Brig Bagley (bbagley) and Tyler Holland (tyhollan)

CPE 453 – 01

Pnico

Project 3 – Magic 8 Ball

24 May 2010

**Project 3: Magic 8 Ball ®**

**Overall architecture of the driver**

**Initialization:** The driver is initialized using the System Event Framework (SEF) which exposes library calls for developers to easily handle initialization. This essentially receives initialization data as input and returns a status code to determine the result of the initialization process. The three main initializations are *fresh, live update, and restart.*

**Starting up:** The driver is started using the *service* command in the shell, then by making a new device. Because the major device was set in dmap.c (see modified files below), the new device can be created using this new major device (18). To shut down the device, the service command is again used.

$service up /usr/sbin/8ball

$mknod /dev/8ball c 18 0

$cat /dev/8ball

$service down 8ball

The main() in the driver source code then calls the driver\_task function that uses our structure of functions that handles the different messages the driver receives.

**Randomizing:** Once a write request takes place, the driver reads the new buffer looking for new lines. It keeps count of all the new lines it receives. When a read takes place, the device returns a random message from a global structure of messages. This is done using the rand() and srand() functions from the time.h header. It reads the system clock to get a random number and then modulates it to get an index for the structure of messages to choose from.

**Answering Questions:** Because it counts the number of questions received (by new lines), the device knows how many times it should answer. When a read takes place, the number of reads asked for will return answers up to the point where it has no more questions to answer. This is kept in a count of questions left to answer.

**OS Integration:** The 8ball driver uses libraries and definition already defined in the kernel. By giving the driver permission and implementing all the functions that a driver needs to communicate to the system, the driver can operate safely without interrupting the rest of the OS environment. When the driver receives any signal, it responds with either OK or whatever the message is asking for. For example, the message could be asking to read. The driver then sends answers to the system. For a write, the driver evaluates the buffer and decides what to do with it. The most important header file that we implemented was ../drivers.h, as it handled a lot of the background driver things so that we could more modularly implement our driver function. The driver struct saved a lot of time.

**Description of the driver implementation**

The Minix Kernel version used is 3.1.6 within the VMware virtual machine environment.

**Files Modified:**

* /usr/src/servers/vfs/dmap.c

The following line was added in the listed device array to recognize the 8ball as major device 18:

DT(0, no\_dev,   0,     NONE,        DMAP\_MUTABLE, "")   /\*18 = /dev/8ball \*/

* /etc/system.conf

This section was added at the end of the file to give the device the proper permissions:

service 8ball

{

system

UMAP # 14

IRQCTL # 19

DEVIO # 21

SETALARM # 24

TIMES # 25

GETINFO # 26

SAFECOPYFROM # 31

SAFECOPYTO # 32

SETGRANT # 34

PROFBUF # 38

SYSCTL

;

ipc

SYSTEM PM RS LOG TTY DS VM VFS

pci inet amddev

;

uid 0;

};

* The remaining files were created for the driver source code displayed below.

**The complete code of your Magic 8 Ball driver**

(Most of the code is derived from the tutorial at http://wiki.minix3.org/en/DevelopersGuide/DriverProgramming)

----------------------------------------------------------------------

MAKEFILE

----------------------------------------------------------------------

#

# Makefile for the 8ball driver.

#

DRIVER = 8ball

#

# Directories.

#

u = /usr

i = $u/include

s = $i/sys

m = $i/minix

b = $i/ibm

d = ..

#

# Build Programs, Flags and Variables.

#

CC = exec cc

CFLAGS = -I$i $(CPROFILE)

LDFLAGS = -i -L../libdriver

LIBS = -ldriver -lsys

OBJ = 8ball.o

# build local binary

all build: $(DRIVER)

$(DRIVER): $(OBJ)

$(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)

install -S 128k $(DRIVER)

# install with other drivers

install: /usr/sbin/$(DRIVER)

