An Exercise in Formal Verification

Verifying a Linux Device Driver with SMACK

By Matthew Elbert

# Introduction

Formal verification of software can provide strong reassurance that a program will work as the developer intended, without entering forbidden states or causing dangerous side effects. This is especially important when the software can affect the safety or health of individuals. A common example is software that controls the traffic lights at an intersection: The lights must never enter a state where both directions receive a green signal at the same time. A formal verification of the correctly written program can, for example, prove that the software cannot enter that state.

Other systems also benefit greatly from formal verification, like heavily used kernel programs in the Linux operating system. A bug or failure in the operating system could potentially cause much greater damage than one in a standalone user application. We should recognize sufficient motivation for formally verifying these types of programs, if not all programs.

So why not formally verify everything? It would certainly be wonderful, but there are very real limitations to the tools that have currently been developed for such verification purposes. As it turns out, some programs are far too large and far too complex for our tools to tackle them. In practice, formal verification of large projects requires great effort and specialized focus. The verification process I went through for a simple Linux device driver shows some of these complexities, and they will be described below.

# Goals

My goals throughout the project shifted as I made, or in some cases did not make, progress. I wanted to explore SMACK as a static analysis tool in the verification of a Linux driver. I originally chose to try the USB boot protocol keyboard driver, or usbkbd. I had read a paper [1] that showed this driver’s successful verification with a tool called Verifast[2]. I wanted to install SMACK and learn how to use it to verify a simple “Hello World” program first, and then use it to verify a Linux driver. I had never done any software verification before so I had no idea if I would be able to completely verify anything, but I wanted to learn more about the verification process and some of the capabilities and limitations of the tools. After trying unsuccessfully with the keyboard driver for a while, and with help from others, I decided to switch to verifying the DS1286 real time clock interface driver, and was able to have a little more success.

# Problems Encountered

I had many problems right from the beginning. My first goal was to get access to SMACK and use it to verify a simple problem. What I didn’t understand is that SMACK is built on a number of other tools like Boogie and Z3 and in order to use it you need to install a host of tools and files. I was using my user account in the Linux CADE lab on campus to try and run this tool. I was prevented from installing the full package because it required administrator privileges, which I didn’t have, and even if I could get them the size of the full package would take most of my allotted space. I then tried to install a precompiled binary that was provided from the SMACK developers. It seemed to work in part, but I was receiving strange errors even trying to verify my simple “hello world” program. It was determined that SMACK was behaving differently on the CADE machines than it was for the developers.

I abandoned my attempts to use the CADE machines and started setting up my own environment on my Windows laptop where I could download, compile and run the full program in the same environment as the developers. I achieved this by using Git to clone the development branch of SMACK from GitHub [3], Virtual Box [4] to set up a virtual Linux machine, and Vagrant [5] to set up my Linux virtual environment to be identical to the development environment. After a few hours of frustrating and cryptic error messages, and finding out that I needed to enable hardware virtualization in my Windows BIOS, I was finally set up.

After getting SMACK set up, another problem I had was my lack of understanding of how verification tools worked. I was under the assumption that you could just invoke SMACK with any old source code file and it would parse through it and do its thing. I quickly learned that the tool needs to be intimately woven in with the source code (in my opinion, a major limitation of the tool, yet understandably necessary). I couldn’t simply take my driver source usbkbd.c and pass it to SMACK. I needed to include SMACK files in the source, be able to compile it independently from the rest of the Linux kernel, and provide a ‘main’ function to exercise the driver, otherwise SMACK wouldn’t know what to verify. I spent a lot of time trying to ‘strip’ down the driver file, cut the chain of ‘#include’ dependencies, and link the driver with the proper SMACK files just so that I could independently compile it. In essence, the ‘real’ code you intend to verify must be stripped from its home and woven in an artificial ‘harness’ designed specifically for verification. For a verification newbie, as well as someone with little familiarity to this particular driver, it is not an easy task. I began to understand why not all programs are verified, because just getting some small chunk of code into a verifiable state can be difficult and time consuming.

After consultation with my professor, to expedite my journey to actual verification of some code, I decided to use some code that had already been prepared for verification. I installed DDVerify [6] which had several Linux device drivers prepared with altered headers specifically for verification. I chose to verify the DS1286 driver, which I still needed to tie into SMACK before finally starting verification.

I began to finally have success when I looked more into the ‘simple-project’ that SMACK had in the examples directory. It provided a make file to compile and link the source code with SMACK tools. I was able to move the prepared header and source files for the DS1286 driver into the example directory and start verifying from there.

