Real Time Clock (RTC) Drivers for Linux

When Linux developers talk about a "Real Time Clock", they usually mean something that tracks wall clock time and is battery backed so that it works even with system power off. Such clocks will normally not track the local time zone or daylight savings time -- unless they dual boot with MS-Windows -- but will instead be set to Coordinated Universal Time (UTC, formerly "Greenwich Mean Time").

The newest non-PC hardware tends to just count seconds, like the time(2) system call reports, but RTCs also very commonly represent time using the Gregorian calendar and 24 hour time, as reported by gmtime(3).

Linux has two largely-compatible userspace RTC API families you may need to know about:

* /dev/rtc ... is the RTC provided by PC compatible systems, so it's not very portable to non-x86 systems.

* /dev/rtc0, /dev/rtc1 ... are part of a framework that's supported by a wide variety of RTC chips on all systems.

Programmers need to understand that the PC/AT functionality is not always available, and some systems can do much more. That is, the RTCs use the same API to make requests in both RTC frameworks (using different filenames of course), but the hardware may not offer the same functionality. For example, not every RTC is hooked up to an IRQ, so they can't all issue alarms; and where standard PC RTCs can only issue an alarm up to 24 hours in the future, other hardware may be able to schedule one any time in the upcoming century.

Old PC/AT-Compatible driver: /dev/rtc

All PCs (even Alpha machines) have a Real Time Clock built into them. Usually they are built into the chipset of the computer, but some may actually have a Motorola MC146818 (or clone) on the board. This is the clock that keeps the date and time while your computer is turned off.

ACPI has standardized that MC146818 functionality, and extended it in a few ways (enabling longer alarm periods, and wake-from-hibernate). That functionality is NOT exposed in the old driver.

However it can also be used to generate signals from a slow 2Hz to a relatively fast 8192Hz, in increments of powers of two. These signals are reported by interrupt number 8. (Oh! So *that* is what IRQ 8 is for...) It can also function as a 24hr alarm, raising IRQ 8 when the alarm goes off. The alarm can also be programmed to only check any subset of the three programmable values, meaning that it could be set to ring on the 30th second of the 30th minute of every hour, for example. The clock can also be set to generate an interrupt upon every clock update, thus generating a 1Hz signal.

The interrupts are reported via /dev/rtc (major 10, minor 135, read only character device) in the form of an unsigned long. The low byte contains the type of interrupt (update-done, alarm-rang, or periodic) that was raised, and the remaining bytes contain the number of interrupts since the last read. Status information is reported through the pseudo-file /proc/driver/rtc if the /proc filesystem was enabled. The driver has built in locking so that only one process is allowed to have the /dev/rtc interface open at a time.

A user process can monitor these interrupts by doing a read(2) or a select(2) on /dev/rtc -- either will block/stop the user process until the next interrupt is received. This is useful for things like reasonably high frequency data acquisition where one doesn't want to burn up 100% CPU by polling gettimeofday etc. etc.

At high frequencies, or under high loads, the user process should check the number of interrupts received since the last read to determine if there has been any interrupt "pileup" so to speak. Just for reference, a typical 486-33 running a tight read loop on /dev/rtc will start to suffer occasional interrupt pileup (i.e. > 1 IRQ event since last read) for frequencies above 1024Hz. So you really should check the high bytes of the value you read, especially at frequencies above that of the normal timer interrupt, which is 100Hz.

Programming and/or enabling interrupt frequencies greater than 64Hz is only allowed by root. This is perhaps a bit conservative, but we don't want an evil user generating lots of IRQs on a slow 386sx-16, where it might have a negative impact on performance. This 64Hz limit can be changed by writing a different value to /proc/sys/dev/rtc/max-user-freq. Note that the interrupt handler is only a few lines of code to minimize any possibility of this effect.

Also, if the kernel time is synchronized with an external source, the kernel will write the time back to the CMOS clock every 11 minutes. In the process of doing this, the kernel briefly turns off RTC periodic interrupts, so be aware of this if you are doing serious work. If you don't synchronize the kernel time with an external source (via ntp or whatever) then the kernel will keep its hands off the RTC, allowing you exclusive access to the device for your applications.