/usr/sbin/$(DRIVER): $(DRIVER)

install -o root -cs $? $@

# clean up local files

clean:

rm -f \*.o $(DRIVER)

depend:

mkdep "$(CC) -E $(CPPFLAGS)" \*.c > .depend

# Include generated dependencies.

include .depend

----------------------------------------------------------------------

8ball.h

----------------------------------------------------------------------

/\* 8 Ball Driver header

\* Written by Tyler Holland and Brig Bagley

\* Project 3 for CPE 453

\*/

#ifndef \_\_HELLO\_H

#define \_\_HELLO\_H

/\*\* The Hello, World! message. \*/

#define BALL\_MESSAGE ""

#undef DEBUG

#define DEBUG

#ifdef DEBUG

#define print1(x) printf(x)

#define print2(x,y) printf(x,y)

#define print3(x,y,z) printf(x,y,z)

#else

#define print1(x)

#define print2(x,y)

#define print3(x,y,z)

#endif

/\*Major ID of /dev/8ball. \*/

#define BALL\_MAJOR 18

#endif /\*\_\_HELLO\_H\*/

----------------------------------------------------------------------

8ball.c

----------------------------------------------------------------------

/\* 8 Ball Driver

\* Written by Tyler Holland and Brig Bagley

\* Project 3 for CPE 453

\*/

#include "../drivers.h"

#include "../libdriver/driver.h"

#include <stdio.h>

#include <stdlib.h>

#include <minix/ds.h>

#include <time.h>

#include "8ball.h"

/\*

\* Function prototypes for the ball driver.

\*/

\_PROTOTYPE( PRIVATE char \* ball\_name, (void) );

\_PROTOTYPE( PRIVATE int ball\_open, (struct driver \*d, message \*m) );

\_PROTOTYPE( PRIVATE int ball\_close, (struct driver \*d, message \*m) );

\_PROTOTYPE( PRIVATE struct device \* ball\_prepare, (int device) );

\_PROTOTYPE( PRIVATE int ball\_transfer, (int procnr, int opcode,

u64\_t position, iovec\_t \*iov,

unsigned nr\_req) );

\_PROTOTYPE( PRIVATE void ball\_geometry, (struct partition \*entry) );

/\* Added prototypes: \*/

\_PROTOTYPE( PRIVATE int ball\_ioctl, (struct driver \*dp, message \*m\_ptr) );

\_PROTOTYPE( PRIVATE void ball\_signal, (struct driver \*dp, message \*m\_ptr) );

\_PROTOTYPE( PRIVATE int ball\_cancel, (struct driver \*dp, message \*m\_ptr) );

\_PROTOTYPE( PRIVATE int ball\_hardint, (struct driver \*dp, message \*m\_ptr) );

/\* SEF functions and variables. \*/

PRIVATE void sef\_local\_startup(void);

PRIVATE int sef\_cb\_init\_fresh(int type, sef\_init\_info\_t \*info);

/\* Entry points to the ball driver. \*/

PRIVATE struct driver ball\_tab =

{

ball\_name, /\* Current Device's name \*/

ball\_open, /\* open or mount \*/

ball\_close, /\* close \*/

ball\_ioctl, /\* Just return EINVAL \*/

ball\_prepare, /\* prepare for I/O on a given minor device \*/

ball\_transfer, /\* do the I/O \*/

nop\_cleanup, /\* Cleanup \*/

ball\_geometry, /\* device "geometry" \*/

nop\_signal, /\* SYS\_SIG ? \*/

nop\_alarm, /\* get randomness from kernel (from random's main.c) \*/

ball\_cancel, /\* CANCEL \*/

nop\_select,

nop\_ioctl,

ball\_hardint, /\* HARD\_INT \*/

};

/\*\* Represents the /dev/8ball device. \*/

PRIVATE struct device ball\_device;

/\*\*Message data structure and random generator\*\*/

PRIVATE unsigned int iseed;

PRIVATE int questions = 0;

PRIVATE char \*reply[] =

{

"Signs point to yes.\n",

"Without a doubt.\n",

"You may rely on it.\n",

"It is decidedly so.\n",

"Yes-definitely.\n",

"Most likely\n",

"Outlook good.\n",

"Yes.\n",

"My sources say no.\n",

"Concentrate and ask again.\n",

"Better not tell you now.\n",

"It is certain.\n",

"Ask again later.\n",

"Don't count on it.\n",

"Reply hazy, try again.\n",

"As I see it, yes.\n",

"Outlook not so good.\n",

"Very doubtful.\n",

"Cannot predict now.\n",

"My reply is no.\n"