# Results

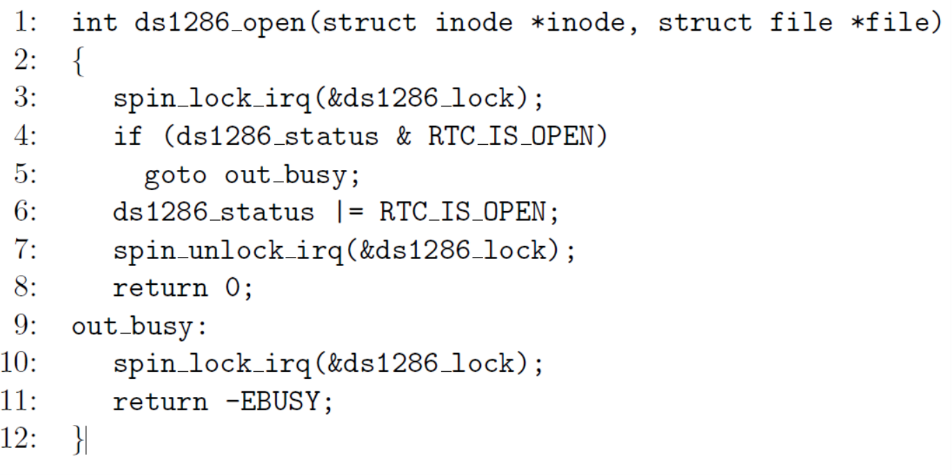
I learned a few tricks to verification in the process of verifying the DS1286 driver. One major finding is that verification is only as powerful as the way you exercise it. For example I created a ‘main’ function in the driver source file that did nothing, and SMACK reported to me that there were no bugs in my program. Even when I put something clearly false like “assert(false);” in several of the driver functions, SMACK was cool with it. It helped me realize that SMACK wasn’t even looking at it. For one, you have to call a function from your main in order for SMACK to even look at it. You also need to include ‘assert’ statements, because if there are none, the verifier doesn’t know what to check.

As a result, the user needs to be very careful in crafting their test to make sure they exercise all of the code, and verify every important aspect of the program. Now that would be a difficult task for even the person who wrote the original driver, someone who thoroughly understands the driver’s function and limitations, and likely an even more difficult task for someone unfamiliar with the driver. I figured I would at least catch most of the code by instantiating objects and calling every function from the driver code in my main block at least once. I even experimented with multiple calls to functions like ds1286\_init() and ds1286\_exit() to see if I could trigger some bad event that may not normally happen.

I realized that SMACK didn’t care about the order of function calls to the driver when it had no problem with me calling ds1286\_exit() followed by something like ds1286\_ioctl(…). I was suspicious of this because that seemed to violate the intended use of the driver. I realized in order to check these kinds of properties I needed to introduce additional variables (more artificial ‘harness’). I added two global variables ‘driver\_exited’ and ‘driver\_init’, and initialized them at the top of the main block. I set init🡪true and exited🡪false in ds1286\_init() and vice-versa for ds1286\_exit(); I then added assert statements in every other function of the driver checking that the driver had been initialized and not exited. SMACK was then able to return errors when I called the functions out of order. This technique would also work to check other flow-type properties, but it would require specific additions to the code. This reiterates the tool’s weakness, it is not powerful enough to help unless the user clearly defines the problem for it.

Because the DS1286 allows the use of the real-time clock built into the computer from user space for multiple threads or processes, it uses spinlock mechanisms to protect critical sections of code. One important property I wanted to check was that no locking conditions were violated. I modified some of the assert statements in the file ‘spinlock.c’ to throw false errors, and was surprised when SMACK returned once again saying everything was happily verified. Clearly SMACK wasn’t even looking there. My professor helped me realize that the spinlock code had to be specifically included in the makefile and modified to include SMACK headers before SMACK found any errors. This is once again a reminder that if the user isn’t fully aware of what the verifier is doing, they could get a false sense of security that the code is bug-free, when in reality the test harness is insufficient bugs may still exist.

Once I was certain the spinlock properties were being checked and were properly asserted, I was surprised to find that SMACK found a locking violation in the function ds1286\_open(…) which I show below in Figure 1 for illustration purposes.



**Figure 1**

The bug in ds1286\_open(…) shown in Figure 1 was revealed on line 10, where the lock variable ds1286\_lock was reacquired after being previously locked in line 3. Line 10 should be changed to spin\_**un**lock\_irq(&ds1286\_lock); As it turned out, this was a legitimate bug in the driver source code which had already been reported to the Linux Kernel team, and a patched driver had been submitted [7]. I hadn’t realized the version of the driver I was working with was outdated until this point, but had some cause for celebration being assured that SMACK was doing a good job verifying the driver. With the correction made, SMACK was able to verify that no other locking violations occurred in the driver.

# Conclusions

While formally verifying programs is a very important step in software development, it is often not accomplished because of its difficulty. The cost of verification can be easily justified, however, when the program is used in critical systems where bugs and failures cannot be tolerated. In my (admittedly) very simple exercise of formally verifying a small device driver for the Linux operating system, I was surprised how much extra effort it took just to get the small piece of code in a state where the verifier SMACK could reason about it. I was also surprised at how much the verifier could miss if I (the user) hadn’t laid out specific assertions and ensured that the code harness had been properly crafted. It is very difficult to properly exercise the code in verification unless you have an intimate knowledge of the code and its purposes and limits.

