平台：

    硬件：at91sam9261，ds3231。软件：linux-3.6.9。

驱动部分：

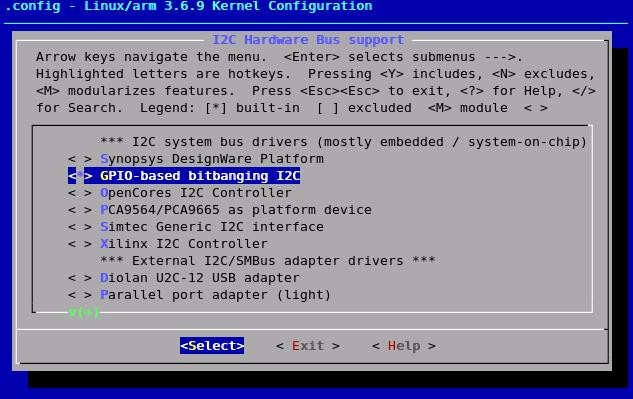
    使用rtc-ds1307.c在其基础上，进行修改。改动见最底部代码部分。

    内核需要选中I2C和RTC部分驱动。

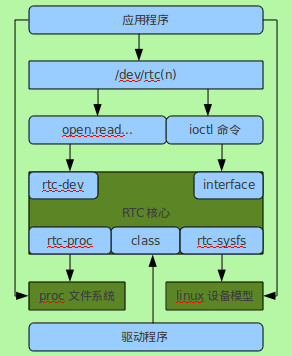
Real Time Clock选中I2C RTC drivers中DS1307。

 I2C support--->需要选中I2C Hardware Bus support中GPIO-based bitbanging I2C。否则，即使rtc驱动正确，也不能使用，提示：

drivers/rtc/hctosys.c: unable to open rtc device (rtc0)。问题源自，驱动配置正确，I2C驱动没有加载正确，udevd不能创建rtc0设备节点。



rtc驱动和设备使用关系树：



 如果提示class下面的问题，那基本都是设备驱动没有初始化完善，导致不能正确创建设备节点，读写操作就更是不可能了。

在该嵌入式linux中使用DS3231，需要在DS1307这个驱动上增加修改，同时在board-sam9261.c中加入如下修改：

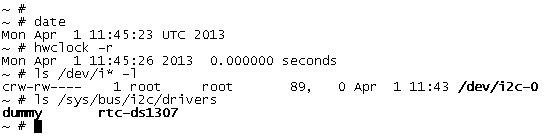
I2C\_BOARD\_INFO中标明，使用rtc-ds1307的驱动文件，并采用其子项中ds3231的芯片，i2c设备地址为0x68。

static struct i2c\_board\_info \_\_initdata ek\_i2c\_devices[] = {  
 {  
  I2C\_BOARD\_INFO("rtc-ds1307", 0x68),  
  .type = "ds3231",  
 }  
};

static void \_\_init ek\_board\_init(void)  
{  
 /\* Serial \*/  
 /\* DBGU on ttyS0. (Rx & Tx only) \*/  
 at91\_register\_uart(0, 0, 0);  
 at91\_register\_uart(AT91SAM9261\_ID\_US0, 1, 0);  
 at91\_register\_uart(AT91SAM9261\_ID\_US1, 2, ATMEL\_UART\_CTS | ATMEL\_UART\_RTS);  
 at91\_add\_device\_serial();  
 /\* USB Host \*/  
 at91\_add\_device\_usbh(&ek\_usbh\_data);  
 /\* USB Device \*/  
 at91\_add\_device\_udc(&ek\_udc\_data);  
 /\* I2C \*/  
 at91\_add\_device\_i2c(ek\_i2c\_devices, ARRAY\_SIZE(ek\_i2c\_devices));  
 /\* NAND \*/  
 ek\_add\_device\_nand();  
 /\* DM9000 ethernet \*/  
 ek\_add\_device\_dm9000();

 /\* spi0 and mmc/sd share the same PIO pins \*/  
 /\* SPI \*/  
 at91\_add\_device\_spi(ek\_spi\_devices, ARRAY\_SIZE(ek\_spi\_devices));  
 at91\_add\_device\_ssc(AT91SAM9261\_ID\_SSC1, ATMEL\_SSC\_TX);  
 /\* Push Buttons \*/  
 ek\_add\_device\_buttons();  
 /\* LEDs \*/  
 at91\_gpio\_leds(ek\_leds, ARRAY\_SIZE(ek\_leds));  
}

驱动正确配置后，I2C和RTC正常，硬件时钟芯片读写正确如下所示：



 /dev/i2c-0 提示ic2初始化正确；

/sys/bus/i2c/drivers/rtc-ds1307 提示驱动已加载。

//////////////////////////rtc-ds1307.c部分加入ds3231支持驱动代码////////////////////////////

#include <linux/module.h>  
#include <linux/init.h>  
#include <linux/slab.h>  
#include <linux/i2c.h>  
#include <linux/string.h>  
#include <linux/rtc.h>  
#include <linux/bcd.h>  
#include <linux/rtc/ds1307.h>

/\*  
 \* We can't determine type by probing, but if we expect pre-Linux code  
 \* to have set the chip up as a clock (turning on the oscillator and  
 \* setting the date and time), Linux can ignore the non-clock features.  
 \* That's a natural job for a factory or repair bench.  
 \*/  
enum ds\_type {  
 ds\_1307,  
 ds\_1337,  
 ds\_1338,  
 ds\_1339,  
 ds\_1340,  
 ds\_1388,  
 ds\_3231,  
 m41t00,  
 mcp7941x,  
 rx\_8025,  
 last\_ds\_type /\* always last \*/  
 /\* rs5c372 too?  different address... \*/  
};

