Migrating a DOS-based application to Linux: A digital I/O example

By Zeke Burgess

This article guides the reader through the process of migrating a DOS-based application to Linux. A simple example program is provided which demonstrates key concepts for direct I/O port access in both DOS and Linux environments. The article concludes with the development of a Linux device driver for a PC/104 card with an onboard 82C55 programmable peripheral interface device.

There are many reasons people move from DOS to Linux for their embedded applications. Among these reasons are robust networking capabilities, efficient multitasking, and freely available source code. With this transition comes added flexibility and complexity. This article is intended to help ease that transition by presenting the process of converting a simple DOS-based program to a Linux-based program utilizing a device driver.

Simple DOS example

Many applications require accessing a digital I/O port. In this example, we look at accessing an 82C55 digital I/O controller with a base address of 0x358 as is available on the Micro/sys MPC148 High Density Digital I/O board (Figure 1). The 82C55 requires four continuous port addresses: the first three are used for 24 bits of I/O, and the fourth for configuration.

The program in Listing 1 was written for 16-bit DOS using Borland C 5.02. It demonstrates configuring the 82C55 to read on Port A and write on Port B. It then reads the value of port A and writes that value to Port B.

```
#include <dos.h>
#define PORTA     0x358
#define PORTB     PORTA+1
#define CFGPORT     PORTA+3
int main(){
  char readvalue;
/* Set up 82C55 to read PORTA
 * and write PORTB      */
  outp(CFGPORT, 0x98);
  while(1){
/* Read from PORTA */
  readvalue = inp(PORTA);
 /* Write to PORTB */
  outp(PORTB, readvalue);
  }
}
```

Listing 1

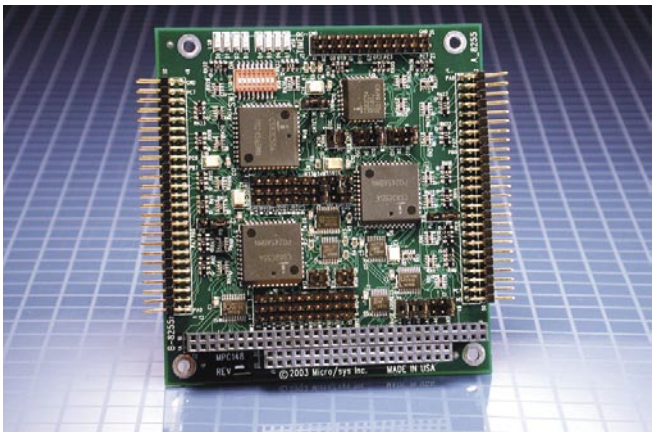

Figure 1

There are two important function calls in this code: outp() and inp(). These calls are responsible for directly accessing the computer's I/O ports. The syntax in Borland C is:

outp(address,data)
int inp(address)

Simple Linux example

The syntax for some functions differs between the Linux compiler and Borland's DOS compiler. For example, with GCC, the standard Linux C compiler, the direct I/O port access calls are defined as:

outb(data,address)
char inb(address)

There are similar functions for word and double word accesses.

Replacing outp() and inp() with outb() and inb() and changing the order of the parameters in outb() are the first steps in migrating this application. If the program is compiled with just these changes, the program will crash with a segmentation fault. In order to access the ports from a user-space program (as opposed to a device driver) the program must have the correct permissions. There are two ways to do this:

ioperm()
iopl()

These functions perform similar tasks but have important differences. The most important is that ioperm() can only enable ranges of ports in the range 0x000 to 0x3FF while iopl() must enable all 65536 ports.