One useful trick I learned in the verification process is the importance of inserting obvious checks at various points in the program, like an ‘assert(false)’, to gage whether then verifier is responding expectedly. It was only after using this trick and getting unexpected results that I was able to identify that SMACK wasn’t analyzing certain parts of the code. I was then able to fix the problem and be more confident that SMACK was bringing in all the code I was expecting it to. In the end, after I corrected a legitimate bug in the original driver, SMACK reported that the driver was free of bugs. Our confidence in this report depends on our confidence in the method used to verify it. So I won’t say I can guarantee the driver is free of bugs, but I feel sufficient properties were checked to provide a high degree of confidence in its validity.

Perhaps the greatest lesson I learned from this exercise is the importance for me to write my code in such a way that it lends itself to verification right from the beginning. Verification languages like Microsoft’s Dafny[8] provide the developer with a framework to annotate their code for verification right from the start. It is likely that nobody understands the software’s specifications better than those involved in the development process, so verification is arguably best accomplished by these individuals. When properly used, these verification tools provide an added insight to the developer, and can easily catch bugs and errors that may otherwise be overlooked.

I would still need a lot of practice using SMACK before I could say I really understood it, but I think this simple experiment has helped me understand quite a bit about the process of verification. I have a better understanding of the limitations of formal verification, and realize that the majority of the burden is placed on the user’s shoulders. We would be well to remember the old adage: “A tool is only as good as the person who uses it”. Another cliché phrase, “The best defense is a good offense”, reminds us that it is better to write code with verification in mind from the start than for someone who doesn’t understand it as well to try to verify it later. These lessons will alter the way I approach software development in the future.

# References

1. Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Sound formal verification of linux's USB BP keyboard driver. In Proceedings of the 4th international conference on NASA Formal Methods (NFM'12), Alwyn E. Goodloe and Suzette Person (Eds.). Springer-Verlag, Berlin, Heidelberg, 210-215. DOI=10.1007/978-3-642-28891-3\_21 http://dx.doi.org/10.1007/978-3-642-28891-3\_21
2. Bart Jacobs, Jan Smans and Frank Piessens, VeriFast verification tool, http://people.cs.kuleuven.be/~bart.jacobs/verifast/
3. SMACK verification tool, https://github.com/smackers/smack
4. Virtual Box utility, https://www.virtualbox.org/
5. Vagrant portable development environment utility, https://www.vagrantup.com/
6. DDVerify verification tool, http://www.cprover.org/ddverify/
7. Thomas Bogendoerfer, Patched DS1286 driver, http://www.linux-mips.org/archives/linux-mips/2008-08/msg00017.html
8. Microsoft Research, Dafny, http://research.microsoft.com/en-us/projects/dafny/

Appendix

Files used in verification:

makefile:

# Set this variable to point to folder share of your SMACK installation

INSTALL\_SHARE = ../../share

CC = clang -I/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/

#="-I../../../models/seq1/include/"

LD = llvm-link

INC = $(INSTALL\_SHARE)/smack/include

CFLAGS = -DMEMORY\_MODEL\_NO\_REUSE\_IMPLS -c -Wall -emit-llvm -O0 -g -I$(INC)

SOURCES = incr.c simple.c spinlock.c

OBJS = $(subst .c,.bc,$(SOURCES)) smack.bc

all: $(OBJS)

$(LD) -o simple-project.bc $(OBJS)

# BITPRECISE has to be set during compilation to enable bit-precise mode

bitprecise: CFLAGS = -DBITPRECISE -c -Wall -emit-llvm -O0 -g -I$(INC)

bitprecise: $(OBJS)

$(LD) -o simple-project.bc $(OBJS)

simple.bc: simple.c simple.h

incr.bc: incr.c incr.h

spinlock.bc: spinlock.c

smack.bc: $(INSTALL\_SHARE)/smack/lib/smack.c $(INC)/smack.h

$(CC) $(CFLAGS) $< -o smack.bc

%.bc: %.c

$(CC) $(CFLAGS) $< -o $@

clean:

rm -f \*.bc \*.bpl

simple.c:

//

// This file is distributed under the MIT License. See LICENSE for details.

//

#include "simple.h"

int driver\_exited;

int driver\_init;

void test(int a) {

int b = a;

a = incr(a);

assert(a > b);

}

//////////////////////////////////////////

/\*

\* DS1286 Real Time Clock interface for Linux

\*

\* Copyright (C) 1998, 1999, 2000 Ralf Baechle

\*

\* Based on code written by Paul Gortmaker.

\*

\* This driver allows use of the real time clock (built into nearly all

\* computers) from user space. It exports the /dev/rtc interface supporting

\* various ioctl() and also the /proc/rtc pseudo-file for status

\* information.

\*

\* The ioctls can be used to set the interrupt behaviour and generation rate

\* from the RTC via IRQ 8. Then the /dev/rtc interface can be used to make

\* use of these timer interrupts, be they interval or alarm based.