/\* RTC registers don't differ much, except for the century flag \*/  
#define DS1307\_REG\_SECS  0x00 /\* 00-59 \*/  
# define DS1307\_BIT\_CH  0x80  
# define DS1340\_BIT\_nEOSC  0x80  
# define MCP7941X\_BIT\_ST  0x80  
#define DS1307\_REG\_MIN  0x01 /\* 00-59 \*/  
#define DS1307\_REG\_HOUR  0x02 /\* 00-23, or 1-12{am,pm} \*/  
# define DS1307\_BIT\_12HR  0x40 /\* in REG\_HOUR \*/  
# define DS1307\_BIT\_PM  0x20 /\* in REG\_HOUR \*/  
# define DS1340\_BIT\_CENTURY\_EN 0x80 /\* in REG\_HOUR \*/  
# define DS1340\_BIT\_CENTURY 0x40 /\* in REG\_HOUR \*/  
#define DS1307\_REG\_WDAY  0x03 /\* 01-07 \*/  
# define MCP7941X\_BIT\_VBATEN 0x08  
#define DS1307\_REG\_MDAY  0x04 /\* 01-31 \*/  
#define DS1307\_REG\_MONTH 0x05 /\* 01-12 \*/  
# define DS1337\_BIT\_CENTURY 0x80 /\* in REG\_MONTH \*/  
#define DS1307\_REG\_YEAR  0x06 /\* 00-99 \*/

/\*  
 \* Other registers (control, status, alarms, trickle charge, NVRAM, etc)  
 \* start at 7, and they differ a LOT. Only control and status matter for  
 \* basic RTC date and time functionality; be careful using them.  
 \*/  
#define DS1307\_REG\_CONTROL 0x07  /\* or ds1338 \*/  
# define DS1307\_BIT\_OUT  0x80  
# define DS1338\_BIT\_OSF  0x20  
# define DS1307\_BIT\_SQWE  0x10  
# define DS1307\_BIT\_RS1  0x02  
# define DS1307\_BIT\_RS0  0x01  
#define DS1337\_REG\_CONTROL 0x0e  
# define DS1337\_BIT\_nEOSC  0x80  
# define DS1339\_BIT\_BBSQI  0x20  
# define DS3231\_BIT\_BBSQW  0x40 /\* same as BBSQI \*/  
# define DS1337\_BIT\_RS2  0x10  
# define DS1337\_BIT\_RS1  0x08  
# define DS1337\_BIT\_INTCN  0x04  
# define DS1337\_BIT\_A2IE  0x02  
# define DS1337\_BIT\_A1IE  0x01  
#define DS1340\_REG\_CONTROL 0x07  
# define DS1340\_BIT\_OUT  0x80  
# define DS1340\_BIT\_FT  0x40  
# define DS1340\_BIT\_CALIB\_SIGN 0x20  
# define DS1340\_M\_CALIBRATION 0x1f  
#define DS1340\_REG\_FLAG  0x09  
# define DS1340\_BIT\_OSF  0x80  
#define DS1337\_REG\_STATUS 0x0f  
# define DS1337\_BIT\_OSF  0x80  
# define DS1337\_BIT\_A2I  0x02  
# define DS1337\_BIT\_A1I  0x01  
#define DS1339\_REG\_ALARM1\_SECS 0x07

#define DS13XX\_TRICKLE\_CHARGER\_MAGIC 0xa0

#define RX8025\_REG\_CTRL1 0x0e  
# define RX8025\_BIT\_2412  0x20  
#define RX8025\_REG\_CTRL2 0x0f  
# define RX8025\_BIT\_PON  0x10  
# define RX8025\_BIT\_VDET  0x40  
# define RX8025\_BIT\_XST  0x20

struct ds1307 {  
 u8   offset; /\* register's offset \*/  
 u8   regs[11];  
 u16   nvram\_offset;  
 struct bin\_attribute \*nvram;  
 enum ds\_type  type;  
 unsigned long  flags;  
#define HAS\_NVRAM 0  /\* bit 0 == sysfs file active \*/  
#define HAS\_ALARM 1  /\* bit 1 == irq claimed \*/  
 struct i2c\_client \*client;  
 struct rtc\_device \*rtc;  
 struct work\_struct work;  
 s32 (\*read\_block\_data)(const struct i2c\_client \*client, u8 command,  
          u8 length, u8 \*values);  
 s32 (\*write\_block\_data)(const struct i2c\_client \*client, u8 command,  
    u8 length, const u8 \*values);  
};

struct chip\_desc {  
 unsigned  alarm:1;  
 u16   nvram\_offset;  
 u16   nvram\_size;  
 u16   trickle\_charger\_reg;  
};

static const struct chip\_desc chips[last\_ds\_type] = {  
 [ds\_1307] = {  
  .nvram\_offset = 8,  
  .nvram\_size = 56,  
 },  
 [ds\_1337] = {  
  .alarm  = 1,  
 },  
 [ds\_1338] = {  
  .nvram\_offset = 8,  
  .nvram\_size = 56,  
 },  
 [ds\_1339] = {  
  .alarm  = 1,  
  .trickle\_charger\_reg = 0x10,  
 },  
 [ds\_1340] = {  
  .trickle\_charger\_reg = 0x08,  
 },  
 [ds\_1388] = {  
  .trickle\_charger\_reg = 0x0a,  
 },  
 [ds\_3231] = {  
  .alarm  = 1,  
 },  
 [mcp7941x] = {  
  /\* this is battery backed SRAM \*/  
  .nvram\_offset = 0x20,  
  .nvram\_size = 0x40,  
 },  
};

static const struct i2c\_device\_id ds1307\_id[] = {  
 { "ds1307", ds\_1307 },  
 { "ds1337", ds\_1337 },  
 { "ds1338", ds\_1338 },  
 { "ds1339", ds\_1339 },  
 { "ds1388", ds\_1388 },  
 { "ds1340", ds\_1340 },  
 { "ds3231", ds\_3231 },  
 { "m41t00", m41t00 },  
 { "mcp7941x", mcp7941x },  
 { "pt7c4338", ds\_1307 },  
 { "rx8025", rx\_8025 },  
 { }  
};  
MODULE\_DEVICE\_TABLE(i2c, ds1307\_id);