Ioperm() example

The code in Listing 2 makes use of ioperm() to allow access to the four I/O ports starting at 0x358. After allowing for port accesses, the program reads port A and writes that value to port B, as in the DOS example above. Also, the header files that are included have been modified for compiling under Linux. This code needs to be compiled with the -O2 command line switch so that the outb() and inb() functions will be inlined.

```
#include <sys/io.h>
#include <stdio.h>
#define PORTA           0x358
#define PORTB           PORTA+1
#define PORTC           PORTA+2
#define CFGPORT         PORTA+3
int main(){
 char readvalue;
 /* enable ports PORTA-PORTA+4,
  * quit if they are not available */
 if(ioperm(PORTA, 4, 1)) {
  printf("Access denied, quitting\n");
  return(-1);
 }
/* Set up to read A and bottom of C;
 * Write B and top of C */
 outb(0x98, CFGPORT);
 while(1){
  /* Read from Port A */
  readvalue = inb(PORTA);
  /* Write to Port B */
  outb(readvalue, PORTB);
 }
}
```

Listing 2

If this program is run by a non-root user, the "Access denied, quitting" message will be displayed and the program will abort. This is once again an issue with permissions. The iopl() and ioperm() functions require that the program be run with root privileges. For applications in a single-user environment, this may be acceptable. However, when used in a multi-user environment, having an application that runs as root raises security concerns. Another limitation with this method is that the ports that are requested with ioperm() are locked to this particular application.

Introducing the device driver

The device driver addresses these concerns. With a device driver for the 82C55 loaded into the kernel, a user-level application can access the ports and read or write data. Furthermore, with careful programming, multiple applications may access the ports.

The device driver is a set of function calls that the kernel exports to the user's application. The driver is also dependent on the kernel version. For this example, a 2.4 series kernel is used. For the 82C55 device driver, the primary interface will be through the ioctl(), or I/O control, functions. A unique ioctl number that is based on a magic number defined in the header file is given to each ioctl call. There are three things that this driver can do: configure the 82C55, read from a port, and write to a port. In the header file(dio_82C55.h), three ioctl() functions and a magic number are defined:

```
#define DIO_MAGIC   'Z'
#define DIO_CONFIG _IOWR(DIO_MAGIC, 0, sizeof(port));
#define DIO_READ   _IOWR(DIO_MAGIC, 1, sizeof(port));
#define DIO_WRITE  _IOWR(DIO_MAGIC, 2, sizeof(port));
```

The header file also defines a structure dio_port:

```
typedef   struct   dio_port{
   char   portnum;
    char   val;
}
```

This structure is used to pass information between the user application and the device driver. These structures can be as simple as the one above or much more complicated, depending on the hardware that is being accessed and the types and amount of data being transferred between the driver and application.

Linux device driver example

In a device driver there are three things that must be handled. The driver must be initialized, de-initialized, and there should be something that the driver actually does. These necessities are handled by the functions dio_init(), dio_cleanup(), and dio_ioctl().

The function dio_init() is responsible for requesting the ports that will be used by the driver. Here we request ports 0x358 to 0x35C for the driver dio_82C55. This function is called by the kernel module loader when the device is opened.

The function dio_cleanup() releases the resources back to the operating system. This function is called by the kernel module unloader when the device is closed.

The "meat" of the device driver is the function dio_ioctl(). This function determines which ioctl() function was requested and runs the associated code. The Listing 3 code shows the dio_ioctl() function. This code ensures that the argument that is being passed

to it is valid and then determines which ioctl() command to use based on the value of cmd.

```
static int dio_82C55_ioctl(
 struct inode* inode,
 struct file* file,
  unsigned int cmd,
 unsigned long arg)
{
struct dio_port channel;
int rval =0;
/* Make sure that we can access the data
 * from the user application */
 if(!access_ok(VERIFY_WRITE,
    (void *)arg,
   sizeof(port)))
 {
  printk("DIO: cannot read argument\n");
  return -EBUSY;
 }
/* Make a local copy of the data from the
 * user application */
 copy_from_user(&channel,
    (void*)arg,
   sizeof(port));
 switch(cmd){
 case DIO_READ:
  channel.val = inb(channel.portnum
     +PORTA_BASE);
 /* copy the value read back to
    * the user application */
  copy_to_user((void*)arg,
   &channel,
   sizeof(port));
  break;
 case DIO_WRITE:
 /* write the value to the port */
  outb(channel.val,
   channel.portnum+PORTA_BASE);
  break;
 case DIO_CFG:
  /* write configuration settings to
   * the 82C55 configuration register */
    outb(channel.val,
   PORT_CONFIG+PORTA_BASE);
   break;
  default:
   * Return an error since the command
   * is not recognized */
  printk("DIO: Unknown IOCTL %x", cmd);
  return -EINVAL;
  }
return rval;
}
```

Listing 3

Using the Linux driver

While it is possible to write an entire application in the driver, this is not the preferred method. Instead, a user application is written which uses the driver to interface with the hardware. The application code is more complicated than that of the DOS example and the ioperm() example. Instead of just opening ports and writing or reading them, we open a file descriptor and issue ioctl() commands, as in Listing 4 below.