\*

\* The /dev/rtc interface will block on reads until an interrupt has been

\* received. If a RTC interrupt has already happened, it will output an

\* unsigned long and then block. The output value contains the interrupt

\* status in the low byte and the number of interrupts since the last read

\* in the remaining high bytes. The /dev/rtc interface can also be used with

\* the select(2) call.

\*

\* This program is free software; you can redistribute it and/or modify it

\* under the terms of the GNU General Public License as published by the

\* Free Software Foundation; either version 2 of the License, or (at your

\* option) any later version.

\*/

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/types.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/errno.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/miscdevice.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/slab.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/ioport.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/fcntl.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/init.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/poll.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/rtc.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/spinlock.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/bcd.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/linux/proc\_fs.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/asm/uaccess.h"

#include "/home/vagrant/smack/ddverify-2010-04-30/models/seq1/include/asm/system.h"

#define DS1286\_VERSION "1.0"

/\*

\* We sponge a minor off of the misc major. No need slurping

\* up another valuable major dev number for this. If you add

\* an ioctl, make sure you don't conflict with SPARC's RTC

\* ioctls.

\*/

DECLARE\_WAIT\_QUEUE\_HEAD(ds1286\_wait);

ssize\_t ds1286\_read(struct file \*file, char \*buf,

size\_t count, loff\_t \*ppos);

int ds1286\_ioctl(struct inode \*inode, struct file \*file,

unsigned int cmd, unsigned long arg);

unsigned int ds1286\_poll(struct file \*file, poll\_table \*wait);

void ds1286\_get\_alm\_time (struct rtc\_time \*alm\_tm);

void ds1286\_get\_time(struct rtc\_time \*rtc\_tm);

int ds1286\_set\_time(struct rtc\_time \*rtc\_tm);

inline unsigned char ds1286\_is\_updating(void);

DEFINE\_SPINLOCK(ds1286\_lock);

int ds1286\_read\_proc(char \*page, char \*\*start, off\_t off,

int count, int \*eof, void \*data);

/\*

\* Bits in rtc\_status. (7 bits of room for future expansion)

\*/

#define RTC\_IS\_OPEN 0x01 /\* means /dev/rtc is in use \*/

#define RTC\_TIMER\_ON 0x02 /\* missed irq timer active \*/

unsigned char ds1286\_status; /\* bitmapped status byte. \*/

unsigned char days\_in\_mo[] = {

0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31

};

/\*

\* Now all the various file operations that we export.

\*/

ssize\_t ds1286\_read(struct file \*file, char \*buf,

size\_t count, loff\_t \*ppos)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

return -EIO;

}

int ds1286\_ioctl(struct inode \*inode, struct file \*file,

unsigned int cmd, unsigned long arg)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

struct rtc\_time wtime;

switch (cmd) {

case RTC\_AIE\_OFF: /\* Mask alarm int. enab. bit \*/

{

unsigned long flags;

unsigned char val;

if (!capable(CAP\_SYS\_TIME))

return -EACCES;

spin\_lock\_irqsave(&ds1286\_lock, flags);

val = rtc\_read(RTC\_CMD);

val |= RTC\_TDM;

rtc\_write(val, RTC\_CMD);

spin\_unlock\_irqrestore(&ds1286\_lock, flags);

return 0;

}

case RTC\_AIE\_ON: /\* Allow alarm interrupts. \*/

{

unsigned long flags;

unsigned char val;

if (!capable(CAP\_SYS\_TIME))

return -EACCES;

spin\_lock\_irqsave(&ds1286\_lock, flags);

val = rtc\_read(RTC\_CMD);

val &= ~RTC\_TDM;

rtc\_write(val, RTC\_CMD);

spin\_unlock\_irqrestore(&ds1286\_lock, flags);

return 0;

}

case RTC\_WIE\_OFF: /\* Mask watchdog int. enab. bit \*/

{

unsigned long flags;

unsigned char val;

if (!capable(CAP\_SYS\_TIME))

return -EACCES;

spin\_lock\_irqsave(&ds1286\_lock, flags);

val = rtc\_read(RTC\_CMD);

val |= RTC\_WAM;

rtc\_write(val, RTC\_CMD);

spin\_unlock\_irqrestore(&ds1286\_lock, flags);

return 0;

}

case RTC\_WIE\_ON: /\* Allow watchdog interrupts. \*/

{

unsigned long flags;

unsigned char val;

if (!capable(CAP\_SYS\_TIME))

return -EACCES;

spin\_lock\_irqsave(&ds1286\_lock, flags);

val = rtc\_read(RTC\_CMD);

val &= ~RTC\_WAM;

rtc\_write(val, RTC\_CMD);

spin\_unlock\_irqrestore(&ds1286\_lock, flags);

return 0;

}

case RTC\_ALM\_READ: /\* Read the present alarm time \*/

{

/\*

\* This returns a struct rtc\_time. Reading >= 0xc0

\* means "don't care" or "match all". Only the tm\_hour,

\* tm\_min, and tm\_sec values are filled in.

