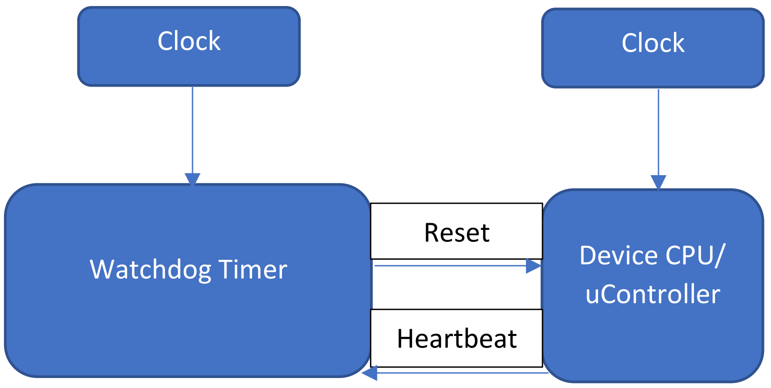
Simple Watchdog Architecture

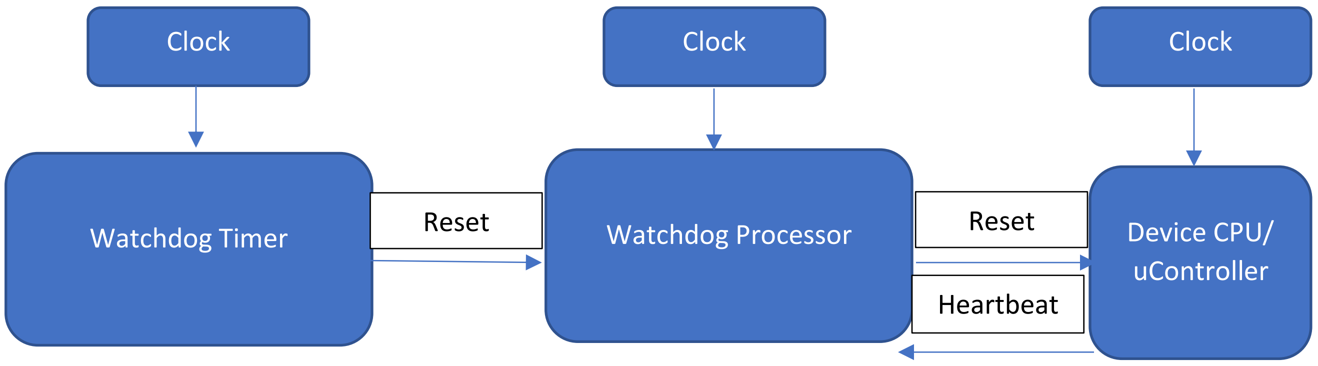


Core watchdog functionality:

1. Watchdog Timer is configured to periodically monitors the device's heartbeat signal.
2. If the heartbeat signal is not received within the predefined timeout interval, the device shall be reset.
3. See https://github.com/torvalds/linux/blob/master/samples/watchdog/watchdog-simple.c for a reference implementation of this simple architecture. /dev/watchdog is the Linux device interface to a hardware watchdog timer. See https://linux.die.net/man/8/watchdog
4. Watchdog daemon is configured via watchdog.conf https://linux.die.net/man/5/watchdog.conf

---

Typical Watchdog Architecture for Safety Critical Applications



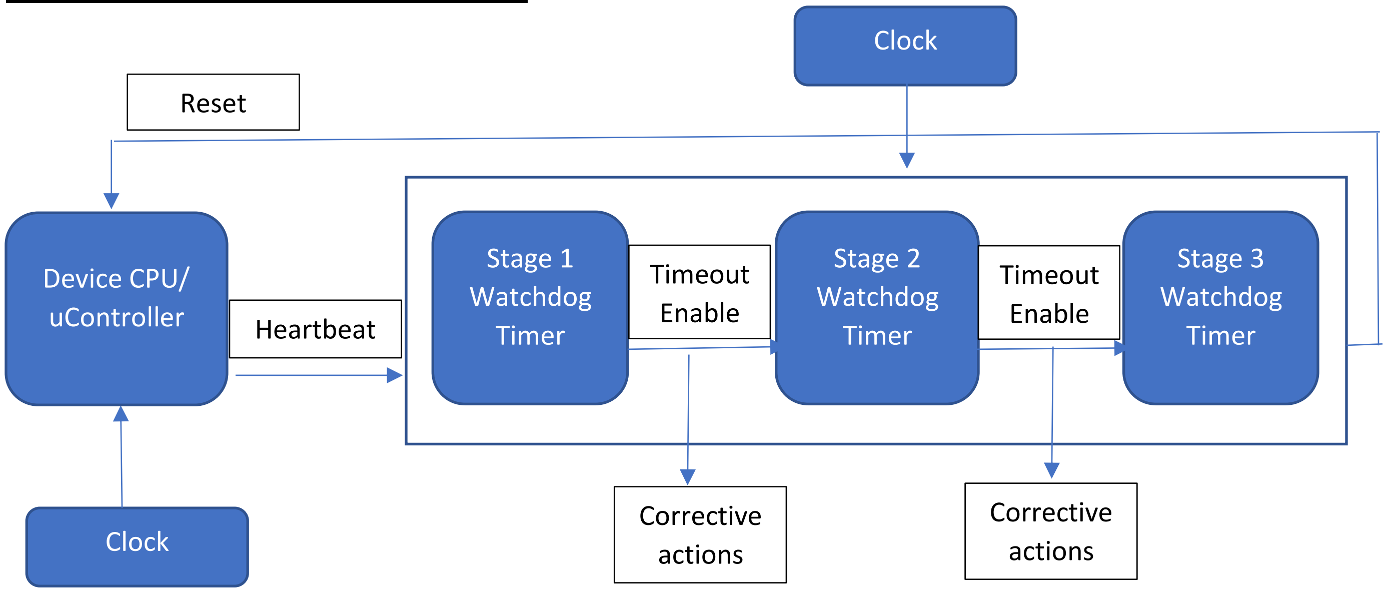
Typical interactions:

