualAddress, size,
		pageaddr) == 0)
	{
		value = *((unsigned short *)((unsigned long)virtualAddress +
offset));
		return value;
	}
     return 0xFF;
}


static inline unsigned long breadl(unsigned long addr)
{
	void * virtualAddress = NULL;
	unsigned long	pageaddr = 0;
	unsigned long	size = 0x1000;
	unsigned long	offset = 0;
	unsigned long value = 0;

	pageaddr = addr & 0xFFFFF000;
	offset = addr & 0x00000FFF;

	if (__djgpp_map_physical_memory(virtualAddress, size,
		pageaddr) == 0)
	{
		value = *((unsigned long *)((unsigned long)virtualAddress +
offset));
		return value;
	}
     return 0xFF;
}
inline void bwriteb(unsigned char value, unsigned long addr)
{
	void * virtualAddress = NULL;
	unsigned long	pageaddr = 0;
	unsigned long	size = 0x1000;
	unsigned long	offset = 0;
	unsigned char * address = NULL;

	pageaddr = addr & 0xFFFFF000;
	offset = addr & 0x00000FFF;

	if (__djgpp_map_physical_memory(virtualAddress, size,
		pageaddr) == 0)
	{
		address = (unsigned char *)((unsigned long)virtualAddress +
offset);
		*address =  value;
	}
}
inline void bwritew(unsigned short value, unsigned long addr)
{
	void * virtualAddress = NULL;
	unsigned long	pageaddr = 0;
	unsigned long	size = 0x1000;
	unsigned long	offset = 0;
	unsigned short * address = NULL;
```

```
	pageaddr = addr & 0xFFFFF000;
	offset = addr & 0x00000FFF;

	if (__djgpp_map_physical_memory(virtualAddress, size,
		pageaddr) == 0)
	{
		address = (unsigned short *)((unsigned long)virtualAddress +
offset);
		*address =  value;
	}
}
inline void bwritel(unsigned long value, unsigned long addr)
{
	void * virtualAddress = NULL;
	unsigned long	pageaddr = 0;
	unsigned long	size = 0x1000;
	unsigned long	offset = 0;
	unsigned long * address = NULL;

	pageaddr = addr & 0xFFFFF000;
	offset = addr & 0x00000FFF;

	if (__djgpp_map_physical_memory(virtualAddress, size,
		pageaddr) == 0)
	{
		address = (unsigned long *)((unsigned long)virtualAddress +
offset);
		*address =  value;
	}
}
```

```
inline static unsigned long GetPCIeBase(void)
{
   unsigned int    ret = 0;
   unsigned long    base =0;
   ret = pci_bus_read_config_dword(0, 0x80, 0x64, &base);
   base &= 0x00FFF000;
   base = base << 28;
   return base;

   //Commonly the PCIe base is here, but we read it from HECBASE
   //return 0xE0000000;
}

UINT pcie_bus_read_config_dword(UCHAR bus, UCHAR devfn, USHORT offset,
ULONG * pvalue)
{
    ULONG    base = 0;
    ULONG    addr = 0;

    if (pvalue == NULL)
        return 1;        // Fix Here
    base = GetPCIeBase();
    addr = base + (bus << 20) + (devfn << 12) + offset;
    *pvalue = breadl(addr);

    return 0;
```

```
}

UINT pcie_bus_read_config_word(UCHAR bus, UCHAR devfn, USHORT offset,
USHORT * pvalue)
{
    ULONG    base = 0;
    ULONG    addr = 0;

    if (pvalue == NULL)
        return 1;        // Fix Here
    base = GetPCIeBase();
    addr = base + (bus << 20) + (devfn << 12) + offset;
    *pvalue = breadw(addr);

    return 0;
}

UINT pcie_bus_read_config_byte(UCHAR bus, UCHAR devfn, USHORT offset, UCHAR
* pvalue)
{
    ULONG    base = 0;
    ULONG    addr = 0;

    if (pvalue == NULL)
        return 1;        // Fix Here
    base = GetPCIeBase();
    addr = base + (bus << 20) + (devfn << 12) + offset;
    *pvalue = breadb(addr);

    return 0;
}

UINT pcie_bus_write_config_dword(UCHAR bus, UCHAR devfn, USHORT offset,
ULONG value)
{
    ULONG    base = 0;
    ULONG    addr = 0;

    base = GetPCIeBase();
    addr = base + (bus << 20) + (devfn << 12) + offset;
    bwritel(value, addr);
    return 0;
}

