Linux Watchdog Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

HPE ILO NMI WATCHDOG DRIVER

1.1 for iLO based ProLiant Servers

Last reviewed: 08/20/2018

The HPE iLO NMI Watchdog driver is a kernel module that provides basic watchdog functionality and handler for the iLO “Generate NMI to System” virtual button.

All references to iLO in this document imply it also works on iLO2 and all subsequent generations.

Watchdog functionality is enabled like any other common watchdog driver. That is, an application needs to be started that kicks off the watchdog timer. A basic application exists in tools/testing/selftests/watchdog/ named watchdog-test.c. Simply compile the C file and kick it off. If the system gets into a bad state and hangs, the HPE ProLiant iLO timer register will not be updated in a timely fashion and a hardware system reset (also known as an Automatic Server Recovery (ASR)) event will occur.

The hpwdt driver also has the following module parameters:

soft_	allows the user to set the watchdog timer value. Default value is 30 seconds.
time- out	an alias of soft_margin.
pre- time- out	allows the user to set the watchdog pretimeout value. This is the number of seconds before timeout when an NMI is delivered to the system. Setting the value to zero disables the pretimeout NMI. Default value is 9 seconds.
nowa out	basic watchdog parameter that does not allow the timer to be restarted or an impending ASR to be escaped. Default value is set when compiling the kernel. If it is set to “Y”, then there is no way of disabling the watchdog once it has been started.
kdun ti- me- out	Minimum timeout in seconds to apply upon receipt of an NMI before calling panic. (-1) disables the watchdog. When value is > 0, the timer is reprogrammed with the greater of value or current timeout value.

NOTE:
More information about watchdog drivers in general, including the ioctl interface to /dev/watchdog can be found in *The Linux Watchdog driver API* and Documentation/IPMI.txt.

Due to limitations in the iLO hardware, the NMI pretimeout if enabled, can only be set to 9 seconds. Attempts to set pretimeout to other non-zero values will be rounded, possibly to zero. Users should verify the pretimeout value after attempting to set pretimeout or timeout.

Upon receipt of an NMI from the iLO, the hpwdt driver will initiate a panic. This is to allow for a crash dump to be collected. It is incumbent upon the user to have properly configured the system for kdump.

The default Linux kernel behavior upon panic is to print a kernel tombstone and loop forever. This is generally not what a watchdog user wants.

For those wishing to learn more please see:
Documentation/admin-guide/kdump/kdump.rst
Documentation/admin-guide/kernel-parameters.txt (panic=) Your Linux Distribution specific documentation.

If the hpwdt does not receive the NMI associated with an expiring timer, the iLO will proceed to reset the system at timeout if the timer hasn't been updated.

–

The HPE iLO NMI Watchdog Driver and documentation were originally developed by Tom Mingarelli.

CHAPTER

TWO

MELLANOX WATCHDOG DRIVERS

2.1 for x86 based system switches

This driver provides watchdog functionality for various Mellanox Ethernet and Infiniband switch systems.

Mellanox watchdog device is implemented in a programmable logic device.

There are 2 types of HW watchdog implementations.

Type 1:
Actual HW timeout can be defined as a power of 2 msec. e.g. timeout 20 sec will be rounded up to 32768 msec. The maximum timeout period is 32 sec (32768 msec.), Get time-left isn't supported

Type 2:
Actual HW timeout is defined in sec. and it's the same as a user-defined timeout. Maximum timeout is 255 sec. Get time-left is supported.

Type 3:
Same as Type 2 with extended maximum timeout period. Maximum timeout is 65535 sec.

Type 1 HW watchdog implementation exist in old systems and all new systems have type 2 HW watchdog. Two types of HW implementation have also different register map.

Type 3 HW watchdog implementation can exist on all Mellanox systems with new programmer logic device. It's differentiated by WD capability bit. Old systems still have only one main watchdog.

Mellanox system can have 2 watchdogs: main and auxiliary. Main and auxiliary watchdog devices can be enabled together on the same system. There are several actions that can be defined in the watchdog: system reset, start fans on full speed and increase register counter. The last 2 actions are performed without a system reset. Actions without reset are provided for auxiliary watchdog device, which is optional. Watchdog can be started during a probe, in this case it will be pinged by watchdog core before watchdog device will be opened by user space application. Watchdog can be initialised in nowayout way, i.e. oncse started it can't be stopped.

This mlx-wdt driver supports both HW watchdog implementations.

Watchdog driver is probed from the common mlx_platform driver. Mlx_platform driver provides an appropriate set of registers for Mellanox watchdog device, identity name (mlx-wdt-main or mlx-wdt-aux), initial timeout, performed action in ex-

piration and configuration flags. watchdog configuration flags: nowayout and start_at_boot, hw watchdog version - type1 or type2. The driver checks during initialization if the previous system reset was done by the watchdog. If yes, it makes a notification about this event.

Access to HW registers is performed through a generic regmap interface. Programmable logic device registers have little-endian order.

CHAPTER

THREE

BERKSHIRE PRODUCTS PC WATCHDOG CARD

Last reviewed: 10/05/2007

3.1 Support for ISA Cards Revision A and C

Documentation and Driver by Ken Hollis <kenji@bitgate.com>

The PC Watchdog is a card that offers the same type of functionality that the WDT card does, only it doesn't require an IRQ to run. Furthermore, the Revision C card allows you to monitor any IO Port to automatically trigger the card into being reset. This way you can make the card monitor hard drive status, or anything else you need.

The Watchdog Driver has one basic role: to talk to the card and send signals to it so it doesn't reset your computer ···at least during normal operation.

The Watchdog Driver will automatically find your watchdog card, and will attach a running driver for use with that card. After the watchdog drivers have initialized, you can then talk to the card using a PC Watchdog program.

I suggest putting a "watchdog -d" before the beginning of an fsck, and a "watchdog -e -t 1" immediately after the end of an fsck. (Remember to run the program with an "&" to run it in the background!)

