The Linux WatchDog Timer Driver Core kernel API

Last reviewed: 12-Feb-2013

Wim Van Sebroeck <wim@iguana.be>

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: Documentation/watchdog/watchdog-api.rst .

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).

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.