\*/

memset(&wtime, 0, sizeof(wtime));

ds1286\_get\_alm\_time(&wtime);

break;

}

case RTC\_ALM\_SET: /\* Store a time into the alarm \*/

{

/\*

\* This expects a struct rtc\_time. Writing 0xff means

\* "don't care" or "match all". Only the tm\_hour,

\* tm\_min and tm\_sec are used.

\*/

unsigned char hrs, min, sec;

struct rtc\_time alm\_tm;

if (!capable(CAP\_SYS\_TIME))

return -EACCES;

if (copy\_from\_user(&alm\_tm, (struct rtc\_time\*)arg,

sizeof(struct rtc\_time)))

return -EFAULT;

hrs = alm\_tm.tm\_hour;

min = alm\_tm.tm\_min;

if (hrs >= 24)

hrs = 0xff;

if (min >= 60)

min = 0xff;

BIN\_TO\_BCD(sec);

BIN\_TO\_BCD(min);

BIN\_TO\_BCD(hrs);

spin\_lock(&ds1286\_lock);

rtc\_write(hrs, RTC\_HOURS\_ALARM);

rtc\_write(min, RTC\_MINUTES\_ALARM);

spin\_unlock(&ds1286\_lock);

return 0;

}

case RTC\_RD\_TIME: /\* Read the time/date from RTC \*/

{

memset(&wtime, 0, sizeof(wtime));

ds1286\_get\_time(&wtime);

break;

}

case RTC\_SET\_TIME: /\* Set the RTC \*/

{

struct rtc\_time rtc\_tm;

if (!capable(CAP\_SYS\_TIME))

return -EACCES;

if (copy\_from\_user(&rtc\_tm, (struct rtc\_time\*)arg,

sizeof(struct rtc\_time)))

return -EFAULT;

return ds1286\_set\_time(&rtc\_tm);

}

default:

return -EINVAL;

}

return copy\_to\_user((void \*)arg, &wtime, sizeof wtime) ? -EFAULT : 0;

}

/\*

\* We enforce only one user at a time here with the open/close.

\* Also clear the previous interrupt data on an open, and clean

\* up things on a close.

\*/

int ds1286\_open(struct inode \*inode, struct file \*file)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

spin\_lock\_irq(&ds1286\_lock);

if (ds1286\_status & RTC\_IS\_OPEN)

goto out\_busy;

ds1286\_status |= RTC\_IS\_OPEN;

spin\_unlock\_irq(&ds1286\_lock);

return 0;

out\_busy:

spin\_unlock\_irq(&ds1286\_lock); //MKE changed lock to unlock

return -EBUSY;

}

int ds1286\_release(struct inode \*inode, struct file \*file)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

ds1286\_status &= ~RTC\_IS\_OPEN;

return 0;

}

unsigned int ds1286\_poll(struct file \*file, poll\_table \*wait)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

poll\_wait(file, &ds1286\_wait, wait);

return 0;

}

/\*

\* The various file operations we support.

\*/

const struct file\_operations ds1286\_fops = {

.llseek = no\_llseek,

.read = ds1286\_read,

.poll = ds1286\_poll,

.ioctl = ds1286\_ioctl,

.open = ds1286\_open,

.release = ds1286\_release,

};

struct miscdevice ds1286\_dev=

{

.minor = RTC\_MINOR,

.name = "rtc",

.fops = &ds1286\_fops,

};

int \_\_init ds1286\_init(void)

{

driver\_init = 1;

driver\_exited = 0;

int err;

printk(KERN\_INFO "DS1286 Real Time Clock Driver v%s\n", DS1286\_VERSION);

err = misc\_register(&ds1286\_dev);

if (err)

goto out;

if (!create\_proc\_read\_entry("driver/rtc", 0, 0, ds1286\_read\_proc, NULL)) {

err = -ENOMEM;

goto out\_deregister;

}

return 0;

out\_deregister:

misc\_deregister(&ds1286\_dev);

out:

return err;

}

void \_\_exit ds1286\_exit(void)

{

assert(driver\_init); //MKE check

remove\_proc\_entry("driver/rtc", NULL);

misc\_deregister(&ds1286\_dev);

driver\_exited = 1;

driver\_init = 0; //MKE require re-init

}

char \*days[] = {

"\*\*\*", "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"

};

/\*

\* Info exported via "/proc/rtc".

\*/

int ds1286\_proc\_output(char \*buf)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

char \*p, \*s;

struct rtc\_time tm;

unsigned char hundredth, month, cmd, amode;

p = buf;

ds1286\_get\_time(&tm);

hundredth = rtc\_read(RTC\_HUNDREDTH\_SECOND);

BCD\_TO\_BIN(hundredth);

p += sprintf(p,

"rtc\_time\t: %02d:%02d:%02d.%02d\n"

"rtc\_date\t: %04d-%02d-%02d\n",

tm.tm\_hour, tm.tm\_min, tm.tm\_sec, hundredth,

tm.tm\_year + 1900, tm.tm\_mon + 1, tm.tm\_mday);

/\*

\* We implicitly assume 24hr mode here. Alarm values >= 0xc0 will

\* match any value for that particular field. Values that are

\* greater than a valid time, but less than 0xc0 shouldn't appear.