If you want to write a program to be compatible with the PC Watchdog driver, simply use of modify the watchdog test program: tools/testing/selftests/watchdog/watchdog-test.c

Other IOCTL functions include:

WDIOC_GETSUPPORT
This returns the support of the card itself. This returns in structure "PCWDS" which returns:

options = WDIOS_TEMPPANIC
(This card supports temperature)

firmware_version = xxxx
(Firmware version of the card)

WDIOC_GETSTATUS
This returns the status of the card, with the bits of

WDIOF_* bitwise-anded into the value. (The comments are in linux/pcwd.h)

WDIOC_GETBOOTSTATUS
This returns the status of the card that was reported at bootup.

WDIOC_GETTEMP
This returns the temperature of the card. (You can also read /dev/watchdog, which gives a temperature update every second.)

WDIOC_SETOPTIONS
This lets you set the options of the card. You can either enable or disable the card this way.

WDIOC_KEEPALIVE
This pings the card to tell it not to reset your computer.

And that's all she wrote!

—Ken Hollis (kenji@bitgate.com)

CHAPTER

FOUR

THE LINUX WATCHDOG DRIVER API

Last reviewed: 10/05/2007

Some parts of this document are copied verbatim from the sbc60xxwdt driver which is (c) Copyright 2000 Jakob Oestergaard <jakob@ostenfeld.dk>

This document describes the state of the Linux 2.4.18 kernel.

4.1 Introduction

A Watchdog Timer (WDT) is a hardware circuit that can reset the computer system in case of a software fault. You probably knew that already.

Usually a userspace daemon will notify the kernel watchdog driver via the /dev/watchdog special device file that userspace is still alive, at regular intervals. When such a notification occurs, the driver will usually tell the hardware watchdog that everything is in order, and that the watchdog should wait for yet another little while to reset the system. If userspace fails (RAM error, kernel bug, whatever), the notifications cease to occur, and the hardware watchdog will reset the system (causing a reboot) after the timeout occurs.

The Linux watchdog API is a rather ad-hoc construction and different drivers implement different, and sometimes incompatible, parts of it. This file is an attempt to document the existing usage and allow future driver writers to use it as a reference.

4.2 The simplest API

All drivers support the basic mode of operation, where the watchdog activates as soon as /dev/watchdog is opened and will reboot unless the watchdog is pinged within a certain time, this time is called the timeout or margin. The simplest way to ping the watchdog is to write some data to the device. So a very simple watchdog daemon would look like this source file: see samples/watchdog/watchdog-simple.c

A more advanced driver could for example check that a HTTP server is still responding before doing the write call to ping the watchdog.

When the device is closed, the watchdog is disabled, unless the "Magic Close" feature is supported (see below). This is not always such a good idea, since if

there is a bug in the watchdog daemon and it crashes the system will not reboot. Because of this, some of the drivers support the configuration option "Disable watchdog shutdown on close", CONFIG_WATCHDOG_NOWAYOUT. If it is set to Y when compiling the kernel, there is no way of disabling the watchdog once it has been started. So, if the watchdog daemon crashes, the system will reboot after the timeout has passed. Watchdog devices also usually support the nowayout module parameter so that this option can be controlled at runtime.

4.3 Magic Close feature

If a driver supports "Magic Close", the driver will not disable the watchdog unless a specific magic character 'V' has been sent to /dev/watchdog just before closing the file. If the userspace daemon closes the file without sending this special character, the driver will assume that the daemon (and userspace in general) died, and will stop pinging the watchdog without disabling it first. This will then cause a reboot if the watchdog is not re-opened in sufficient time.

4.4 The ioctl API

All conforming drivers also support an ioctl API.

Pinging the watchdog using an ioctl:

All drivers that have an ioctl interface support at least one ioctl, KEEPALIVE. This ioctl does exactly the same thing as a write to the watchdog device, so the main loop in the above program could be replaced with:

```
while (1) {
        ioctl(fd, WDIOC_KEEPALIVE, 0);
        sleep(10);
}
```

the argument to the ioctl is ignored.

4.5 Setting and getting the timeout

For some drivers it is possible to modify the watchdog timeout on the fly with the SETTIMEOUT ioctl, those drivers have the WDIOF_SETTIMEOUT flag set in their option field. The argument is an integer representing the timeout in seconds. The driver returns the real timeout used in the same variable, and this timeout might differ from the requested one due to limitation of the hardware:

```
int timeout = 45;
ioctl(fd, WDIOC_SETTIMEOUT, &timeout);
printf("The timeout was set to %d seconds\n", timeout);
```

This example might actually print "The timeout was set to 60 seconds" if the device has a granularity of minutes for its timeout.

Starting with the Linux 2.4.18 kernel, it is possible to query the current timeout using the GETTIMEOUT ioctl:

```
ioctl(fd, WDIOC_GETTIMEOUT, &timeout);
printf("The timeout was is %d seconds\n", timeout);
```

4.6 Pretimeouts

Some watchdog timers can be set to have a trigger go off before the actual time they will reset the system. This can be done with an NMI, interrupt, or other mechanism. This allows Linux to record useful information (like panic information and kernel coredumps) before it resets:

```
pretimeout = 10;
ioctl(fd, WDIOC_SETPRETIMEOUT, &pretimeout);
```

Note that the pretimeout is the number of seconds before the time when the timeout will go off. It is not the number of seconds until the pretimeout. So, for instance, if you set the timeout to 60 seconds and the pretimeout to 10 seconds, the pretimeout will go off in 50 seconds. Setting a pretimeout to zero disables it.

There is also a get function for getting the pretimeout:

```
ioctl(fd, WDIOC_GETPRETIMEOUT, &timeout);
printf("The pretimeout was is %d seconds\n", timeout);
```

Not all watchdog drivers will support a pretimeout.

4.7 Get the number of seconds before reboot

Some watchdog drivers have the ability to report the remaining time before the system will reboot. The WDIOC_GETTIMELEFT is the ioctl that returns the number of seconds before reboot:

```
ioctl(fd, WDIOC_GETTIMELEFT, &timeleft);
printf("The timeout was is %d seconds\n", timeleft);
```

4.8 Environmental monitoring

All watchdog drivers are required return more information about the system, some do temperature, fan and power level monitoring, some can tell you the reason for the last reboot of the system. The GETSUPPORT ioctl is available to ask what the device can do:

```
struct watchdog_info ident;
ioctl(fd, WDIOC_GETSUPPORT, &ident);
```

the fields returned in the ident struct are:

identity	a string identifying the watchdog driver
firmware_version	the firmware version of the card if available
options	a flags describing what the device supports

the options field can have the following bits set, and describes what kind of information that the GET_STATUS and GET_BOOT_STATUS ioctls can return.