};

/\* Buffer for the /dev/8ball 8ball driver \*/

#define BALL\_BUF\_SIZE 1024

PRIVATE char ball\_buf[BALL\_BUF\_SIZE];

/\* based on r\_ioctl from main.c in the random driver \*/

/\* We only need to return EINVAL \*/

PRIVATE int ball\_ioctl(dp, m\_ptr)

struct driver \*dp; /\* pointer to driver structure \*/

message \*m\_ptr; /\* pointer to control message \*/

{

struct device \*dv;

if ((dv = ball\_prepare(m\_ptr->DEVICE)) == NIL\_DEV) return(ENXIO);

switch (m\_ptr->REQUEST) {

default:

return(do\_diocntl(&ball\_tab, m\_ptr));

}

return(OK);

/\*return EINVAL;\*/

}

PRIVATE void ball\_signal (struct driver \*dp, message \*m\_ptr)

/\*struct driver \*dp;\*/ /\* pointer to driver structure \*/

/\*message \*m\_ptr;\*/ /\* pointer to control message \*/

{

/\* From printer.c \*/

sigset\_t sigset;

if (getsigset(&sigset) != 0) return;

/\* Expect a SIGTERM signal when this server must shutdown. \*/

if (sigismember(&sigset, SIGTERM)) {

exit(0);

}

/\* Ignore all other signals. \*/

}

PRIVATE int ball\_cancel (dp, m\_ptr)

struct driver \*dp; /\* pointer to driver structure \*/

message \*m\_ptr; /\* pointer to control message \*/

{

return EINTR;

}

PRIVATE int ball\_hardint (dp, m\_ptr)

struct driver \*dp; /\* pointer to driver structure \*/

message \*m\_ptr; /\* pointer to control message \*/

{

return OK; /\* Not sure what to do here \*/

}

PRIVATE char \* ball\_name(void)

{

return "8ball";

}

PRIVATE int ball\_open(d, m)

struct driver \*d;

message \*m;

{

return OK;

}

PRIVATE int ball\_close(d, m)

struct driver \*d;

message \*m;

{

return OK;

}

PRIVATE struct device \* ball\_prepare(dev)

int dev;

{

ball\_device.dv\_base.lo = 0;

ball\_device.dv\_base.hi = 0;

ball\_device.dv\_size.lo = strlen(BALL\_MESSAGE);

ball\_device.dv\_size.hi = 0;

return &ball\_device;

}

PRIVATE int ball\_transfer(proc\_nr, opcode, position, iov, nr\_req)

int proc\_nr;

int opcode;

u64\_t position;

iovec\_t \*iov;

unsigned nr\_req;

{

/\*From hello.c, plus SCATTER logic and question counter\*/

int bytes, ret, i;

char \*fortune;

/\* Set up fortune \*/

/\* This fixes the problem with head printing out weird characters \*/

/\* with fortune = "", bytes is 0, so it fixes the head problem? \*/

if(questions > 0)

{

fortune = reply[rand() % 20];

}

else

{

fortune = "";

}

/\* from hello.c, get smallest length \*/

bytes = strlen(fortune);

if (iov->iov\_size <= 0 || position.lo > strlen(fortune))

{

return OK;

}

/\* Decide if it is a read, write, or EINVAL \*/

switch (opcode)

{

case DEV\_GATHER\_S: /\* Writing out \*/

ret = 0;

if(questions > 0)

{

ret = sys\_safecopyto(proc\_nr, iov->iov\_addr, 0,

(vir\_bytes) fortune,

bytes, D);

questions--;

}

iov->iov\_size -= bytes;