\*/

ds1286\_get\_alm\_time(&tm);

p += sprintf(p, "alarm\t\t: %s ", days[tm.tm\_wday]);

if (tm.tm\_hour <= 24)

p += sprintf(p, "%02d:", tm.tm\_hour);

else

p += sprintf(p, "\*\*:");

if (tm.tm\_min <= 59)

p += sprintf(p, "%02d\n", tm.tm\_min);

else

p += sprintf(p, "\*\*\n");

month = rtc\_read(RTC\_MONTH);

p += sprintf(p,

"oscillator\t: %s\n"

"square\_wave\t: %s\n",

(month & RTC\_EOSC) ? "disabled" : "enabled",

(month & RTC\_ESQW) ? "disabled" : "enabled");

amode = ((rtc\_read(RTC\_MINUTES\_ALARM) & 0x80) >> 5) |

((rtc\_read(RTC\_HOURS\_ALARM) & 0x80) >> 6) |

((rtc\_read(RTC\_DAY\_ALARM) & 0x80) >> 7);

if (amode == 7) s = "each minute";

else if (amode == 3) s = "minutes match";

else if (amode == 1) s = "hours and minutes match";

else if (amode == 0) s = "days, hours and minutes match";

else s = "invalid";

p += sprintf(p, "alarm\_mode\t: %s\n", s);

cmd = rtc\_read(RTC\_CMD);

p += sprintf(p,

"alarm\_enable\t: %s\n"

"wdog\_alarm\t: %s\n"

"alarm\_mask\t: %s\n"

"wdog\_alarm\_mask\t: %s\n"

"interrupt\_mode\t: %s\n"

"INTB\_mode\t: %s\_active\n"

"interrupt\_pins\t: %s\n",

(cmd & RTC\_TDF) ? "yes" : "no",

(cmd & RTC\_WAF) ? "yes" : "no",

(cmd & RTC\_TDM) ? "disabled" : "enabled",

(cmd & RTC\_WAM) ? "disabled" : "enabled",

(cmd & RTC\_PU\_LVL) ? "pulse" : "level",

(cmd & RTC\_IBH\_LO) ? "low" : "high",

(cmd & RTC\_IPSW) ? "unswapped" : "swapped");

return p - buf;

}

int ds1286\_read\_proc(char \*page, char \*\*start, off\_t off,

int count, int \*eof, void \*data)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

int len = ds1286\_proc\_output (page);

if (len <= off+count) \*eof = 1;

\*start = page + off;

len -= off;

if (len>count)

len = count;

if (len<0)

len = 0;

return len;

}

/\*

\* Returns true if a clock update is in progress

\*/

inline unsigned char ds1286\_is\_updating(void)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

return rtc\_read(RTC\_CMD) & RTC\_TE;

}

void ds1286\_get\_time(struct rtc\_time \*rtc\_tm)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

unsigned char save\_control;

unsigned long flags;

unsigned long uip\_watchdog = jiffies;

/\*

\* read RTC once any update in progress is done. The update

\* can take just over 2ms. We wait 10 to 20ms. There is no need to

\* to poll-wait (up to 1s - eeccch) for the falling edge of RTC\_UIP.

\* If you need to know \*exactly\* when a second has started, enable

\* periodic update complete interrupts, (via ioctl) and then

\* immediately read /dev/rtc which will block until you get the IRQ.

\* Once the read clears, read the RTC time (again via ioctl). Easy.

\*/

if (ds1286\_is\_updating() != 0)

while (jiffies - uip\_watchdog < 2\*HZ/100)

barrier();

/\*

\* Only the values that we read from the RTC are set. We leave

\* tm\_wday, tm\_yday and tm\_isdst untouched. Even though the

\* RTC has RTC\_DAY\_OF\_WEEK, we ignore it, as it is only updated

\* by the RTC when initially set to a non-zero value.