WDIOF_OVERHEAT	Reset due to CPU overheat

The machine was last rebooted by the watchdog because the thermal limit was exceeded:

WDIOF_FANFAULT	Fan failed

A system fan monitored by the watchdog card has failed

WDIOF_EXTERN1	External relay 1

External monitoring relay/source 1 was triggered. Controllers intended for real world applications include external monitoring pins that will trigger a reset.

WDIOF_EXTERN2	External relay 2

External monitoring relay/source 2 was triggered

WDIOF_POWERUNDER	Power bad/power fault

The machine is showing an undervoltage status

WDIOF_CARDRESET	Card previously reset the CPU

The last reboot was caused by the watchdog card

WDIOF_POWEROVER	Power over voltage

The machine is showing an overvoltage status. Note that if one level is under and one over both bits will be set - this may seem odd but makes sense.

WDIOF_KEEPALIVEPING	Keep alive ping reply

The watchdog saw a keepalive ping since it was last queried.

WDIOF_SETTIMEOUT	Can set/get the timeout

The watchdog can do pretimeouts.

WDIOF_PRETIMEOUT	Pretimeout (in seconds), get/set

For those drivers that return any bits set in the option field, the GETSTATUS and GETBOOTSTATUS ioctls can be used to ask for the current status, and the status at the last reboot, respectively:

```
int flags;
ioctl(fd, WDIOC_GETSTATUS, &flags);

or

ioctl(fd, WDIOC_GETBOOTSTATUS, &flags);
```

Note that not all devices support these two calls, and some only support the GETBOOTSTATUS call.

Some drivers can measure the temperature using the GETTEMP ioctl. The returned value is the temperature in degrees fahrenheit:

```
int temperature;
ioctl(fd, WDIOC_GETTEMP, &temperature);
```

Finally the SETOPTIONS ioctl can be used to control some aspects of the cards operation:

```
int options = 0;
ioctl(fd, WDIOC_SETOPTIONS, &options);
```

The following options are available:

WDIOS_DISABLECARD	Turn off the watchdog timer
WDIOS_ENABLECARD	Turn on the watchdog timer
WDIOS_TEMPPANIC	Kernel panic on temperature trip

[FIXME – better explanations]

CHAPTER

FIVE

THE LINUX WATCHDOG TIMER DRIVER CORE KERNEL API

Last reviewed: 12-Feb-2013

Wim Van Sebroeck <wim@iguana.be>

5.1 Introduction

This document does not describe what a WatchDog Timer (WDT) Driver or Device is. It also does not describe the API which can be used by user space to communicate with a WatchDog Timer. If you want to know this then please read the following file: *The Linux Watchdog driver API* .

So what does this document describe? It describes the API that can be used by WatchDog Timer Drivers that want to use the WatchDog Timer Driver Core Framework. This framework provides all interfacing towards user space so that the same code does not have to be reproduced each time. This also means that a watchdog timer driver then only needs to provide the different routines (operations) that control the watchdog timer (WDT).

5.2 The API

Each watchdog timer driver that wants to use the WatchDog Timer Driver Core must #include <linux/watchdog.h> (you would have to do this anyway when writing a watchdog device driver). This include file contains following register/unregister routines:

```
extern int watchdog_register_device(struct watchdog_device *);
extern void watchdog_unregister_device(struct watchdog_device *);
```

The watchdog_register_device routine registers a watchdog timer device. The parameter of this routine is a pointer to a watchdog_device structure. This routine returns zero on success and a negative errno code for failure.

The watchdog_unregister_device routine deregisters a registered watchdog timer device. The parameter of this routine is the pointer to the registered watchdog_device structure.

The watchdog subsystem includes an registration deferral mechanism, which allows you to register an watchdog as early as you wish during the boot process.

The watchdog device structure looks like this:

```
struct watchdog_device {
      int id;
      struct device *parent;
      const struct attribute_group **groups;
      const struct watchdog_info *info;
      const struct watchdog_ops *ops;
      const struct watchdog_governor *gov;
      unsigned int bootstatus;
      unsigned int timeout;
      unsigned int pretimeout;
      unsigned int min_timeout;
      unsigned int max_timeout;
      unsigned int min_hw_heartbeat_ms;
      unsigned int max_hw_heartbeat_ms;
      struct notifier_block reboot_nb;
      struct notifier_block restart_nb;
      void *driver_data;
      struct watchdog_core_data *wd_data;
      unsigned long status;
      struct list_head deferred;
};
```

It contains following fields:

- id: set by watchdog_register_device, id 0 is special. It has both a /dev/watchdog0 cdev (dynamic major, minor 0) as well as the old /dev/watchdog miscdev. The id is set automatically when calling watchdog_register_device.
- parent: set this to the parent device (or NULL) before calling watchdog_register_device.
- groups: List of sysfs attribute groups to create when creating the watchdog device.
- info: a pointer to a watchdog_info structure. This structure gives some additional information about the watchdog timer itself. (Like it's unique name)
- ops: a pointer to the list of watchdog operations that the watchdog supports.
- gov: a pointer to the assigned watchdog device pretimeout governor or NULL.
- timeout: the watchdog timer's timeout value (in seconds). This is the time after which the system will reboot if user space does not send a heartbeat request if WDOG_ACTIVE is set.
- pretimeout: the watchdog timer's pretimeout value (in seconds).
- min_timeout: the watchdog timer's minimum timeout value (in seconds). If set, the minimum configurable value for 'timeout'.
- max_timeout: the watchdog timer's maximum timeout value (in seconds), as seen from userspace. If set, the maximum configurable value for 'timeout'.

Not used if max_hw_heartbeat_ms is non-zero.