/\*----------------------------------------------------------------------\*/

#define BLOCK\_DATA\_MAX\_TRIES 10

static s32 ds1307\_read\_block\_data\_once(const struct i2c\_client \*client,  
           u8 command, u8 length, u8 \*values)  
{  
 s32 i, data;

 for (i = 0; i < length; i++) {  
  data = i2c\_smbus\_read\_byte\_data(client, command + i);  
  if (data < 0)  
   return data;  
  values[i] = data;  
 }  
 return i;  
}

static s32 ds1307\_read\_block\_data(const struct i2c\_client \*client, u8 command,  
      u8 length, u8 \*values)  
{  
 u8 oldvalues[I2C\_SMBUS\_BLOCK\_MAX];  
 s32 ret;  
 int tries = 0;

 dev\_dbg(&client->dev, "ds1307\_read\_block\_data (length=%d)\n", length);  
 ret = ds1307\_read\_block\_data\_once(client, command, length, values);  
 if (ret < 0)  
  return ret;  
 do {  
  if (++tries > BLOCK\_DATA\_MAX\_TRIES) {  
   dev\_err(&client->dev,  
    "ds1307\_read\_block\_data failed\n");  
   return -EIO;  
  }  
  memcpy(oldvalues, values, length);  
  ret = ds1307\_read\_block\_data\_once(client, command, length,  
        values);  
  if (ret < 0)  
   return ret;  
 } while (memcmp(oldvalues, values, length));  
 return length;  
}

static s32 ds1307\_write\_block\_data(const struct i2c\_client \*client, u8 command,  
       u8 length, const u8 \*values)  
{  
 u8 currvalues[I2C\_SMBUS\_BLOCK\_MAX];  
 int tries = 0;

 dev\_dbg(&client->dev, "ds1307\_write\_block\_data (length=%d)\n", length);  
 do {  
  s32 i, ret;

  if (++tries > BLOCK\_DATA\_MAX\_TRIES) {  
   dev\_err(&client->dev,  
    "ds1307\_write\_block\_data failed\n");  
   return -EIO;  
  }  
  for (i = 0; i < length; i++) {  
   ret = i2c\_smbus\_write\_byte\_data(client, command + i,  
       values[i]);  
   if (ret < 0)  
    return ret;  
  }  
  ret = ds1307\_read\_block\_data\_once(client, command, length,  
        currvalues);  
  if (ret < 0)  
   return ret;  
 } while (memcmp(currvalues, values, length));  
 return length;  
}

/\*----------------------------------------------------------------------\*/

/\*  
 \* The IRQ logic includes a "real" handler running in IRQ context just  
 \* long enough to schedule this workqueue entry.   We need a task context  
 \* to talk to the RTC, since I2C I/O calls require that; and disable the  
 \* IRQ until we clear its status on the chip, so that this handler can  
 \* work with any type of triggering (not just falling edge).  
 \*  
 \* The ds1337 and ds1339 both have two alarms, but we only use the first  
 \* one (with a "seconds" field).  For ds1337 we expect nINTA is our alarm  
 \* signal; ds1339 chips have only one alarm signal.  
 \*/  
static void ds1307\_work(struct work\_struct \*work)  
{  
 struct ds1307  \*ds1307;  
 struct i2c\_client \*client;  
 struct mutex  \*lock;  
 int   stat, control;

 ds1307 = container\_of(work, struct ds1307, work);  
 client = ds1307->client;  
 lock = &ds1307->rtc->ops\_lock;

 mutex\_lock(lock);  
 stat = i2c\_smbus\_read\_byte\_data(client, DS1337\_REG\_STATUS);  
 if (stat < 0)  
  goto out;

 if (stat & DS1337\_BIT\_A1I) {  
  stat &= ~DS1337\_BIT\_A1I;  
  i2c\_smbus\_write\_byte\_data(client, DS1337\_REG\_STATUS, stat);

  control = i2c\_smbus\_read\_byte\_data(client, DS1337\_REG\_CONTROL);  
  if (control < 0)  
   goto out;

  control &= ~DS1337\_BIT\_A1IE;  
  i2c\_smbus\_write\_byte\_data(client, DS1337\_REG\_CONTROL, control);

  rtc\_update\_irq(ds1307->rtc, 1, RTC\_AF | RTC\_IRQF);  
 }

out:  
 if (test\_bit(HAS\_ALARM, &ds1307->flags))  
  enable\_irq(client->irq);  
 mutex\_unlock(lock);  
}

static irqreturn\_t ds1307\_irq(int irq, void \*dev\_id)  
{  
 struct i2c\_client \*client = dev\_id;  
 struct ds1307  \*ds1307 = i2c\_get\_clientdata(client);

 disable\_irq\_nosync(irq);  
 schedule\_work(&ds1307->work);  
 return IRQ\_HANDLED;  
}

/\*----------------------------------------------------------------------\*/

static int ds1307\_get\_time(struct device \*dev, struct rtc\_time \*t)  
{  
 struct ds1307 \*ds1307 = dev\_get\_drvdata(dev);  
 int  tmp;