\*/

spin\_lock\_irqsave(&ds1286\_lock, flags);

save\_control = rtc\_read(RTC\_CMD);

rtc\_write((save\_control|RTC\_TE), RTC\_CMD);

rtc\_tm->tm\_sec = rtc\_read(RTC\_SECONDS);

rtc\_tm->tm\_min = rtc\_read(RTC\_MINUTES);

rtc\_tm->tm\_hour = rtc\_read(RTC\_HOURS) & 0x3f;

rtc\_tm->tm\_mday = rtc\_read(RTC\_DATE);

rtc\_tm->tm\_mon = rtc\_read(RTC\_MONTH) & 0x1f;

rtc\_tm->tm\_year = rtc\_read(RTC\_YEAR);

rtc\_write(save\_control, RTC\_CMD);

spin\_unlock\_irqrestore(&ds1286\_lock, flags);

BCD\_TO\_BIN(rtc\_tm->tm\_sec);

BCD\_TO\_BIN(rtc\_tm->tm\_min);

BCD\_TO\_BIN(rtc\_tm->tm\_hour);

BCD\_TO\_BIN(rtc\_tm->tm\_mday);

BCD\_TO\_BIN(rtc\_tm->tm\_mon);

BCD\_TO\_BIN(rtc\_tm->tm\_year);

/\*

\* Account for differences between how the RTC uses the values

\* and how they are defined in a struct rtc\_time;

\*/

if (rtc\_tm->tm\_year < 45)

rtc\_tm->tm\_year += 30;

if ((rtc\_tm->tm\_year += 40) < 70)

rtc\_tm->tm\_year += 100;

rtc\_tm->tm\_mon--;

}

int ds1286\_set\_time(struct rtc\_time \*rtc\_tm)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

unsigned char mon, day, hrs, min, sec, leap\_yr;

unsigned char save\_control;

unsigned int yrs;

unsigned long flags;

yrs = rtc\_tm->tm\_year + 1900;

mon = rtc\_tm->tm\_mon + 1; /\* tm\_mon starts at zero \*/

day = rtc\_tm->tm\_mday;

hrs = rtc\_tm->tm\_hour;

min = rtc\_tm->tm\_min;

sec = rtc\_tm->tm\_sec;

if (yrs < 1970)

return -EINVAL;

leap\_yr = ((!(yrs % 4) && (yrs % 100)) || !(yrs % 400));

if ((mon > 12) || (day == 0))

return -EINVAL;

if (day > (days\_in\_mo[mon] + ((mon == 2) && leap\_yr)))

return -EINVAL;

if ((hrs >= 24) || (min >= 60) || (sec >= 60))

return -EINVAL;

if ((yrs -= 1940) > 255) /\* They are unsigned \*/

return -EINVAL;

if (yrs >= 100)

yrs -= 100;

BIN\_TO\_BCD(sec);

BIN\_TO\_BCD(min);

BIN\_TO\_BCD(hrs);

BIN\_TO\_BCD(day);

BIN\_TO\_BCD(mon);

BIN\_TO\_BCD(yrs);

spin\_lock\_irqsave(&ds1286\_lock, flags);

save\_control = rtc\_read(RTC\_CMD);

rtc\_write((save\_control|RTC\_TE), RTC\_CMD);

rtc\_write(yrs, RTC\_YEAR);

rtc\_write(mon, RTC\_MONTH);

rtc\_write(day, RTC\_DATE);

rtc\_write(hrs, RTC\_HOURS);

rtc\_write(min, RTC\_MINUTES);

rtc\_write(sec, RTC\_SECONDS);

rtc\_write(0, RTC\_HUNDREDTH\_SECOND);

rtc\_write(save\_control, RTC\_CMD);

spin\_unlock\_irqrestore(&ds1286\_lock, flags);

return 0;

}

void ds1286\_get\_alm\_time(struct rtc\_time \*alm\_tm)

{

assert(driver\_init); //MKE check

assert(!driver\_exited); //MKE check

unsigned char cmd;

unsigned long flags;

/\*

\* Only the values that we read from the RTC are set. That

\* means only tm\_wday, tm\_hour, tm\_min.

\*/

spin\_lock\_irqsave(&ds1286\_lock, flags);

alm\_tm->tm\_min = rtc\_read(RTC\_MINUTES\_ALARM) & 0x7f;

alm\_tm->tm\_hour = rtc\_read(RTC\_HOURS\_ALARM) & 0x1f;

alm\_tm->tm\_wday = rtc\_read(RTC\_DAY\_ALARM) & 0x07;

cmd = rtc\_read(RTC\_CMD);

spin\_unlock\_irqrestore(&ds1286\_lock, flags);

BCD\_TO\_BIN(alm\_tm->tm\_min);

BCD\_TO\_BIN(alm\_tm->tm\_hour);

alm\_tm->tm\_sec = 0;

}

module\_init(ds1286\_init);

module\_exit(ds1286\_exit);

MODULE\_AUTHOR("Ralf Baechle");

MODULE\_LICENSE("GPL");