- min_hw_heartbeat_ms: Hardware limit for minimum time between heartbeats, in milli-seconds. This value is normally 0; it should only be provided if the hardware can not tolerate lower intervals between heartbeats.
- max_hw_heartbeat_ms: Maximum hardware heartbeat, in milli-seconds. If set, the infrastructure will send heartbeats to the watchdog driver if 'timeout' is larger than max_hw_heartbeat_ms, unless WDOG_ACTIVE is set and userspace failed to send a heartbeat for at least 'timeout' seconds. max_hw_heartbeat_ms must be set if a driver does not implement the stop function.
- reboot_nb: notifier block that is registered for reboot notifications, for internal use only. If the driver calls watchdog_stop_on_reboot, watchdog core will stop the watchdog on such notifications.
- restart_nb: notifier block that is registered for machine restart, for internal use only. If a watchdog is capable of restarting the machine, it should define ops->restart. Priority can be changed through watchdog_set_restart_priority.
- bootstatus: status of the device after booting (reported with watchdog WDIOF_* status bits).
- driver_data: a pointer to the drivers private data of a watchdog device. This data should only be accessed via the watchdog_set_drvdata and watchdog_get_drvdata routines.
- wd_data: a pointer to watchdog core internal data.
- status: this field contains a number of status bits that give extra information about the status of the device (Like: is the watchdog timer running/active, or is the nowayout bit set).
- deferred: entry in wtd_deferred_reg_list which is used to register early initialized watchdogs.

The list of watchdog operations is defined as:

```
struct watchdog_ops {
      struct module *owner;
      /* mandatory operations */
      int (*start)(struct watchdog_device *);
      /* optional operations */
      int (*stop)(struct watchdog_device *);
      int (*ping)(struct watchdog_device *);
      unsigned int (*status)(struct watchdog_device *);
      int (*set_timeout)(struct watchdog_device *, unsigned int);
      int (*set_pretimeout)(struct watchdog_device *, unsigned int);
      unsigned int (*get_timeleft)(struct watchdog_device *);
      int (*restart)(struct watchdog_device *);
      long (*ioctl)(struct watchdog_device *, unsigned int,␣
 ↪unsigned long);
};
```

It is important that you first define the module owner of the watchdog timer driver'

s operations. This module owner will be used to lock the module when the watchdog is active. (This to avoid a system crash when you unload the module and /dev/watchdog is still open).

Some operations are mandatory and some are optional. The mandatory operations are:

- start: this is a pointer to the routine that starts the watchdog timer device. The routine needs a pointer to the watchdog timer device structure as a parameter. It returns zero on success or a negative errno code for failure.

Not all watchdog timer hardware supports the same functionality. That's why all other routines/operations are optional. They only need to be provided if they are supported. These optional routines/operations are:

- stop: with this routine the watchdog timer device is being stopped.

  The routine needs a pointer to the watchdog timer device structure as a parameter. It returns zero on success or a negative errno code for failure. Some watchdog timer hardware can only be started and not be stopped. A driver supporting such hardware does not have to implement the stop routine.

  If a driver has no stop function, the watchdog core will set WDOG_HW_RUNNING and start calling the driver's keepalive pings function after the watchdog device is closed.

  If a watchdog driver does not implement the stop function, it must set max_hw_heartbeat_ms.

- ping: this is the routine that sends a keepalive ping to the watchdog timer hardware.

  The routine needs a pointer to the watchdog timer device structure as a parameter. It returns zero on success or a negative errno code for failure.

  Most hardware that does not support this as a separate function uses the start function to restart the watchdog timer hardware. And that's also what the watchdog timer driver core does: to send a keepalive ping to the watchdog timer hardware it will either use the ping operation (when available) or the start operation (when the ping operation is not available).

  (Note: the WDIOC_KEEPALIVE ioctl call will only be active when the WDIOF_KEEPALIVEPING bit has been set in the option field on the watchdog's info structure).

- status: this routine checks the status of the watchdog timer device. The status of the device is reported with watchdog WDIOF_* status flags/bits.

  WDIOF_MAGICCLOSE and WDIOF_KEEPALIVEPING are reported by the watchdog core; it is not necessary to report those bits from the driver. Also, if no status function is provided by the driver, the watchdog core reports the status bits provided in the bootstatus variable of struct watchdog_device.

- set_timeout: this routine checks and changes the timeout of the watchdog timer device. It returns 0 on success, -EINVAL for "parameter out of range" and -EIO for "could not write value to the watchdog". On success this routine should set the timeout value of the watchdog_device to the achieved timeout

value (which may be different from the requested one because the watchdog does not necessarily have a 1 second resolution).

Drivers implementing max_hw_heartbeat_ms set the hardware watchdog heartbeat to the minimum of timeout and max_hw_heartbeat_ms. Those drivers set the timeout value of the watchdog_device either to the requested timeout value (if it is larger than max_hw_heartbeat_ms), or to the achieved timeout value. (Note: the WDIOF_SETTIMEOUT needs to be set in the options field of the watchdog's info structure).

If the watchdog driver does not have to perform any action but setting the watchdog_device.timeout, this callback can be omitted.

If set_timeout is not provided but, WDIOF_SETTIMEOUT is set, the watchdog infrastructure updates the timeout value of the watchdog_device internally to the requested value.

If the pretimeout feature is used (WDIOF_PRETIMEOUT), then set_timeout must also take care of checking if pretimeout is still valid and set up the timer accordingly. This can't be done in the core without races, so it is the duty of the driver.

- set_pretimeout: this routine checks and changes the pretimeout value of the watchdog. It is optional because not all watchdogs support pretimeout notification. The timeout value is not an absolute time, but the number of seconds before the actual timeout would happen. It returns 0 on success, -EINVAL for "parameter out of range" and -EIO for "could not write value to the watchdog". A value of 0 disables pretimeout notification.

  (Note: the WDIOF_PRETIMEOUT needs to be set in the options field of the watchdog's info structure).

  If the watchdog driver does not have to perform any action but setting the watchdog_device.pretimeout, this callback can be omitted. That means if set_pretimeout is not provided but WDIOF_PRETIMEOUT is set, the watchdog infrastructure updates the pretimeout value of the watchdog_device internally to the requested value.

- get_timeleft: this routines returns the time that's left before a reset.

- restart: this routine restarts the machine. It returns 0 on success or a negative errno code for failure.

- ioctl: if this routine is present then it will be called first before we do our own internal ioctl call handling. This routine should return -ENOIOCTLCMD if a command is not supported. The parameters that are passed to the ioctl call are: watchdog_device, cmd and arg.