Compiling the Linux Driver

For DOS, the Borland compiler has a slick user interface that allows you to compile by selecting a command from a pull down

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include "dio_82C55.h"
int main(){
int fd;
struct dio_port porta, portb;
struct dio_port portc, portcfg;
porta.portnum = PORTA;
portb.portnum = PORTB;
/* A = In, B = Out */
portcfg.val = PORTA_IN|PORT_ENABLE;
/* open device for read/write */
fd = open("/dev/dio_82C55", O_RDWR);
/* configure the 82C55 */
ioctl(fd, DIO_CFG, &portcfg);
while(1){
 /* Read port A */
 ioctl(fd, DIO_READ, &porta);
 portb.val = porta.val;
 /* Write port B */
 ioctl(fd, DIO_WRITE, &portb);
}
/* Close device */
close(fd);
return 0;
}
```

Listing 4

menu. While there are user interfaces like this in Linux, many people prefer to use the command line.

Compiling the device driver requires compiler flags set to let the compiler know that it is compiling kernel level code. The Makefile in Listing 5 can be used by issuing the make command. The output file, dio_82C55.o, is a kernel module. This module will be inserted into a running kernel and give access to the digital I/O ports.

```
KERNELDIR = /usr/src/linux-2.4.20
include $(KERNELDIR)/.config
CFLAGS = -D__KERNEL__ -DMODULE
CFLAGS += -I$(KERNELDIR)/include -O -Wall
all: dio_82C55.o
clean:
 rm -rf *.o core
```

Listing 5

Compiling the Linux example program

The example program is quite simple with only one file to compile. Using make with a Makefile is one option. However, the example program can easily be compiled from the command line with the command:

gcc -I/usr/src/linux/include dio_example.c -o dio_example

This will produce a binary file called dio_example that may be run from the command line.

Testing the driver and example program

Before the system is ready to run the program, a device in /dev must be created:

mknod /dev/dio_82C55 c 241 0 -m666

This command must be issued while logged in as root. A special file called /dev/dio_82C55 will be created. This file represents a character device with major number 241 and minor number 0. This is in the defined range of "experimental" device numbers, so should not conflict with other drivers that are installed on the system.

We can then insert the device driver into the running system with the insmod command:

insmod dio_driver.o

And run the example program:

./dio_example

If a logic high is applied to the 82C55 port A, the program will read it and output the same value to port B.

SIDEBAR

Who/What Uses DOS

Is DOS Dead?

Though nearly 25 years old, DOS is still being used in many applications. Many systems running with 8086 to Pentium processors use some variant of DOS. A wide variety of embedded applications from data logging systems to control systems use DOS as their operating system.

DOS provides a well-known platform from which to launch applications. It boots quickly, allows direct access to the hardware and memory, and gives the developer complete control of the system, making it an ideal solution for many tasks. However, applications that require a multitasking environment or support for some newer technologies will find other operating systems such as Linux a better match for their needs.

Conclusion

We have now gone from a simple DOS based control program to a simple Linux driver-based control program. Obviously, this is a simplified example. There are times when interrupt handling, DMA transfer, and memory mapped I/O are necessary. Linux provides interfaces to the kernel that handle each of these. Even better, Linux provides source code for drivers that make use of these functions.

***Zeke Burgess** is a software engineer for Micro/sys, Inc., a manufacturer of embedded computer boards. He holds a B.S. in Computer Science and Mathematics from Harvey Mudd College. He has developed Linux device drivers for embedded applications since 2001.*

For further information and copies of the source code used in this article, contact Zeke at:

Micro/sys, Inc.
3730 Park Place
Montrose, CA 91020
Tel: 818-244-4600
Fax: 818-244-4246
E-mail: zburgess@embeddedsys.com
Website: www.embeddedsys.com