 /\* read the RTC date and time registers all at once \*/  
 tmp = ds1307->read\_block\_data(ds1307->client,  
  ds1307->offset, 7, ds1307->regs);  
 if (tmp != 7) {  
  dev\_err(dev, "%s error %d\n", "read", tmp);  
  return -EIO;  
 }

 dev\_dbg(dev, "%s: %02x %02x %02x %02x %02x %02x %02x\n",  
   "read",  
   ds1307->regs[0], ds1307->regs[1],  
   ds1307->regs[2], ds1307->regs[3],  
   ds1307->regs[4], ds1307->regs[5],  
   ds1307->regs[6]);

 t->tm\_sec = bcd2bin(ds1307->regs[DS1307\_REG\_SECS] & 0x7f);  
 t->tm\_min = bcd2bin(ds1307->regs[DS1307\_REG\_MIN] & 0x7f);  
 tmp = ds1307->regs[DS1307\_REG\_HOUR] & 0x3f;  
 t->tm\_hour = bcd2bin(tmp);  
 t->tm\_wday = bcd2bin(ds1307->regs[DS1307\_REG\_WDAY] & 0x07) - 1;  
 t->tm\_mday = bcd2bin(ds1307->regs[DS1307\_REG\_MDAY] & 0x3f);  
 tmp = ds1307->regs[DS1307\_REG\_MONTH] & 0x1f;  
 t->tm\_mon = bcd2bin(tmp) - 1;

 /\* assume 20YY not 19YY, and ignore DS1337\_BIT\_CENTURY \*/  
 t->tm\_year = bcd2bin(ds1307->regs[DS1307\_REG\_YEAR]) + 100;

 dev\_dbg(dev, "%s secs=%d, mins=%d, "  
  "hours=%d, mday=%d, mon=%d, year=%d, wday=%d\n",  
  "read", t->tm\_sec, t->tm\_min,  
  t->tm\_hour, t->tm\_mday,  
  t->tm\_mon, t->tm\_year, t->tm\_wday);

 /\* initial clock setting can be undefined \*/  
 return rtc\_valid\_tm(t);  
}

static int ds1307\_set\_time(struct device \*dev, struct rtc\_time \*t)  
{  
 struct ds1307 \*ds1307 = dev\_get\_drvdata(dev);  
 int  result;  
 int  tmp;  
 u8  \*buf = ds1307->regs;

 dev\_dbg(dev, "%s secs=%d, mins=%d, "  
  "hours=%d, mday=%d, mon=%d, year=%d, wday=%d\n",  
  "write", t->tm\_sec, t->tm\_min,  
  t->tm\_hour, t->tm\_mday,  
  t->tm\_mon, t->tm\_year, t->tm\_wday);

 buf[DS1307\_REG\_SECS] = bin2bcd(t->tm\_sec);  
 buf[DS1307\_REG\_MIN] = bin2bcd(t->tm\_min);  
 buf[DS1307\_REG\_HOUR] = bin2bcd(t->tm\_hour);  
 buf[DS1307\_REG\_WDAY] = bin2bcd(t->tm\_wday + 1);  
 buf[DS1307\_REG\_MDAY] = bin2bcd(t->tm\_mday);  
 buf[DS1307\_REG\_MONTH] = bin2bcd(t->tm\_mon + 1);

 /\* assume 20YY not 19YY \*/  
 tmp = t->tm\_year - 100;  
 buf[DS1307\_REG\_YEAR] = bin2bcd(tmp);

 switch (ds1307->type) {  
 case ds\_1337:  
 case ds\_1339:  
 case ds\_3231:  
  buf[DS1307\_REG\_MONTH] |= DS1337\_BIT\_CENTURY;  
  break;  
 case ds\_1340:  
  buf[DS1307\_REG\_HOUR] |= DS1340\_BIT\_CENTURY\_EN  
    | DS1340\_BIT\_CENTURY;  
  break;  
 case mcp7941x:  
  /\*  
   \* these bits were cleared when preparing the date/time  
   \* values and need to be set again before writing the  
   \* buffer out to the device.  
   \*/  
  buf[DS1307\_REG\_SECS] |= MCP7941X\_BIT\_ST;  
  buf[DS1307\_REG\_WDAY] |= MCP7941X\_BIT\_VBATEN;  
  break;  
 default:  
  break;  
 }

 dev\_dbg(dev, "%s: %02x %02x %02x %02x %02x %02x %02x\n",  
  "write", buf[0], buf[1], buf[2], buf[3],  
  buf[4], buf[5], buf[6]);

 result = ds1307->write\_block\_data(ds1307->client,  
  ds1307->offset, 7, buf);  
 if (result < 0) {  
  dev\_err(dev, "%s error %d\n", "write", result);  
  return result;  
 }  
 return 0;  
}

static int ds1337\_read\_alarm(struct device \*dev, struct rtc\_wkalrm \*t)  
{  
 struct i2c\_client       \*client = to\_i2c\_client(dev);  
 struct ds1307  \*ds1307 = i2c\_get\_clientdata(client);  
 int   ret;

 if (!test\_bit(HAS\_ALARM, &ds1307->flags))  
  return -EINVAL;

 /\* read all ALARM1, ALARM2, and status registers at once \*/  
 ret = ds1307->read\_block\_data(client,  
   DS1339\_REG\_ALARM1\_SECS, 9, ds1307->regs);  
 if (ret != 9) {  
  dev\_err(dev, "%s error %d\n", "alarm read", ret);  
  return -EIO;  
 }

 dev\_dbg(dev, "%s: %02x %02x %02x %02x, %02x %02x %02x, %02x %02x\n",  
   "alarm read",  
   ds1307->regs[0], ds1307->regs[1],  
   ds1307->regs[2], ds1307->regs[3],  
   ds1307->regs[4], ds1307->regs[5],  
   ds1307->regs[6], ds1307->regs[7],  
   ds1307->regs[8]);