The status bits should (preferably) be set with the set_bit and clear_bit alike bit-operations. The status bits that are defined are:

- WDOG_ACTIVE: this status bit indicates whether or not a watchdog timer device is active or not from user perspective. User space is expected to send heartbeat requests to the driver while this flag is set.

- WDOG_NO_WAY_OUT: this bit stores the nowayout setting for the watchdog. If this bit is set then the watchdog timer will not be able to stop.

- WDOG_HW_RUNNING: Set by the watchdog driver if the hardware watchdog is running. The bit must be set if the watchdog timer hardware can not be stopped. The bit may also be set if the watchdog timer is running after booting, before the watchdog device is opened. If set, the watchdog infrastructure will send keepalives to the watchdog hardware while WDOG_ACTIVE is not set. Note: when you register the watchdog timer device with this bit set, then opening /dev/watchdog will skip the start operation but send a keepalive request instead.

  To set the WDOG_NO_WAY_OUT status bit (before registering your watchdog timer device) you can either:

  - set it statically in your watchdog_device struct with

    .status = WATCHDOG_NOWAYOUT_INIT_STATUS,

    (this will set the value the same as CONFIG_WATCHDOG_NOWAYOUT) or

  - use the following helper function:

```
static inline void watchdog_set_nowayout(struct watchdog_
↪device *wdd,
                                         int nowayout)
```

Note:
The WatchDog Timer Driver Core supports the magic close feature and the nowayout feature. To use the magic close feature you must set the WDIOF_MAGICCLOSE bit in the options field of the watchdog's info structure.

The nowayout feature will overrule the magic close feature.

To get or set driver specific data the following two helper functions should be used:

```
static inline void watchdog_set_drvdata(struct watchdog_device *wdd,
                                        void *data)
static inline void *watchdog_get_drvdata(struct watchdog_device␣
↪*wdd)
```

The watchdog_set_drvdata function allows you to add driver specific data. The arguments of this function are the watchdog device where you want to add the driver specific data to and a pointer to the data itself.

The watchdog_get_drvdata function allows you to retrieve driver specific data. The argument of this function is the watchdog device where you want to retrieve data from. The function returns the pointer to the driver specific data.

To initialize the timeout field, the following function can be used:

```
extern int watchdog_init_timeout(struct watchdog_device *wdd,
                                 unsigned int timeout_parm,
                                 struct device *dev);
```

The watchdog_init_timeout function allows you to initialize the timeout field using the module timeout parameter or by retrieving the timeout-sec property from the

device tree (if the module timeout parameter is invalid). Best practice is to set the default timeout value as timeout value in the watchdog_device and then use this function to set the user "preferred" timeout value. This routine returns zero on success and a negative errno code for failure.

To disable the watchdog on reboot, the user must call the following helper:

```
static inline void watchdog_stop_on_reboot(struct watchdog_device␣
↪*wdd);
```

To disable the watchdog when unregistering the watchdog, the user must call the following helper. Note that this will only stop the watchdog if the nowayout flag is not set.

```
static inline void watchdog_stop_on_unregister(struct watchdog_
↪device *wdd);
```

To change the priority of the restart handler the following helper should be used:

```
void watchdog_set_restart_priority(struct watchdog_device *wdd, int␣
↪priority);
```

User should follow the following guidelines for setting the priority:

- 0: should be called in last resort, has limited restart capabilities
- 128: default restart handler, use if no other handler is expected to be available, and/or if restart is sufficient to restart the entire system
- 255: highest priority, will preempt all other restart handlers

To raise a pretimeout notification, the following function should be used:

```
void watchdog_notify_pretimeout(struct watchdog_device *wdd)
```

The function can be called in the interrupt context. If watchdog pretimeout governor framework (kbuild CONFIG_WATCHDOG_PRETIMEOUT_GOV symbol) is enabled, an action is taken by a preconfigured pretimeout governor preassigned to the watchdog device. If watchdog pretimeout governor framework is not enabled, watchdog_notify_pretimeout() prints a notification message to the kernel log buffer.

To set the last known HW keepalive time for a watchdog, the following function should be used:

```
int watchdog_set_last_hw_keepalive(struct watchdog_device *wdd,
                                   unsigned int last_ping_ms)
```

This function must be called immediately after watchdog registration. It sets the last known hardware heartbeat to have happened last_ping_ms before current time. Calling this is only needed if the watchdog is already running when probe is called, and the watchdog can only be pinged after the min_hw_heartbeat_ms time has passed from the last ping.

CHAPTER

SIX

WATCHDOG MODULE PARAMETERS

This file provides information on the module parameters of many of the Linux watchdog drivers. Watchdog driver parameter specs should be listed here unless the driver has its own driver-specific information file.

See Documentation/admin-guide/kernel-parameters.rst for information on providing kernel parameters for builtin drivers versus loadable modules.

---

watchdog core:

open_timeout:
Maximum time, in seconds, for which the watchdog framework will take care of pinging a running hardware watchdog until userspace opens the corresponding /dev/watchdogN device. A value of 0 means an infinite timeout. Setting this to a non-zero value can be useful to ensure that either userspace comes up properly, or the board gets reset and allows fallback logic in the bootloader to try something else.

---

acquirewdt:

wdt_stop:
Acquire WDT 'stop' io port (default 0x43)

wdt_start:
Acquire WDT 'start' io port (default 0x443)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

advantechwdt:

wdt_stop:
Advantech WDT 'stop' io port (default 0x443)

wdt_start:
Advantech WDT 'start' io port (default 0x443)

timeout:
Watchdog timeout in seconds. 1<= timeout <=63, default=60.