/\*ret = sys\_safecopyto(proc\_nr, iov->iov\_addr, 0,

(vir\_bytes) (reply[0] + position.lo),

bytes, D);\*/

break;

case DEV\_SCATTER\_S: /\* Reading in \*/

ret = sys\_safecopyfrom(proc\_nr, iov->iov\_addr, 0,

(vir\_bytes) ball\_buf, (phys\_bytes) iov->iov\_size, D);

print2("----------\nBUFFER\n---------\n%s---------\n",ball\_buf);

print2("Questions read before:%d\n",questions);

for(i = 0; i < iov->iov\_size; i++) /\*TODO\*/

{

if(ball\_buf[i] == '\n')

{

questions++;

}

}

iov->iov\_size = 0;

print2("Questions read after:%d\n",questions);

break;

default:

return EINVAL; /\* Invalid opcode \*/

}

return ret;

}

PRIVATE void ball\_geometry(entry)

struct partition \*entry;

{

entry->cylinders = 0;

entry->heads = 0;

entry->sectors = 0;

}

PRIVATE void sef\_local\_startup()

{

/\* Register init callbacks. \*/

sef\_setcb\_init\_fresh(sef\_cb\_init\_fresh);

sef\_setcb\_init\_lu(sef\_cb\_init\_fresh); /\* treat live updates as fresh inits \*/

sef\_setcb\_init\_restart(sef\_cb\_init\_fresh); /\* treat restarts as fresh inits \*/

/\* Register live update callbacks. \*/

sef\_setcb\_lu\_prepare(sef\_cb\_lu\_prepare\_always\_ready); /\* agree to update immediately when a LU request is received in a supported state \*/

sef\_setcb\_lu\_state\_isvalid(sef\_cb\_lu\_state\_isvalid\_standard); /\* support live update starting from any standard state \*/

/\* Let SEF perform startup. \*/

sef\_startup();

}

PRIVATE int sef\_cb\_init\_fresh(int type, sef\_init\_info\_t \*info)

{

/\* Initialize the 8ball driver. \*/

u32\_t this\_proc;

switch(type) {

case SEF\_INIT\_FRESH:

print2("%s", BALL\_MESSAGE);

break;

case SEF\_INIT\_LU:

print1("Driver Updated!\n");

break;

case SEF\_INIT\_RESTART:

print1("Driver restarted!\n");

break;

}

/\* Lookup our task number. \*/

if (ds\_retrieve\_label\_num("8ball", &this\_proc) != OK)

{

printf("8ball: ds\_retrieve\_label\_num() failed: %s\n",

strerror(errno));

return EXIT\_FAILURE;

}

/\* Map major number to our process. \*/

if (mapdriver("8ball", BALL\_MAJOR, STYLE\_DEV, TRUE) != OK)

{

printf("8ball: mapdriver() failed: %s\n",

strerror(errno));

return EXIT\_FAILURE;

}

/\* Initialization completed successfully. \*/

return(OK);

}

PUBLIC int main(int argc, char \*\*argv)

{

/\*

\*Perform initialization.

\*/

iseed = (unsigned int)time(NULL);

srand(iseed);

sef\_local\_startup();

/\*

\*Run the main loop.