 /\*  
  \* report alarm time (ALARM1); assume 24 hour and day-of-month modes,  
  \* and that all four fields are checked matches  
  \*/  
 t->time.tm\_sec = bcd2bin(ds1307->regs[0] & 0x7f);  
 t->time.tm\_min = bcd2bin(ds1307->regs[1] & 0x7f);  
 t->time.tm\_hour = bcd2bin(ds1307->regs[2] & 0x3f);  
 t->time.tm\_mday = bcd2bin(ds1307->regs[3] & 0x3f);  
 t->time.tm\_mon = -1;  
 t->time.tm\_year = -1;  
 t->time.tm\_wday = -1;  
 t->time.tm\_yday = -1;  
 t->time.tm\_isdst = -1;

 /\* ... and status \*/  
 t->enabled = !!(ds1307->regs[7] & DS1337\_BIT\_A1IE);  
 t->pending = !!(ds1307->regs[8] & DS1337\_BIT\_A1I);

 dev\_dbg(dev, "%s secs=%d, mins=%d, "  
  "hours=%d, mday=%d, enabled=%d, pending=%d\n",  
  "alarm read", t->time.tm\_sec, t->time.tm\_min,  
  t->time.tm\_hour, t->time.tm\_mday,  
  t->enabled, t->pending);

 return 0;  
}

static int ds1337\_set\_alarm(struct device \*dev, struct rtc\_wkalrm \*t)  
{  
 struct i2c\_client \*client = to\_i2c\_client(dev);  
 struct ds1307  \*ds1307 = i2c\_get\_clientdata(client);  
 unsigned char  \*buf = ds1307->regs;  
 u8   control, status;  
 int   ret;

 if (!test\_bit(HAS\_ALARM, &ds1307->flags))  
  return -EINVAL;

 dev\_dbg(dev, "%s secs=%d, mins=%d, "  
  "hours=%d, mday=%d, enabled=%d, pending=%d\n",  
  "alarm set", t->time.tm\_sec, t->time.tm\_min,  
  t->time.tm\_hour, t->time.tm\_mday,  
  t->enabled, t->pending);

 /\* read current status of both alarms and the chip \*/  
 ret = ds1307->read\_block\_data(client,  
   DS1339\_REG\_ALARM1\_SECS, 9, buf);  
 if (ret != 9) {  
  dev\_err(dev, "%s error %d\n", "alarm write", ret);  
  return -EIO;  
 }  
 control = ds1307->regs[7];  
 status = ds1307->regs[8];

 dev\_dbg(dev, "%s: %02x %02x %02x %02x, %02x %02x %02x, %02x %02x\n",  
   "alarm set (old status)",  
   ds1307->regs[0], ds1307->regs[1],  
   ds1307->regs[2], ds1307->regs[3],  
   ds1307->regs[4], ds1307->regs[5],  
   ds1307->regs[6], control, status);

 /\* set ALARM1, using 24 hour and day-of-month modes \*/  
 buf[0] = bin2bcd(t->time.tm\_sec);  
 buf[1] = bin2bcd(t->time.tm\_min);  
 buf[2] = bin2bcd(t->time.tm\_hour);  
 buf[3] = bin2bcd(t->time.tm\_mday);

 /\* set ALARM2 to non-garbage \*/  
 buf[4] = 0;  
 buf[5] = 0;  
 buf[6] = 0;

 /\* optionally enable ALARM1 \*/  
 buf[7] = control & ~(DS1337\_BIT\_A1IE | DS1337\_BIT\_A2IE);  
 if (t->enabled) {  
  dev\_dbg(dev, "alarm IRQ armed\n");  
  buf[7] |= DS1337\_BIT\_A1IE; /\* only ALARM1 is used \*/  
 }  
 buf[8] = status & ~(DS1337\_BIT\_A1I | DS1337\_BIT\_A2I);

 ret = ds1307->write\_block\_data(client,  
   DS1339\_REG\_ALARM1\_SECS, 9, buf);  
 if (ret < 0) {  
  dev\_err(dev, "can't set alarm time\n");  
  return ret;  
 }

 return 0;  
}

static int ds1307\_alarm\_irq\_enable(struct device \*dev, unsigned int enabled)  
{  
 struct i2c\_client \*client = to\_i2c\_client(dev);  
 struct ds1307  \*ds1307 = i2c\_get\_clientdata(client);  
 int   ret;

 if (!test\_bit(HAS\_ALARM, &ds1307->flags))  
  return -ENOTTY;

 ret = i2c\_smbus\_read\_byte\_data(client, DS1337\_REG\_CONTROL);  
 if (ret < 0)  
  return ret;

 if (enabled)  
  ret |= DS1337\_BIT\_A1IE;  
 else  
  ret &= ~DS1337\_BIT\_A1IE;

 ret = i2c\_smbus\_write\_byte\_data(client, DS1337\_REG\_CONTROL, ret);  
 if (ret < 0)  
  return ret;

 return 0;  
}

static const struct rtc\_class\_ops ds13xx\_rtc\_ops = {  
 .read\_time = ds1307\_get\_time,  
 .set\_time = ds1307\_set\_time,  
 .read\_alarm = ds1337\_read\_alarm,  
 .set\_alarm = ds1337\_set\_alarm,  
 .alarm\_irq\_enable = ds1307\_alarm\_irq\_enable,  
};