UINT pcie_bus_write_config_word(UCHAR bus, UCHAR devfn, USHORT offset,
USHORT value)
{
    ULONG    base = 0;
    ULONG    addr = 0;

    base = GetPCIeBase();
    addr = base + (bus << 20) + (devfn << 12) + offset;
    bwritew(value, addr);
    return 0;
}

UINT pcie_bus_write_config_byte(UCHAR bus, UCHAR devfn, USHORT offset,
UCHAR value)
{
    ULONG    base = 0;
```

```
    ULONG    addr = 0;

    base = GetPCIeBase();
    addr = base + (bus << 20) + (devfn << 12) + offset;
    bwriteb(value, addr);
    return 0;
}
```

访问 PCIe 设备函数的封装 以上函数定义了下列一些访问 PCIe 设备的函数

读 PCIe 设备的配置空间

UINT pcie_bus_read_config_dword(UCHAR bus, UCHAR devfn, USHORT offset, ULONG * pvalue);
UINT pcie_bus_read_config_word(UCHAR bus, UCHAR devfn, USHORT offset, USHORT * pvalue);
UINT pcie_bus_read_config_byte(UCHAR bus, UCHAR devfn, USHORT offset, UCHAR * pvalue);

写 PCIe 设备的配置空间

UINT pcie_bus_write_config_dword(UCHAR bus, UCHAR devfn, USHORT offset, ULONG value);
UINT pcie_bus_write_config_word(UCHAR bus, UCHAR devfn, USHORT offset, USHORT value);
UINT pcie_bus_write_config_byte(UCHAR bus, UCHAR devfn, USHORT offset, UCHAR value);

使用这些函数就可以访问 PCI Express 设备的配置空间了。

应用 3 读写 CMOS

CMOS 是 RTC 的一块存储区域，在 RTC 不断电的情况下 CMOS 存储的值可以被一直保存下去，legacy BIOS 大量的使用 CMOS 来保存 BIOS 设置，所以有时候在客户端大量部署系统的时候，这样大量的系统，如果每个都手动的设置 BIOS，那么需要耗费大量的时间，所以这个时候可以使用一个工具，将配置好系统的 CMOS 设置 dump 出来，然后导入到其他未配置的系统中，可以节省大量的时间。CMOS 的读写使用 IO 端口 0x70~0x73，具体的使用方法请参考南桥的 spec，下面给出两个读写 CMOS 设定的函数。

```
#define QIO_CMOS_INDEX_PORT       0x70
#define QIO_CMOS_DATA_PORT        0x71
#define QIO_CMOS_EXT_INDEX_PORT 0x72
#define QIO_CMOS_EXT_DATA_PORT  0x73

UINT cmos_read(UCHAR index, UCHAR * pvalue)
{
    // Read the first 0x00 ~ 0x7f index value of CMOS
```

```
    if (index < 0x80)
    {
        outb(index, QIO_CMOS_INDEX_PORT);
        *pvalue = inb(QIO_CMOS_DATA_PORT);
    }
    else
    {
        outb(index, QIO_CMOS_EXT_INDEX_PORT);
        *pvalue = inb(QIO_CMOS_EXT_DATA_PORT);
    }

    return 0 ;
}

UINT cmos_write(UCHAR index, UCHAR value)
{
    if (index < 0x80)
    {
        outb(index, QIO_CMOS_INDEX_PORT);
        outb(value, QIO_CMOS_DATA_PORT);
    }
    else
    {
        outb(index, QIO_CMOS_EXT_INDEX_PORT);
        outb(value, QIO_CMOS_EXT_DATA_PORT);
    }