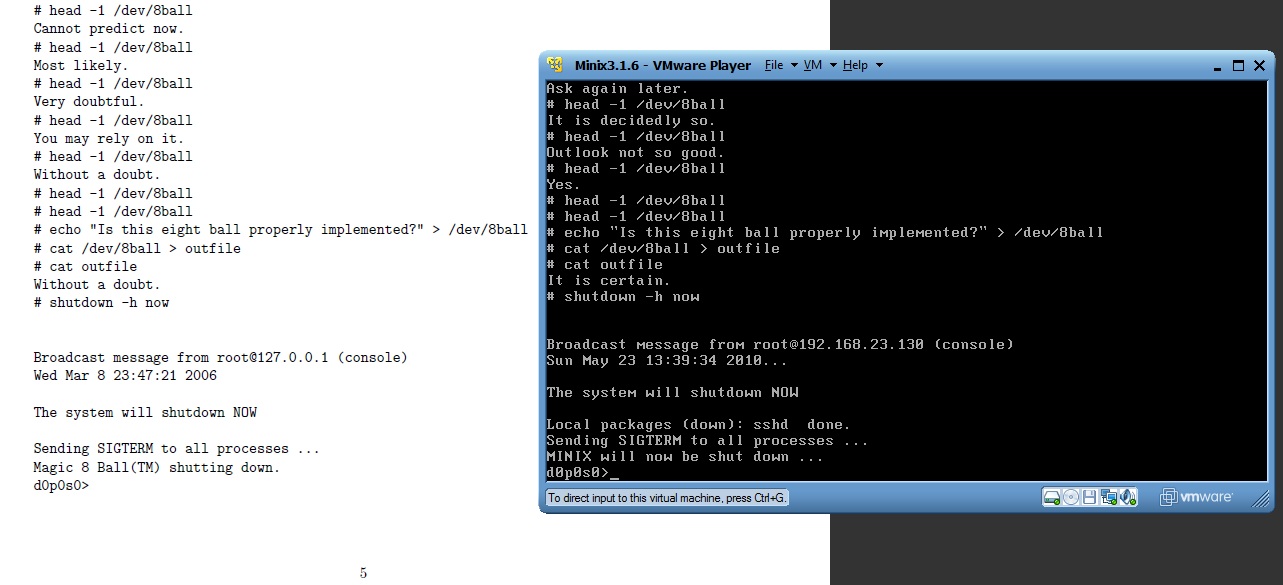
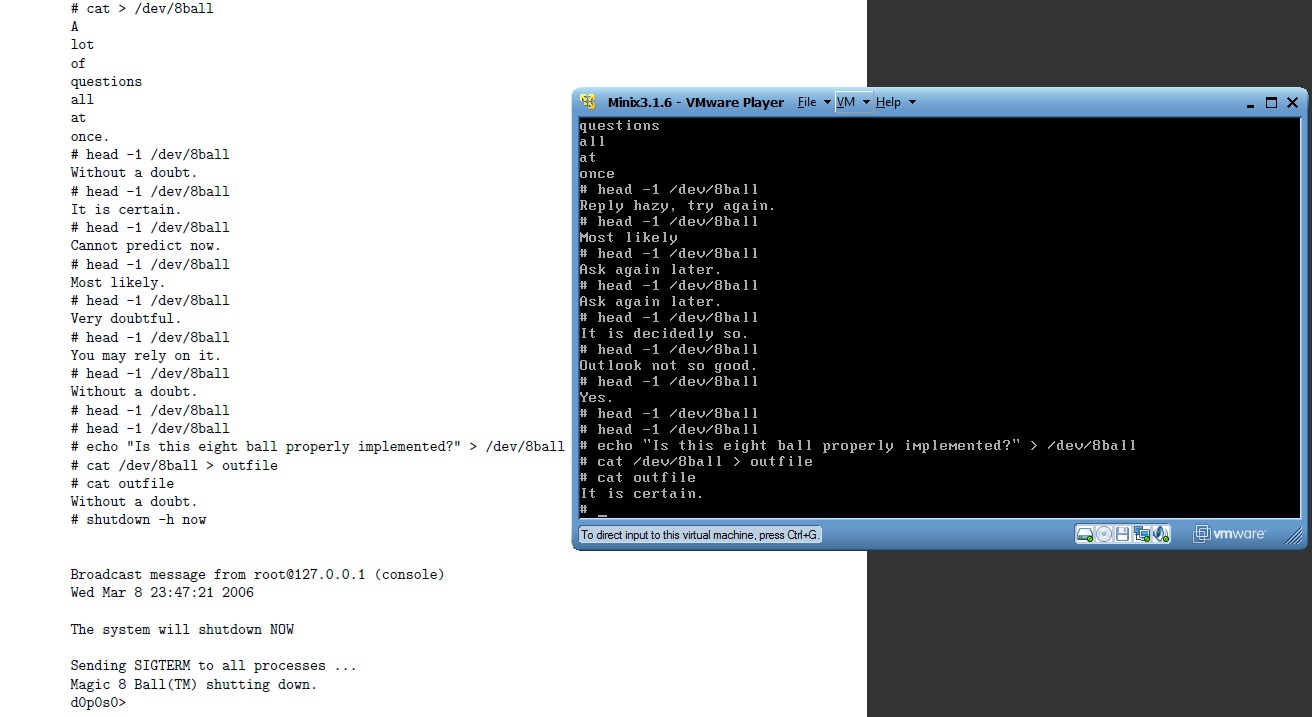
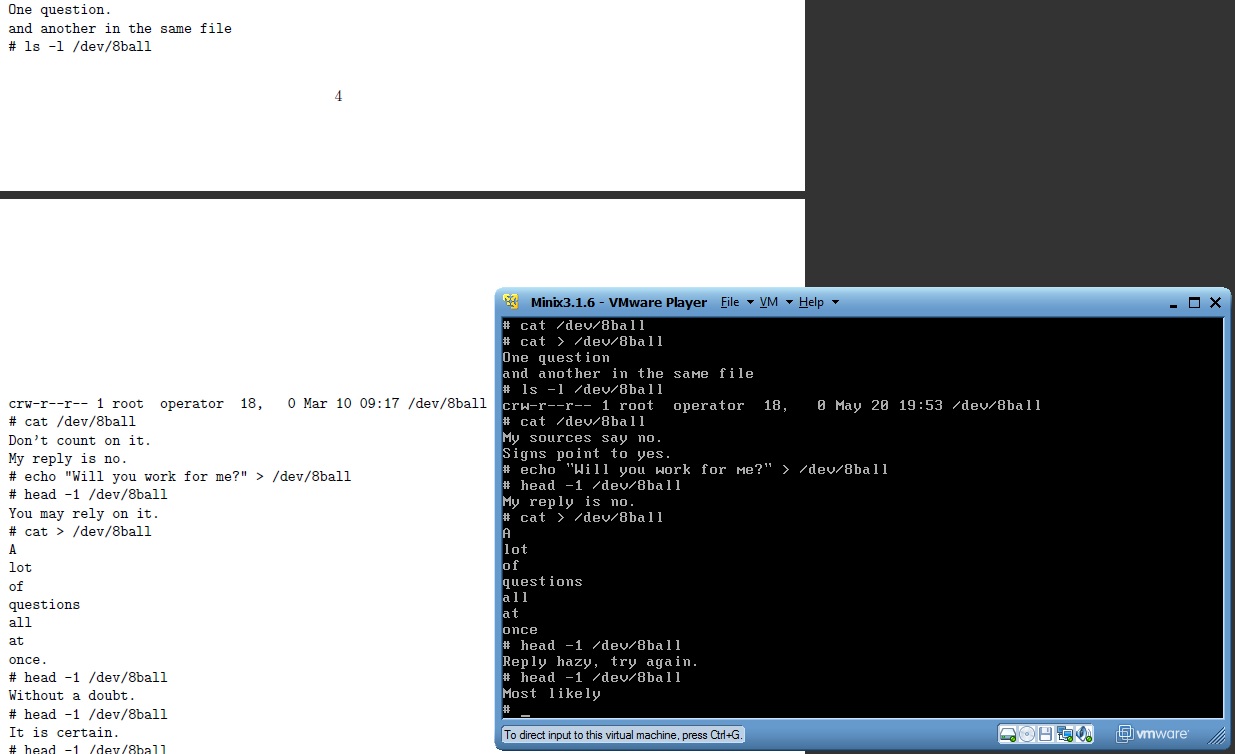
\*/