MODULE\_ALIAS\_MISCDEV(RTC\_MINOR);

int main(void) {

int a = \_\_VERIFIER\_nondet();

driver\_exited = 0;

driver\_init = 0;

a = ds1286\_init();

char \*buf = malloc(1000);

a = ds1286\_proc\_output(buf);

struct inode \*inode = malloc(sizeof(struct inode));

struct file \*file = malloc(sizeof(struct file));

a = ds1286\_open(inode, file);

size\_t count;

loff\_t ppos;

ssize\_t retVal = ds1286\_read(file, buf, count, ppos);

unsigned int cmd;

unsigned long arg;

struct rtc\_time \*rtc\_tm = malloc(sizeof(struct rtc\_time));

a = ds1286\_ioctl(inode, file, cmd, arg);

ds1286\_get\_time(rtc\_tm);

a = ds1286\_release(inode, file);

poll\_table \*wait = malloc(sizeof(poll\_table));

cmd = ds1286\_poll(file, wait);

a = ds1286\_set\_time(rtc\_tm);

char \*page = malloc(1000);

char \*\*start = malloc(1000);

off\_t off;

int count1 = \_\_VERIFIER\_nondet();

int \*eof = \_\_VERIFIER\_nondet();

void \*data;

int x = ds1286\_read\_proc(page, start, off, count1, eof, data);

char up = ds1286\_is\_updating();

ds1286\_exit();

return 0;

}

simple.h:

//

// This file is distributed under the MIT License. See LICENSE for details.

//

#ifndef SIMPLE\_H

#define SIMPLE\_H

#include "smack.h"

#include "incr.h"

void test(int);

//#endif // SIMPLE\_H

///////////////////////////////////////////

/\*

\* Copyright (C) 1998, 1999, 2003 Ralf Baechle

\*

\* This file is subject to the terms and conditions of the GNU General Public

\* License. See the file "COPYING" in the main directory of this archive

\* for more details.

\*/

//#ifndef \_\_LINUX\_DS1286\_H

//#define \_\_LINUX\_DS1286\_H

//#include "asm\_ds1286.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* register summary

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define RTC\_HUNDREDTH\_SECOND 0

#define RTC\_SECONDS 1

#define RTC\_MINUTES 2

#define RTC\_MINUTES\_ALARM 3

#define RTC\_HOURS 4

#define RTC\_HOURS\_ALARM 5

#define RTC\_DAY 6

#define RTC\_DAY\_ALARM 7

#define RTC\_DATE 8

#define RTC\_MONTH 9

#define RTC\_YEAR 10

#define RTC\_CMD 11

#define RTC\_WHSEC 12

#define RTC\_WSEC 13

#define RTC\_UNUSED 14

/\* RTC\_\*\_alarm is always true if 2 MSBs are set \*/

# define RTC\_ALARM\_DONT\_CARE 0xC0

/\*

\* Bits in the month register

\*/

#define RTC\_EOSC 0x80

#define RTC\_ESQW 0x40

/\*

\* Bits in the Command register

\*/

#define RTC\_TDF 0x01

#define RTC\_WAF 0x02

#define RTC\_TDM 0x04

#define RTC\_WAM 0x08

#define RTC\_PU\_LVL 0x10

#define RTC\_IBH\_LO 0x20

#define RTC\_IPSW 0x40

#define RTC\_TE 0x80

#endif /\* \_\_LINUX\_DS1286\_H \*/

spinlock.c:

#include <linux/spinlock.h>

#include "smack.h"

inline void spin\_lock\_init(spinlock\_t \* lock)

{

lock->init = 1;

lock->locked = 0;

}

inline void spin\_lock(spinlock\_t \* lock)

{

\_\_VERIFIER\_atomic\_begin();

\_\_VERIFIER\_assert(lock->init);

\_\_VERIFIER\_assert(!lock->locked);

lock->locked = 1;

\_\_VERIFIER\_atomic\_end();

}

inline void spin\_lock\_irqsave(spinlock\_t \*lock, unsigned long flags)

{

\_\_VERIFIER\_atomic\_begin();

\_\_VERIFIER\_assert(lock->init);

\_\_VERIFIER\_assert(!lock->locked);

lock->locked = 1;

\_\_VERIFIER\_atomic\_end();

}

inline void spin\_lock\_irq(spinlock\_t \*lock)

{

\_\_VERIFIER\_atomic\_begin();

\_\_VERIFIER\_assert(lock->init);

\_\_VERIFIER\_assert(!lock->locked);

lock->locked = 1;

\_\_VERIFIER\_atomic\_end();

}

inline void spin\_lock\_bh(spinlock\_t \*lock)

{

\_\_VERIFIER\_atomic\_begin();

\_\_VERIFIER\_assert(lock->init);

\_\_VERIFIER\_assert(!lock->locked);

lock->locked = 1;

\_\_VERIFIER\_atomic\_end();

}

inline void spin\_unlock(spinlock\_t \*lock)

{

\_\_VERIFIER\_assert(lock->locked);

lock->locked = 0;

}

inline void spin\_unlock\_irqrestore(spinlock\_t \*lock, unsigned long flags)

{

\_\_VERIFIER\_assert(lock->locked);

lock->locked = 0;

}

inline void spin\_unlock\_irq(spinlock\_t \*lock)

{

\_\_VERIFIER\_assert(lock->locked);

lock->locked = 0;

}

inline void spin\_unlock\_bh(spinlock\_t \*lock)

{

\_\_VERIFIER\_assert(lock->locked);

lock->locked = 0;

}