    return 0 ;
}
```

访问 CMOS 函数的封装 以上一些代码定义了下面两个函数

从 CMOS 的某个位置读取一个 BYTE
UINT cmos_read(UCHAR index, UCHAR * pvalue);

将一个 BYTE 写到 CMOS 的某个位置
UINT cmos_write(UCHAR index, UCHAR value);

调用 DOS 或者 BIOS 的 Interrupt

在 DJGPP 中可调用传统的 BIOS 或者 DOS interrupt，此时可定义__dpmi_regs，将参数传递给他，然后调用__dpmi_int 来调用 interrupt，调用完成后，检查__dpmi_regs 中的值来确认调用结果，具体的值取决于调用的 interrupt。下面的代码展示如何调用 int10 来调节亮度。

```
__dpmi_regs  regs;

regs.x.ax = 0x4F14;         /* AH = 4Fh, AL = 14h  */
regs.x.bx = 0x0094;         /* BH = 38h, BL = 00h  */
```

```
__dpmi_int (0x10, &regs);  /* call DOS  */

regs.x.ax = 0x4F14;        /* AH = 4Fh, AL = 14h  */
regs.x.bx = 0x0194;        /* BH = 01h, BL = 94h  */
regs.h.cl = regs.h.cl + 0x20;

__dpmi_int (0x10, &regs);  /* call DOS  */
```

Hook BIOS 或者 DOS 的 interrupt

有时在 Debug 时或者特殊的情况下，需要 Hook 某个 Interrupt，在 DJGPP 中同样可以使用简单的代码实现。下面的 code 演示 Hook Int9 的情况，在 Hook 成功后，当 int 9 触发后，我们的 hook 会先被调用，然后才是原来的 interrupt handler，这样可以在 interrupt 被处理之前做一些处理。

首先定义一个用于 hook 的 interrupt handler `int9_hook_handler`。在 `handler` 中做一些喜欢的事情。

```
int9_hook_handler()
{
  USHORT  key_flag1 = 0;
  USHORT  key_flag2 = 0;
  USHORT  key_buffer_head = 0;
  USHORT  key_buffer_tail = 0;
  UCHAR   key_buffer = 0;
  ULONG   i = 0;
  USHORT  key = 0;

  //printf("Receive int 9\n");
  key_flag1 = readw(0x417);
  printf("Keyboard Shift Qualifier States: %02X\n", key_flag1);
  key_flag2 = readw(0x418);
  printf("Keyboard Toggle Key States : %02X\n", key_flag2);
  key_buffer_head = readw(0x41A);
  printf("Pointer to next character in keyboard buffer: %02X\n",
key_buffer_head);
  key_buffer_tail = readw(0x41C);
  printf("Pointer to last character in keyboard buffer: %02X\n",
key_buffer_tail);

  for (i =0; i < 32; i += 2)
  {
    key = readw(0x41E + i);
    printf("Key[%d] = %04X\n",i/2, key);
  }
}
```

接下来通过下面的调用，将 hander hook 到对应的 interrupt，即 int 9。Hook 成功后，当 int 9 被触发后，我们的 handler 就会被调用。

```
  _go32_dpmi_seginfo old_handler, new_handler;

  printf("Enter int9 hook\n");
  _go32_dpmi_get_protected_mode_interrupt_vector(9, &old_handler);
```

```
new_handler.pm_offset = (int)int9_hook_handler;
new_handler.pm_selector = _go32_my_cs();
_go32_dpmi_chain_protected_mode_interrupt_vector(9, &new_handler);

while(1)
{
}

printf("Releasing int9 hook\n");
_go32_dpmi_set_protected_mode_interrupt_vector(9, &old_handler);
printf("Exit int9 hook\n");
```

结束

OK，到此为止，我们整个学习过程就结束了，你可以使用给出的几组函数，任意达成你的目标，希望学习完这个简单的教程之后，会帮助你提高开发 DOS 程序的效率。 Enjoy the programming!

这些代码是我本人开发中的 lychee project 的一部分，现在将其中 DOS 平台的部分代码以 BSD License 发布，希望对大家的开发有所帮助，代码部分请参考 djapi.pdf。