/\*----------------------------------------------------------------------\*/

static ssize\_t  
ds1307\_nvram\_read(struct file \*filp, struct kobject \*kobj,  
  struct bin\_attribute \*attr,  
  char \*buf, loff\_t off, size\_t count)  
{  
 struct i2c\_client \*client;  
 struct ds1307  \*ds1307;  
 int   result;

 client = kobj\_to\_i2c\_client(kobj);  
 ds1307 = i2c\_get\_clientdata(client);

 if (unlikely(off >= ds1307->nvram->size))  
  return 0;  
 if ((off + count) > ds1307->nvram->size)  
  count = ds1307->nvram->size - off;  
 if (unlikely(!count))  
  return count;

 result = ds1307->read\_block\_data(client, ds1307->nvram\_offset + off,  
        count, buf);  
 if (result < 0)  
  dev\_err(&client->dev, "%s error %d\n", "nvram read", result);  
 return result;  
}

static ssize\_t  
ds1307\_nvram\_write(struct file \*filp, struct kobject \*kobj,  
  struct bin\_attribute \*attr,  
  char \*buf, loff\_t off, size\_t count)  
{  
 struct i2c\_client \*client;  
 struct ds1307  \*ds1307;  
 int   result;

 client = kobj\_to\_i2c\_client(kobj);  
 ds1307 = i2c\_get\_clientdata(client);

 if (unlikely(off >= ds1307->nvram->size))  
  return -EFBIG;  
 if ((off + count) > ds1307->nvram->size)  
  count = ds1307->nvram->size - off;  
 if (unlikely(!count))  
  return count;

 result = ds1307->write\_block\_data(client, ds1307->nvram\_offset + off,  
        count, buf);  
 if (result < 0) {  
  dev\_err(&client->dev, "%s error %d\n", "nvram write", result);  
  return result;  
 }  
 return count;  
}

/\*----------------------------------------------------------------------\*/

static int \_\_devinit ds1307\_probe(struct i2c\_client \*client,  
      const struct i2c\_device\_id \*id)  
{  
 struct ds1307  \*ds1307;  
 int   err = -ENODEV;  
 int   tmp;  
 const struct chip\_desc \*chip = &chips[id->driver\_data];  
 struct i2c\_adapter \*adapter = to\_i2c\_adapter(client->dev.parent);  
 int   want\_irq = false;  
 unsigned char  \*buf;  
 struct ds1307\_platform\_data \*pdata = client->dev.platform\_data;  
 static const int bbsqi\_bitpos[] = {  
  [ds\_1337] = 0,  
  [ds\_1339] = DS1339\_BIT\_BBSQI,  
  [ds\_3231] = DS3231\_BIT\_BBSQW,  
 };

 if (!i2c\_check\_functionality(adapter, I2C\_FUNC\_SMBUS\_BYTE\_DATA)  
     && !i2c\_check\_functionality(adapter, I2C\_FUNC\_SMBUS\_I2C\_BLOCK))  
  return -EIO;

 ds1307 = kzalloc(sizeof(struct ds1307), GFP\_KERNEL);  
 if (!ds1307)  
  return -ENOMEM;

 i2c\_set\_clientdata(client, ds1307);

 ds1307->client = client;  
 ds1307->type = id->driver\_data;

 if (pdata && pdata->trickle\_charger\_setup && chip->trickle\_charger\_reg)  
  i2c\_smbus\_write\_byte\_data(client, chip->trickle\_charger\_reg,  
   DS13XX\_TRICKLE\_CHARGER\_MAGIC | pdata->trickle\_charger\_setup);

 buf = ds1307->regs;  
 if (i2c\_check\_functionality(adapter, I2C\_FUNC\_SMBUS\_I2C\_BLOCK)) {  
  ds1307->read\_block\_data = i2c\_smbus\_read\_i2c\_block\_data;  
  ds1307->write\_block\_data = i2c\_smbus\_write\_i2c\_block\_data;  
 } else {  
  ds1307->read\_block\_data = ds1307\_read\_block\_data;  
  ds1307->write\_block\_data = ds1307\_write\_block\_data;  
 }

 switch (ds1307->type) {  
 case ds\_1337:  
 case ds\_1339:  
 case ds\_3231:  
  /\* get registers that the "rtc" read below won't read... \*/  
  tmp = ds1307->read\_block\_data(ds1307->client,  
    DS1337\_REG\_CONTROL, 2, buf);  
  if (tmp != 2) {  
   pr\_debug("read error %d\n", tmp);  
   err = -EIO;  
   goto exit\_free;  
  }

  /\* oscillator off?  turn it on, so clock can tick. \*/  
  if (ds1307->regs[0] & DS1337\_BIT\_nEOSC)  
   ds1307->regs[0] &= ~DS1337\_BIT\_nEOSC;

  /\*  
   \* Using IRQ?  Disable the square wave and both alarms.  
   \* For some variants, be sure alarms can trigger when we're  
   \* running on Vbackup (BBSQI/BBSQW)  
   \*/  
  if (ds1307->client->irq > 0 && chip->alarm) {  
   INIT\_WORK(&ds1307->work, ds1307\_work);

   ds1307->regs[0] |= DS1337\_BIT\_INTCN  
     | bbsqi\_bitpos[ds1307->type];  
   ds1307->regs[0] &= ~(DS1337\_BIT\_A2IE | DS1337\_BIT\_A1IE);

   want\_irq = true;  
  }

  i2c\_smbus\_write\_byte\_data(client, DS1337\_REG\_CONTROL,  
       ds1307->regs[0]);