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

alim1535_wdt:

timeout:
Watchdog timeout in seconds. (0 < timeout < 18000, default=60

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

alim7101_wdt:

timeout:
Watchdog timeout in seconds. (1<=timeout<=3600, default=30

use_gpio:
Use the gpio watchdog (required by old cobalt boards). default=0/off/no

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

ar7_wdt:

margin:
Watchdog margin in seconds (default=60)

nowayout:
Disable watchdog shutdown on close (default=kernel config parameter)

---

armada_37xx_wdt:

timeout:
Watchdog timeout in seconds. (default=120)

nowayout:
Disable watchdog shutdown on close (default=kernel config parameter)

---

at91rm9200_wdt:

wdt_time:
Watchdog time in seconds. (default=5)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

at91sam9_wdt:

heartbeat:
Watchdog heartbeats in seconds. (default = 15)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

bcm47xx_wdt:

wdt_time:
Watchdog time in seconds. (default=30)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

coh901327_wdt:

margin:
Watchdog margin in seconds (default 60s)

---

cpu5wdt:

port:
base address of watchdog card, default is 0x91

verbose:
be verbose, default is 0 (no)

ticks:
count down ticks, default is 10000

---

cpwd:

wd0_timeout:
Default watchdog0 timeout in 1/10secs

wd1_timeout:
Default watchdog1 timeout in 1/10secs

wd2_timeout:
Default watchdog2 timeout in 1/10secs

---

da9052wdt:

timeout:
Watchdog timeout in seconds. 2<= timeout <=131, default=2.048s

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

davinci_wdt:

heartbeat:
Watchdog heartbeat period in seconds from 1 to 600, default 60

---

ebc-c384_wdt:

timeout:
Watchdog timeout in seconds. (1<=timeout<=15300, default=60)

nowayout:
Watchdog cannot be stopped once started

---

ep93xx_wdt:

nowayout:
Watchdog cannot be stopped once started

timeout:
Watchdog timeout in seconds. (1<=timeout<=3600, default=TBD)

---

eurotechwdt:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

io:
Eurotech WDT io port (default=0x3f0)

irq:
Eurotech WDT irq (default=10)

ev:
Eurotech WDT event type (default is *int*)

---

gef_wdt:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

geodewdt:

timeout:
Watchdog timeout in seconds. 1<= timeout <=131, default=60.

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

i6300esb:

heartbeat:
Watchdog heartbeat in seconds. (1<heartbeat<2046, default=30)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

iTCO_wdt:

heartbeat:
Watchdog heartbeat in seconds. (2<heartbeat<39 (TCO v1) or 613 (TCO v2), default=30)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

iTCO_vendor_support:

vendorsupport:
iTCO vendor specific support mode, default=0 (none), 1=SuperMicro Pent3, 2=SuperMicro Pent4+, 911=Broken SMI BIOS

---

ib700wdt:

timeout:
Watchdog timeout in seconds. 0<= timeout <=30, default=30.

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

ibmasr:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

imx2_wdt:

timeout:
Watchdog timeout in seconds (default 60 s)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

indydog:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

iop_wdt:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

it8712f_wdt:

margin:
Watchdog margin in seconds (default 60)

nowayout:
Disable watchdog shutdown on close (default=kernel config parameter)

---

it87_wdt:

nogameport:
Forbid the activation of game port, default=0

nocir:
Forbid the use of CIR (workaround for some buggy setups); set to 1 if

system resets despite watchdog daemon running, default=0

exclusive:
Watchdog exclusive device open, default=1

timeout:
Watchdog timeout in seconds, default=60

testmode:
Watchdog test mode (1 = no reboot), default=0

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

ixp4xx_wdt:

heartbeat:
Watchdog heartbeat in seconds (default 60s)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

machzwd:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

action:
after watchdog resets, generate: 0 = RESET(*) 1 = SMI 2 = NMI 3 = SCI

---

max63xx_wdt:

heartbeat:
Watchdog heartbeat period in seconds from 1 to 60, default 60

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

nodelay:
Force selection of a timeout setting without initial delay (max6373/74 only, default=0)

---

mixcomwd:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

mpc8xxx_wdt:

timeout:
Watchdog timeout in ticks. (0<timeout<65536, default=65535)

reset:
Watchdog Interrupt/Reset Mode. 0 = interrupt, 1 = reset

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

mv64x60_wdt:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

ni903x_wdt:

timeout:
Initial watchdog timeout in seconds (0<timeout<516, default=60)

---

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

nic7018_wdt:

timeout:
Initial watchdog timeout in seconds (0<timeout<464, default=80)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

omap_wdt:

timer_margin:
initial watchdog timeout (in seconds)

early_enable:
Watchdog is started on module insertion (default=0

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

orion_wdt:

heartbeat:
Initial watchdog heartbeat in seconds

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

pc87413_wdt:

io:
pc87413 WDT I/O port (default: io).

timeout:
Watchdog timeout in minutes (default=timeout).

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

pika_wdt:

heartbeat:
Watchdog heartbeats in seconds. (default = 15)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

pnx4008_wdt:

heartbeat:
Watchdog heartbeat period in seconds from 1 to 60, default 19

nowayout:
Set to 1 to keep watchdog running after device release

---

pnx833x_wdt:

timeout:
Watchdog timeout in Mhz. (68Mhz clock), default=2040000000 (30 seconds)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

start_enabled:
Watchdog is started on module insertion (default=1)

---

rc32434_wdt:

timeout:
Watchdog timeout value, in seconds (default=20)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

riowd:

riowd_timeout:
Watchdog timeout in minutes (default=1)

---

s3c2410_wdt:

tmr_margin:
Watchdog tmr_margin in seconds. (default=15)

tmr_atboot:
Watchdog is started at boot time if set to 1, default=0

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

soft_noboot:
Watchdog action, set to 1 to ignore reboots, 0 to reboot

debug:
Watchdog debug, set to >1 for debug, (default 0)

---

sa1100_wdt:

margin:
Watchdog margin in seconds (default 60s)

---

sb_wdog:

timeout:
Watchdog timeout in microseconds (max/default 8388607 or 8.3ish secs)

---

sbc60xxwdt:

wdt_stop:
SBC60xx WDT 'stop' io port (default 0x45)

wdt_start:
SBC60xx WDT 'start' io port (default 0x443)

timeout:
Watchdog timeout in seconds. (1<=timeout<=3600, default=30)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

sbc7240_wdt:

timeout:
Watchdog timeout in seconds. (1<=timeout<=255, default=30)

nowayout:
Disable watchdog when closing device file

---

sbc8360:

timeout:
Index into timeout table (0-63) (default=27 (60s))