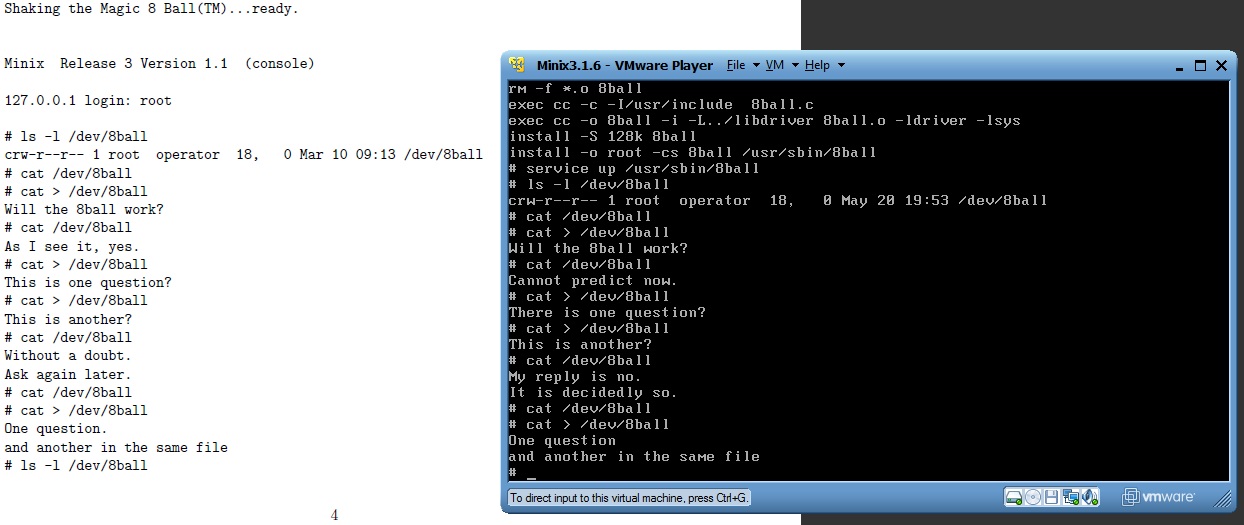
driver\_task(&ball\_tab, DRIVER\_STD);