The alarm and/or interrupt frequency are programmed into the RTC via various ioctl(2) calls as listed in ./include/linux/rtc.h
Rather than write 50 pages describing the ioctl() and so on, it is perhaps more useful to include a small test program that demonstrates how to use them, and demonstrates the features of the driver. This is probably a lot more useful to people interested in writing applications that will be using this driver. See the code at the end of this document.

(The original /dev/rtc driver was written by Paul Gortmaker.)

New portable "RTC Class" drivers: /dev/rtcN

Because Linux supports many non-ACPI and non-PC platforms, some of which have more than one RTC style clock, it needed a more portable solution

than expecting a single battery-backed MC146818 clone on every system. Accordingly, a new "RTC Class" framework has been defined. It offers three different userspace interfaces:

* /dev/rtcN ... much the same as the older /dev/rtc interface

* /sys/class/rtc/rtcN ... sysfs attributes support readonly access to some RTC attributes.

* /proc/driver/rtc ... the first RTC (rtc0) may expose itself using a procfs interface. More information is (currently) shown here than through sysfs.

The RTC Class framework supports a wide variety of RTCs, ranging from those integrated into embeddable system-on-chip (SOC) processors to discrete chips using I2C, SPI, or some other bus to communicate with the host CPU. There's even support for PC-style RTCs ... including the features exposed on newer PCs through ACPI.

The new framework also removes the "one RTC per system" restriction. For example, maybe the low-power battery-backed RTC is a discrete I2C chip, but a high functionality RTC is integrated into the SOC. That system might read the system clock from the discrete RTC, but use the integrated one for all other tasks, because of its greater functionality.

SYSFS INTERFACE

The sysfs interface under /sys/class/rtc/rtcN provides access to various rtc attributes without requiring the use of ioctls. All dates and times are in the RTC's timezone, rather than in system time.

date:	RTC-provided date
hctosys:	1 if the RTC provided the system time at boot via the CONFIG_RTC_HCTOSYS kernel option, 0 otherwise
max_user_freq:	The maximum interrupt rate an unprivileged user may request from this RTC.
name:	The name of the RTC corresponding to this sysfs directory
since_epoch:	The number of seconds since the epoch according to the RTC
time:	RTC-provided time
wakealarm:	The time at which the clock will generate a system wakeup event. This is a one shot wakeup event, so must be reset after wake if a daily wakeup is required. Format is either seconds since the epoch or, if there's a leading +, seconds in the future.

IOCTL INTERFACE

The ioctl() calls supported by /dev/rtc are also supported by the RTC class framework. However, because the chips and systems are not standardized, some PC/AT functionality might not be provided. And in the same way, some newer features -- including those enabled by ACPI -- are exposed by the RTC class framework, but can't be supported by the older driver.

* RTC_RD_TIME, RTC_SET_TIME ... every RTC supports at least reading

time, returning the result as a Gregorian calendar date and 24 hour wall clock time. To be most useful, this time may also be updated.

* RTC_AIE_ON, RTC_AIE_OFF, RTC_ALM_SET, RTC_ALM_READ ... when the RTC is connected to an IRQ line, it can often issue an alarm IRQ up to 24 hours in the future. (Use RTC_WKALM_* by preference.)

* RTC_WKALM_SET, RTC_WKALM_RD ... RTCs that can issue alarms beyond the next 24 hours use a slightly more powerful API, which supports setting the longer alarm time and enabling its IRQ using a single request (using the same model as EFI firmware).

* RTC_UIE_ON, RTC_UIE_OFF ... if the RTC offers IRQs, it probably also offers update IRQs whenever the "seconds" counter changes. If needed, the RTC framework can emulate this mechanism.

* RTC_PIE_ON, RTC_PIE_OFF, RTC_IRQP_SET, RTC_IRQP_READ ... another feature often accessible with an IRQ line is a periodic IRQ, issued at settable frequencies (usually 2^N Hz).

In many cases, the RTC alarm can be a system wake event, used to force Linux out of a low power sleep state (or hibernation) back to a fully operational state. For example, a system could enter a deep power saving state until it's time to execute some scheduled tasks.

Note that many of these ioctls need not actually be implemented by your driver. The common rtc-dev interface handles many of these nicely if your driver returns ENOIOCTLCMD. Some common examples:

* RTC_RD_TIME, RTC_SET_TIME: the read_time/set_time functions will be called with appropriate values.