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

sbc_epx_c3:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

sbc_fitpc2_wdt:

margin:
Watchdog margin in seconds (default 60s)

nowayout:
Watchdog cannot be stopped once started

---

sbsa_gwdt:

timeout:
Watchdog timeout in seconds. (default 10s)

action:
Watchdog action at the first stage timeout, set to 0 to ignore, 1 to panic. (default=0)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

sc1200wdt:

isapnp:
When set to 0 driver ISA PnP support will be disabled (default=1)

io:
io port

timeout:
range is 0-255 minutes, default is 1

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

sc520_wdt:

timeout:
Watchdog timeout in seconds. (1 <= timeout <= 3600, default=30)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

sch311x_wdt:

force_id:
Override the detected device ID

therm_trip:
Should a ThermTrip trigger the reset generator

timeout:
Watchdog timeout in seconds. 1<= timeout <=15300, default=60

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

scx200_wdt:

margin:
Watchdog margin in seconds

nowayout:
Disable watchdog shutdown on close

---

shwdt:

clock_division_ratio:
Clock division ratio. Valid ranges are from 0x5 (1.31ms) to 0x7 (5.25ms). (default=7)

heartbeat:
Watchdog heartbeat in seconds. (1 <= heartbeat <= 3600, default=30

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

smsc37b787_wdt:

timeout:
range is 1-255 units, default is 60

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

softdog:

soft_margin:
Watchdog soft_margin in seconds. (0 < soft_margin < 65536, default=60)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

soft_noboot:
Softdog action, set to 1 to ignore reboots, 0 to reboot (default=0)

---

stmp3xxx_wdt:

heartbeat:
Watchdog heartbeat period in seconds from 1 to 4194304, default 19

---

tegra_wdt:

heartbeat:
Watchdog heartbeats in seconds. (default = 120)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

ts72xx_wdt:

timeout:
Watchdog timeout in seconds. (1 <= timeout <= 8, default=8)

nowayout:
Disable watchdog shutdown on close

---

twl4030_wdt:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

txx9wdt:

timeout:
Watchdog timeout in seconds. (0<timeout<N, default=60)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

uniphier_wdt:

timeout:
Watchdog timeout in power of two seconds. (1 <= timeout <= 128, default=64)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

w83627hf_wdt:

wdt_io:
w83627hf/thf WDT io port (default 0x2E)

timeout:
Watchdog timeout in seconds. 1 <= timeout <= 255, default=60.

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

w83877f_wdt:

timeout:
Watchdog timeout in seconds. (1<=timeout<=3600, default=30)

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

w83977f_wdt:

timeout:
Watchdog timeout in seconds (15..7635), default=45)

testmode:
Watchdog testmode (1 = no reboot), default=0

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

wafer5823wdt:

timeout:
Watchdog timeout in seconds. 1 <= timeout <= 255, default=60.

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

wdt285:

soft_margin:
Watchdog timeout in seconds (default=60)

---

wdt977:

timeout:
Watchdog timeout in seconds (60..15300, default=60)

testmode:
Watchdog testmode (1 = no reboot), default=0

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

wm831x_wdt:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

wm8350_wdt:

nowayout:
Watchdog cannot be stopped once started (default=kernel config parameter)

---

sun4v_wdt:

timeout_ms:
Watchdog timeout in milliseconds 1..180000, default=60000)

nowayout:
Watchdog cannot be stopped once started

CHAPTER

SEVEN

THE LINUX WATCHDOG TIMER POWER MANAGEMENT GUIDE

Last reviewed: 17-Dec-2018

Wolfram Sang <wsa+renesas@sang-engineering.com>

7.1 Introduction

This document states rules about watchdog devices and their power management handling to ensure a uniform behaviour for Linux systems.

7.2 Ping on resume

On resume, a watchdog timer shall be reset to its selected value to give userspace enough time to resume. [1] [2]

[1] https://patchwork.kernel.org/patch/10252209/

[2] https://patchwork.kernel.org/patch/10711625/

CHAPTER

EIGHT

WDT WATCHDOG TIMER INTERFACES FOR THE LINUX OPERATING SYSTEM

Last Reviewed: 10/05/2007

Alan Cox <alan@lxorguk.ukuu.org.uk>

- ICS WDT501-P
- ICS WDT501-P (no fan tachometer)
- ICS WDT500-P

All the interfaces provide /dev/watchdog, which when open must be written to within a timeout or the machine will reboot. Each write delays the reboot time another timeout. In the case of the software watchdog the ability to reboot will depend on the state of the machines and interrupts. The hardware boards physically pull the machine down off their own onboard timers and will reboot from almost anything.

A second temperature monitoring interface is available on the WDT501P cards. This provides /dev/temperature. This is the machine internal temperature in degrees Fahrenheit. Each read returns a single byte giving the temperature.

The third interface logs kernel messages on additional alert events.

The ICS ISA-bus wdt card cannot be safely probed for. Instead you need to pass IO address and IRQ boot parameters. E.g.:

```
wdt.io=0x240 wdt.irq=11
```

Other “wdt” driver parameters are:

heartbeat	Watchdog heartbeat in seconds (default 60)
nowayout	Watchdog cannot be stopped once started (kernel build parameter)
tachometer	WDT501-P Fan Tachometer support (0=disable, default=0)
type	WDT501-P Card type (500 or 501, default=500)

8.1 Features

Reboot Timer	X	X
External Reboot	X	X
I/O Port Monitor	o	o
Temperature	X	o
Fan Speed	X	o
Power Under	X	o
Power Over	X	o
Overheat	X	o

The external event interfaces on the WDT boards are not currently supported. Minor numbers are however allocated for it.

Example Watchdog Driver:

see samples/watchdog/watchdog-simple.c

CHAPTER

NINE

CONVERTING OLD WATCHDOG DRIVERS TO THE WATCHDOG FRAMEWORK

by Wolfram Sang <wsa@kernel.org>

Before the watchdog framework came into the kernel, every driver had to implement the API on its own. Now, as the framework factored out the common components, those drivers can be lightened making it a user of the framework. This document shall guide you for this task. The necessary steps are described as well as things to look out for.