  /\* oscillator fault?  clear flag, and warn \*/  
  if (ds1307->regs[1] & DS1337\_BIT\_OSF) {  
   i2c\_smbus\_write\_byte\_data(client, DS1337\_REG\_STATUS,  
    ds1307->regs[1] & ~DS1337\_BIT\_OSF);  
   dev\_warn(&client->dev, "SET TIME!\n");  
  }  
  break;

 case rx\_8025:  
  tmp = i2c\_smbus\_read\_i2c\_block\_data(ds1307->client,  
    RX8025\_REG\_CTRL1 << 4 | 0x08, 2, buf);  
  if (tmp != 2) {  
   pr\_debug("read error %d\n", tmp);  
   err = -EIO;  
   goto exit\_free;  
  }

  /\* oscillator off?  turn it on, so clock can tick. \*/  
  if (!(ds1307->regs[1] & RX8025\_BIT\_XST)) {  
   ds1307->regs[1] |= RX8025\_BIT\_XST;  
   i2c\_smbus\_write\_byte\_data(client,  
        RX8025\_REG\_CTRL2 << 4 | 0x08,  
        ds1307->regs[1]);  
   dev\_warn(&client->dev,  
     "oscillator stop detected - SET TIME!\n");  
  }

  if (ds1307->regs[1] & RX8025\_BIT\_PON) {  
   ds1307->regs[1] &= ~RX8025\_BIT\_PON;  
   i2c\_smbus\_write\_byte\_data(client,  
        RX8025\_REG\_CTRL2 << 4 | 0x08,  
        ds1307->regs[1]);  
   dev\_warn(&client->dev, "power-on detected\n");  
  }

  if (ds1307->regs[1] & RX8025\_BIT\_VDET) {  
   ds1307->regs[1] &= ~RX8025\_BIT\_VDET;  
   i2c\_smbus\_write\_byte\_data(client,  
        RX8025\_REG\_CTRL2 << 4 | 0x08,  
        ds1307->regs[1]);  
   dev\_warn(&client->dev, "voltage drop detected\n");  
  }

  /\* make sure we are running in 24hour mode \*/  
  if (!(ds1307->regs[0] & RX8025\_BIT\_2412)) {  
   u8 hour;

   /\* switch to 24 hour mode \*/  
   i2c\_smbus\_write\_byte\_data(client,  
        RX8025\_REG\_CTRL1 << 4 | 0x08,  
        ds1307->regs[0] |  
        RX8025\_BIT\_2412);

   tmp = i2c\_smbus\_read\_i2c\_block\_data(ds1307->client,  
     RX8025\_REG\_CTRL1 << 4 | 0x08, 2, buf);  
   if (tmp != 2) {  
    pr\_debug("read error %d\n", tmp);  
    err = -EIO;  
    goto exit\_free;  
   }

   /\* correct hour \*/  
   hour = bcd2bin(ds1307->regs[DS1307\_REG\_HOUR]);  
   if (hour == 12)  
    hour = 0;  
   if (ds1307->regs[DS1307\_REG\_HOUR] & DS1307\_BIT\_PM)  
    hour += 12;

   i2c\_smbus\_write\_byte\_data(client,  
        DS1307\_REG\_HOUR << 4 | 0x08,  
        hour);  
  }  
  break;  
 case ds\_1388:  
  ds1307->offset = 1; /\* Seconds starts at 1 \*/  
  break;  
 default:  
  break;  
 }

read\_rtc:  
 /\* read RTC registers \*/  
 tmp = ds1307->read\_block\_data(ds1307->client, ds1307->offset, 8, buf);  
 if (tmp != 8) {  
  pr\_debug("read error %d\n", tmp);  
  err = -EIO;  
  goto exit\_free;  
 }

 /\*  
  \* minimal sanity checking; some chips (like DS1340) don't  
  \* specify the extra bits as must-be-zero, but there are  
  \* still a few values that are clearly out-of-range.  
  \*/  
 tmp = ds1307->regs[DS1307\_REG\_SECS];  
 switch (ds1307->type) {  
 case ds\_1307:  
 case m41t00:  
  /\* clock halted?  turn it on, so clock can tick. \*/  
  if (tmp & DS1307\_BIT\_CH) {  
   i2c\_smbus\_write\_byte\_data(client, DS1307\_REG\_SECS, 0);  
   dev\_warn(&client->dev, "SET TIME!\n");  
   goto read\_rtc;  
  }  
  break;  
 case ds\_1338:  
  /\* clock halted?  turn it on, so clock can tick. \*/  
  if (tmp & DS1307\_BIT\_CH)  
   i2c\_smbus\_write\_byte\_data(client, DS1307\_REG\_SECS, 0);

  /\* oscillator fault?  clear flag, and warn \*/  
  if (ds1307->regs[DS1307\_REG\_CONTROL] & DS1338\_BIT\_OSF) {  
   i2c\_smbus\_write\_byte\_data(client, DS1307\_REG\_CONTROL,  
     ds1307->regs[DS1307\_REG\_CONTROL]  
     & ~DS1338\_BIT\_OSF);  
   dev\_warn(&client->dev, "SET TIME!\n");  
   goto read\_rtc;  
  }  
  break;  
 case ds\_1340:  
  /\* clock halted?  turn it on, so clock can tick. \*/  
  if (tmp & DS1340\_BIT\_nEOSC)  
   i2c\_smbus\_write\_byte\_data(client, DS1307\_REG\_SECS, 0);

  tmp = i2c\_smbus\_read\_byte\_data(client, DS1340\_REG\_FLAG);  
  if (tmp < 0) {  
   pr\_debug("read error %d\n", tmp);  
   err = -EIO;  
   goto exit\_free;  
  }