* RTC_ALM_SET, RTC_ALM_READ, RTC_WKALM_SET, RTC_WKALM_RD: the set_alarm/read_alarm functions will be called.

* RTC_IRQP_SET, RTC_IRQP_READ: the irq_set_freq function will be called to set the frequency while the framework will handle the read for you since the frequency is stored in the irq_freq member of the rtc_device structure. Your driver needs to initialize the irq_freq member during init. Make sure you check the requested frequency is in range of your hardware in the irq_set_freq function. If it isn't, return -EINVAL. If you cannot actually change the frequency, do not define irq_set_freq.

* RTC_PIE_ON, RTC_PIE_OFF: the irq_set_state function will be called.

If all else fails, check out the rtc-test.c driver!

---------- 8< ---------- 8< ----------

```
/*
 *      Real Time Clock Driver Test/Example Program
 *
 *      Compile with:
 *                  gcc -s -Wall -Wstrict-prototypes rtctest.c -o rtctest
 *
```

```
 *      Copyright (C) 1996, Paul Gortmaker.
 *
 *      Released under the GNU General Public License, version 2,
 *      included herein by reference.
 *
 */

#include <stdio.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>


/*
 * This expects the new RTC class driver framework, working with
 * clocks that will often not be clones of what the PC-AT had.
 * Use the command line to specify another RTC if you need one.
 */
static const char default_rtc[] = "/dev/rtc0";


int main(int argc, char **argv)
{
        int i, fd, retval, irqcount = 0;
        unsigned long tmp, data;
        struct rtc_time rtc_tm;
        const char *rtc = default_rtc;

        switch (argc) {
        case 2:
                rtc = argv[1];
                /* FALLTHROUGH */
        case 1:
                break;
        default:
                fprintf(stderr, "usage:  rtctest [rtcdev]\n");
                return 1;
        }

        fd = open(rtc, O_RDONLY);

        if (fd ==  -1) {
                perror(rtc);
                exit(errno);
        }

        fprintf(stderr, "\n\t\t\tRTC Driver Test Example.\n\n");

        /* Turn on update interrupts (one per second) */
        retval = ioctl(fd, RTC_UIE_ON, 0);
        if (retval == -1) {
```

```
        if (errno == ENOTTY) {
                fprintf(stderr,
                        "\n...Update IRQs not supported.\n");
                goto test_READ;
        }
        perror("RTC_UIE_ON ioctl");
        exit(errno);
}

fprintf(stderr, "Counting 5 update (1/sec) interrupts from reading %s:",
                rtc);
fflush(stderr);
for (i=1; i<6; i++) {
        /* This read will block */
        retval = read(fd, &data, sizeof(unsigned long));
        if (retval == -1) {
                perror("read");
                exit(errno);
        }
        fprintf(stderr, " %d",i);
        fflush(stderr);
        irqcount++;
}

fprintf(stderr, "\nAgain, from using select(2) on /dev/rtc:");
fflush(stderr);
for (i=1; i<6; i++) {
        struct timeval tv = {5, 0};     /* 5 second timeout on select */
        fd_set readfds;

        FD_ZERO(&readfds);
        FD_SET(fd, &readfds);
        /* The select will wait until an RTC interrupt happens. */
        retval = select(fd+1, &readfds, NULL, NULL, &tv);
        if (retval == -1) {
                perror("select");
                exit(errno);
        }
        /* This read won't block unlike the select-less case above. */
        retval = read(fd, &data, sizeof(unsigned long));
        if (retval == -1) {
                perror("read");
                exit(errno);
        }
        fprintf(stderr, " %d",i);
        fflush(stderr);
        irqcount++;
}

/* Turn off update interrupts */
retval = ioctl(fd, RTC_UIE_OFF, 0);
if (retval == -1) {
        perror("RTC_UIE_OFF ioctl");
        exit(errno);
}
```

```
test_READ:
        /* Read the RTC time/date */
        retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
        if (retval == -1) {
                perror("RTC_RD_TIME ioctl");
                exit(errno);
        }

        fprintf(stderr, "\n\nCurrent RTC date/time is %d-%d-%d,
%02d:%02d:%02d.\n",
                rtc_tm.tm_mday, rtc_tm.tm_mon + 1, rtc_tm.tm_year + 1900,
                rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

        /* Set the alarm to 5 sec in the future, and check for rollover */
        rtc_tm.tm_sec += 5;
        if (rtc_tm.tm_sec >= 60) {
                rtc_tm.tm_sec %= 60;
                rtc_tm.tm_min++;
        }
        if (rtc_tm.tm_min == 60) {
                rtc_tm.tm_min = 0;
                rtc_tm.tm_hour++;
        }
        if (rtc_tm.tm_hour == 24)
                rtc_tm.tm_hour = 0;

        retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
        if (retval == -1) {
                if (errno == ENOTTY) {
                        fprintf(stderr,
                                "\n...Alarm IRQs not supported.\n");
                        goto test_PIE;
                }
                perror("RTC_ALM_SET ioctl");
                exit(errno);
        }

        /* Read the current alarm settings */
        retval = ioctl(fd, RTC_ALM_READ, &rtc_tm);
        if (retval == -1) {
                perror("RTC_ALM_READ ioctl");
                exit(errno);
        }

        fprintf(stderr, "Alarm time now set to %02d:%02d:%02d.\n",
                rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

        /* Enable alarm interrupts */
        retval = ioctl(fd, RTC_AIE_ON, 0);
        if (retval == -1) {
                perror("RTC_AIE_ON ioctl");
                exit(errno);
        }

        fprintf(stderr, "Waiting 5 seconds for alarm...");
        fflush(stderr);
```

```
        /* This blocks until the alarm ring causes an interrupt */
        retval = read(fd, &data, sizeof(unsigned long));
        if (retval == -1) {
                perror("read");
                exit(errno);
        }
        irqcount++;
        fprintf(stderr, " okay. Alarm rang.\n");

        /* Disable alarm interrupts */
        retval = ioctl(fd, RTC_AIE_OFF, 0);
        if (retval == -1) {
                perror("RTC_AIE_OFF ioctl");
                exit(errno);
        }

test_PIE:
        /* Read periodic IRQ rate */
        retval = ioctl(fd, RTC_IRQP_READ, &tmp);
        if (retval == -1) {
                /* not all RTCs support periodic IRQs */
                if (errno == ENOTTY) {
                        fprintf(stderr, "\nNo periodic IRQ support\n");
                        goto done;
                }
                perror("RTC_IRQP_READ ioctl");
                exit(errno);
        }
        fprintf(stderr, "\nPeriodic IRQ rate is %ldHz.\n", tmp);

        fprintf(stderr, "Counting 20 interrupts at:");
        fflush(stderr);

        /* The frequencies 128Hz, 256Hz, ... 8192Hz are only allowed for root.
*/
        for (tmp=2; tmp<=64; tmp*=2) {

                retval = ioctl(fd, RTC_IRQP_SET, tmp);
                if (retval == -1) {
                        /* not all RTCs can change their periodic IRQ rate */
                        if (errno == ENOTTY) {
                                fprintf(stderr,
                                        "\n...Periodic IRQ rate is fixed\n");
                                goto done;
                        }
                        perror("RTC_IRQP_SET ioctl");
                        exit(errno);
                }

                fprintf(stderr, "\n%ldHz:\t", tmp);
                fflush(stderr);

                /* Enable periodic interrupts */
                retval = ioctl(fd, RTC_PIE_ON, 0);
                if (retval == -1) {
                        perror("RTC_PIE_ON ioctl");
```

```
                exit(errno);
        }

        for (i=1; i<21; i++) {
                /* This blocks */
                retval = read(fd, &data, sizeof(unsigned long));
                if (retval == -1) {
                        perror("read");
                        exit(errno);
                }
                fprintf(stderr, " %d",i);
                fflush(stderr);
                irqcount++;
        }

        /* Disable periodic interrupts */
        retval = ioctl(fd, RTC_PIE_OFF, 0);
        if (retval == -1) {
                perror("RTC_PIE_OFF ioctl");
                exit(errno);
        }
    }

done:
    fprintf(stderr, "\n\n\t\t\t *** Test complete ***\n");

    close(fd);

    return 0;
}
```