9.1 Remove the file_operations struct

Old drivers define their own file_operations for actions like open(), write(), etc… These are now handled by the framework and just call the driver when needed. So, in general, the 'file_operations' struct and assorted functions can go. Only very few driver-specific details have to be moved to other functions. Here is a overview of the functions and probably needed actions:

- open: Everything dealing with resource management (file-open checks, magic close preparations) can simply go. Device specific stuff needs to go to the driver specific start-function. Note that for some drivers, the start-function also serves as the ping-function. If that is the case and you need start/stop to be balanced (clocks!), you are better off refactoring a separate start-function.
- close: Same hints as for open apply.
- write: Can simply go, all defined behaviour is taken care of by the framework, i.e. ping on write and magic char ('V') handling.
- ioctl: While the driver is allowed to have extensions to the IOCTL interface, the most common ones are handled by the framework, supported by some assistance from the driver:

  WDIOC_GETSUPPORT:
  Returns the mandatory watchdog_info struct from the driver

  WDIOC_GETSTATUS:
  Needs the status-callback defined, otherwise returns 0

  WDIOC_GETBOOTSTATUS:
  Needs the bootstatus member properly set. Make sure it is 0 if you don't have further support!

WDIOC_SETOPTIONS:
No preparations needed

WDIOC_KEEPALIVE:
If wanted, options in watchdog_info need to have WDIOF_KEEPALIVEPING set

WDIOC_SETTIMEOUT:
Options in watchdog_info need to have WDIOF_SETTIMEOUT set and a set_timeout-callback has to be defined. The core will also do limit-checking, if min_timeout and max_timeout in the watchdog device are set. All is optional.

WDIOC_GETTIMEOUT:
No preparations needed

WDIOC_GETTIMELEFT:
It needs get_timeleft() callback to be defined. Otherwise it will return EOPNOTSUPP

Other IOCTLs can be served using the ioctl-callback. Note that this is mainly intended for porting old drivers; new drivers should not invent private IOCTLs. Private IOCTLs are processed first. When the callback returns with -ENOIOCTLCMD, the IOCTLs of the framework will be tried, too. Any other error is directly given to the user.

Example conversion:

```
-static const struct file_operations s3c2410wdt_fops = {
-       .owner          = THIS_MODULE,
-       .llseek         = no_llseek,
-       .write          = s3c2410wdt_write,
-       .unlocked_ioctl = s3c2410wdt_ioctl,
-       .open           = s3c2410wdt_open,
-       .release        = s3c2410wdt_release,
-};
```

Check the functions for device-specific stuff and keep it for later refactoring. The rest can go.

9.2 Remove the miscdevice

Since the file_operations are gone now, you can also remove the 'struct miscdevice'. The framework will create it on watchdog_dev_register() called by watchdog_register_device():

```
-static struct miscdevice s3c2410wdt_miscdev = {
-       .minor          = WATCHDOG_MINOR,
-       .name           = "watchdog",
-       .fops           = &s3c2410wdt_fops,
-};
```

9.3 Remove obsolete includes and defines

Because of the simplifications, a few defines are probably unused now. Remove them. Includes can be removed, too. For example:

```
- #include <linux/fs.h>
- #include <linux/miscdevice.h> (if MODULE_ALIAS_MISCDEV is not␣
↪used)
- #include <linux/uaccess.h> (if no custom IOCTLs are used)
```

9.4 Add the watchdog operations

All possible callbacks are defined in 'struct watchdog_ops'. You can find it explained in 'watchdog-kernel-api.txt' in this directory. start() and owner must be set, the rest are optional. You will easily find corresponding functions in the old driver. Note that you will now get a pointer to the watchdog_device as a parameter to these functions, so you probably have to change the function header. Other changes are most likely not needed, because here simply happens the direct hardware access. If you have device-specific code left from the above steps, it should be refactored into these callbacks.

Here is a simple example:

```
+static struct watchdog_ops s3c2410wdt_ops = {
+       .owner = THIS_MODULE,
+       .start = s3c2410wdt_start,
+       .stop = s3c2410wdt_stop,
+       .ping = s3c2410wdt_keepalive,
+       .set_timeout = s3c2410wdt_set_heartbeat,
+};
```

A typical function-header change looks like:

```
-static void s3c2410wdt_keepalive(void)
+static int s3c2410wdt_keepalive(struct watchdog_device *wdd)
 {
...
+
+       return 0;
 }

...

-       s3c2410wdt_keepalive();
+       s3c2410wdt_keepalive(&s3c2410_wdd);
```

9.5 Add the watchdog device

Now we need to create a 'struct watchdog_device' and populate it with the necessary information for the framework. The struct is also explained in detail in 'watchdog-kernel-api.txt' in this directory. We pass it the mandatory watchdog_info struct and the newly created watchdog_ops. Often, old drivers have their own record-keeping for things like bootstatus and timeout using static variables. Those have to be converted to use the members in watchdog_device. Note that the timeout values are unsigned int. Some drivers use signed int, so this has to be converted, too.

Here is a simple example for a watchdog device:

```
+static struct watchdog_device s3c2410_wdd = {
+       .info = &s3c2410_wdt_ident,
+       .ops = &s3c2410wdt_ops,
+};
```

9.6 Handle the 'nowayout' feature

A few drivers use nowayout statically, i.e. there is no module parameter for it and only CONFIG_WATCHDOG_NOWAYOUT determines if the feature is going to be used. This needs to be converted by initializing the status variable of the watchdog_device like this:

```
.status = WATCHDOG_NOWAYOUT_INIT_STATUS,
```

Most drivers, however, also allow runtime configuration of nowayout, usually by adding a module parameter. The conversion for this would be something like:

```
watchdog_set_nowayout(&s3c2410_wdd, nowayout);
```

The module parameter itself needs to stay, everything else related to nowayout can go, though. This will likely be some code in open(), close() or write().

9.7 Register the watchdog device

Replace misc_register(&miscdev) with watchdog_register_device(&watchdog_dev). Make sure the return value gets checked and the error message, if present, still fits. Also convert the unregister case:

```
-       ret = misc_register(&s3c2410wdt_miscdev);
+       ret = watchdog_register_device(&s3c2410_wdd);

...

-       misc_deregister(&s3c2410wdt_miscdev);
+       watchdog_unregister_device(&s3c2410_wdd);
```

9.8 Update the Kconfig-entry

The entry for the driver now needs to select WATCHDOG_CORE:

- select WATCHDOG_CORE

9.9 Create a patch and send it to upstream

Make sure you understood Documentation/process/submitting-patches.rst and send your patch to linux-watchdog@vger.kernel.org. We are looking forward to it :)