  /\* oscillator fault?  clear flag, and warn \*/  
  if (tmp & DS1340\_BIT\_OSF) {  
   i2c\_smbus\_write\_byte\_data(client, DS1340\_REG\_FLAG, 0);  
   dev\_warn(&client->dev, "SET TIME!\n");  
  }  
  break;  
 case mcp7941x:  
  /\* make sure that the backup battery is enabled \*/  
  if (!(ds1307->regs[DS1307\_REG\_WDAY] & MCP7941X\_BIT\_VBATEN)) {  
   i2c\_smbus\_write\_byte\_data(client, DS1307\_REG\_WDAY,  
     ds1307->regs[DS1307\_REG\_WDAY]  
     | MCP7941X\_BIT\_VBATEN);  
  }

  /\* clock halted?  turn it on, so clock can tick. \*/  
  if (!(tmp & MCP7941X\_BIT\_ST)) {  
   i2c\_smbus\_write\_byte\_data(client, DS1307\_REG\_SECS,  
     MCP7941X\_BIT\_ST);  
   dev\_warn(&client->dev, "SET TIME!\n");  
   goto read\_rtc;  
  }

  break;  
 default:  
  break;  
 }

 tmp = ds1307->regs[DS1307\_REG\_HOUR];  
 switch (ds1307->type) {  
 case ds\_1340:  
 case m41t00:  
  /\*  
   \* NOTE: ignores century bits; fix before deploying  
   \* systems that will run through year 2100.  
   \*/  
  break;  
 case rx\_8025:  
  break;  
 default:  
  if (!(tmp & DS1307\_BIT\_12HR))  
   break;

  /\*  
   \* Be sure we're in 24 hour mode.  Multi-master systems  
   \* take note...  
   \*/  
  tmp = bcd2bin(tmp & 0x1f);  
  if (tmp == 12)  
   tmp = 0;  
  if (ds1307->regs[DS1307\_REG\_HOUR] & DS1307\_BIT\_PM)  
   tmp += 12;  
  i2c\_smbus\_write\_byte\_data(client,  
    ds1307->offset + DS1307\_REG\_HOUR,  
    bin2bcd(tmp));  
 }

 ds1307->rtc = rtc\_device\_register(client->name, &client->dev,  
    &ds13xx\_rtc\_ops, THIS\_MODULE);  
 if (IS\_ERR(ds1307->rtc)) {  
  err = PTR\_ERR(ds1307->rtc);  
  dev\_err(&client->dev,  
   "unable to register the class device\n");  
  goto exit\_free;  
 }

 if (want\_irq) {  
  err = request\_irq(client->irq, ds1307\_irq, IRQF\_SHARED,  
     ds1307->rtc->name, client);  
  if (err) {  
   dev\_err(&client->dev,  
    "unable to request IRQ!\n");  
   goto exit\_irq;  
  }

  device\_set\_wakeup\_capable(&client->dev, 1);  
  set\_bit(HAS\_ALARM, &ds1307->flags);  
  dev\_dbg(&client->dev, "got IRQ %d\n", client->irq);  
 }

 if (chip->nvram\_size) {  
  ds1307->nvram = kzalloc(sizeof(struct bin\_attribute),  
       GFP\_KERNEL);  
  if (!ds1307->nvram) {  
   err = -ENOMEM;  
   goto exit\_nvram;  
  }  
  ds1307->nvram->attr.name = "nvram";  
  ds1307->nvram->attr.mode = S\_IRUGO | S\_IWUSR;  
  sysfs\_bin\_attr\_init(ds1307->nvram);  
  ds1307->nvram->read = ds1307\_nvram\_read,  
  ds1307->nvram->write = ds1307\_nvram\_write,  
  ds1307->nvram->size = chip->nvram\_size;  
  ds1307->nvram\_offset = chip->nvram\_offset;  
  err = sysfs\_create\_bin\_file(&client->dev.kobj, ds1307->nvram);  
  if (err) {  
   kfree(ds1307->nvram);  
   goto exit\_nvram;  
  }  
  set\_bit(HAS\_NVRAM, &ds1307->flags);  
  dev\_info(&client->dev, "%zu bytes nvram\n", ds1307->nvram->size);  
 }

 return 0;

exit\_nvram:  
exit\_irq:  
 rtc\_device\_unregister(ds1307->rtc);  
exit\_free:  
 kfree(ds1307);  
 return err;  
}

static int \_\_devexit ds1307\_remove(struct i2c\_client \*client)  
{  
 struct ds1307 \*ds1307 = i2c\_get\_clientdata(client);

 if (test\_and\_clear\_bit(HAS\_ALARM, &ds1307->flags)) {  
  free\_irq(client->irq, client);  
  cancel\_work\_sync(&ds1307->work);  
 }

 if (test\_and\_clear\_bit(HAS\_NVRAM, &ds1307->flags)) {  
  sysfs\_remove\_bin\_file(&client->dev.kobj, ds1307->nvram);  
  kfree(ds1307->nvram);  
 }

 rtc\_device\_unregister(ds1307->rtc);  
 kfree(ds1307);  
 return 0;  
}

static struct i2c\_driver ds1307\_driver = {  
 .driver = {  
  .name = "rtc-ds1307",  
  .owner = THIS\_MODULE,  
 },  
 .probe  = ds1307\_probe,  
 .remove  = \_\_devexit\_p(ds1307\_remove),  
 .id\_table = ds1307\_id,  
};

module\_i2c\_driver(ds1307\_driver);

MODULE\_DESCRIPTION("RTC driver for DS1307 and similar chips");  
MODULE\_LICENSE("GPL");

//////////////////////end of file//////////////////////////