return OK;

}

**Description of the driver's behavior when running in the system**

Screenshots of driver behavior:



**Problems Encountered**

1. The implementation in the tutorial required Minix 3.1.6 and we had Minix 3.1.1.
2. Compiling had an issue with libdriver.a.
3. We first implemented the output of answers using printf() and had difficulty communicating what we wanted the 8ball to communicate.
4. head -1 /dev/8ball would not work correctly, and either not print or read too far and give weird errors.
5. Sometimes, cat /dev/8ball would give a huge chunk of an error message, ending in me having to Ctrl+c out of it
6. For quite a while, sys\_datacopy would not give any feedback to cat or head -1

**Solutions**

1. We had to install the latest version of Minix from scratch.
2. We found the directory where this library existed and had to compile and make the directory before our driver could compile.
3. We realized that in order to communicate correctly to the tty, we needed to use the function safe\_datacopy() instead of printf().
4. Change the sys\_safecopyto buffer so that it only uses the bytes needed for the message
5. Same as #4, change the buffer size and it worked correctly
6. Re-implement the sys\_safecopyto portion of the ball\_transfer function, looking at the hello.c example driver that we know worked.

**Results and Lessons Learned**

1. Be prepared to upgrade software when modifying the kernel.
2. Be sure to resolve all dependencies before installing a new program or driver.
3. We needed to look over the hello.c example driver more carefully. We ended up going back to the original code from the example in order to get things working again. It is very important to implement things in small steps, it is a lot easier to problem solve that way.
4. Make sure we know what the buffers are doing and how big they need to be. Printf's with the size of the buffers helped a lot here.
5. Same as #4
6. We got it working after looking back to the code we know worked from hello.c example. From there we slowly built it up step by step until we got it working again. Lesson: slowly implement functionality, it is much easier to debug and fix that way

**Other Information:**

Our Magic 8Ball driver has the code necessary, with help from drivers.h, to handle all of the signals stated in the assignment. HARD\_INT is handled with ball\_hardint, DEV\_READ and DEV\_WRITE are taken in by the OS in minix 3.1.6 and turned into DEV\_SCATTER and DEV\_GATHER opcodes, which our ball\_transfer function deals with. DEV\_IOCTL is handled with ball\_ioctl, DEV\_OPEN with ball\_open, DEV\_CLOSE with ball\_close, CANCEL with ball\_cancel, and SYS\_SIG with ball\_signal. Once I found the driver.h and driver.c files, it was much easier to see how these signals/opcodes were handled by the OS.

Overall, the most important thing we learned is that using existing code to help your code work makes everything easier to use and more understandable. Also, well documented source code is extremely easy to work with when compared to undocumented code.