---

1. Watchdog Timer is responsible to periodically reset the watchdog processor, to ensure its correct functionality. The daemon may be coordinated with other user space applications, and accordingly can determine if the watchdog process shall be reset to reflect system health.
2. Watchdog Processor is responsible to monitor the device heartbeat. If a sign of life is not received within the predefined time, the device will be reset.

3. Watchdog Processor may monitor additional device communication bus(es) for reset requests (not depicted in this simple diagram).
4. The Watchdog Process may include an additional internal Watchdog Timer to monitor for flow control errors.

Upgraded Watchdog Architecture



1. Different timeout values may be set for each watchdog timer stage. A "pretimeout" is a trigger (e.g., NMI) which is enabled before the actual reset time.
2. Possible corrective actions:
   - Collect (via sysfs or watchdog ioctl) system information such as fault leading to reset (fan failed or CPU overhead)
   - Collect information to support investigation of failure root cause (e.g., memory dumps; fault register contents; system logs)
   - Run trace utilities
   - Interact with external controllers to determine if there is a redundant system which can take over for failed hardware, then restoring the device to normal operation mode
   - Interrupts: maskable (MI), non-maskable (NMI)
   - Graceful system shutdown (e.g., bring device / vehicle to safe state before shutdown)

Watchdog Architecture in Multitasking Environment – TBD, currently out of scope

---

Watchdog Architecture Analysis

1. In a specific use case, the system architect is responsible to define timeout intervals, corrective actions, and other system features so that the watchdog reset is safe. Safety qualification of the various watchdog elements provides the infrastructure for a safe implementation.
2. System features to be qualified – TBD:

a) watchdog daemon which is responsible to ensure the watchdog (hardware or software) is working properly. The watchdog daemon performs the following tasks:
   - Opens /dev/watchdog and "kicks" it by writing to it periodically.
   - After a timeout in which there is no write to /dev/watchdog, a hardware watchdog will lead to a system reset. Whereas a software watchdog may initiate corrective reactions.
   - Checks system status – e.g., system data structures; free/accessible memory; average workload; network Tx/Rx; ping ETH port; high temperature.
   - Execute system test and/or repair utilities.
   - Perform hard or soft (controlled, graceful) reboot.

b) watchdog.conf and support for all configuration options. Watchdog kernel configurations https://github.com/torvalds/linux/blob/master/drivers/watchdog/Kconfig For example,
   - If CONFIG_WATCHDOG_NOWAYOUT is enabled, this will disable watchdog shutdown if the /dev/watchdog is closed (in which case the watchdog cannot be stopped once it has been started).
   - If CONFIG_WATCHDOG_PRETIMEOUT_GOV_PANIC is enabled, on a watchdog pretimeout event the kernel will be put into panic state.

c) Device registration infrastructure, defined in https://github.com/torvalds/linux/blob/master/drivers/watchdog/watchdog_core.c:
   - Deferred/regular device registration
   - Validate min/max timeout values
   - Initialize timeout field, using all possible setting sources
   - Notify of reboot
   - Notify of restart
   - Update priority of the restart handler
   - Register device (support both mandatory and optional parameters)
   - Unregister device
   - Support deferred registration (storing in deferred list)
   - Initialize watchdog
   - Close watchdog

d) Generic code used by all watchdog timer drivers - https://github.com/torvalds/linux/blob/master/drivers/watchdog/watchdog_dev.c
   - Define worker thread(s) to generate heartbeat requests if userspace application requests a longer timeout than the hardware can handle.
   - Align watchdog timeout to the most recent ping from user space.
   - Update worker thread timeout
   - Watchdog ping ("kick")

- Start watchdog if the caller holds the lock on the watchdog (for multitasking environment)
- Stop watchdog if it is still active, ‘NOWAYOUT’ feature is not set, and the caller holds the lock on the watchdog.
- Return watchdog status
- Set watchdog timeout
- Set watchdog pretimeout (in case of a multistage timer)
- Return time left before a reboot, caller must hold the lock on the watchdog (for multitasking environment)
- Output watchdog features in sysfs
- Set watchdog features (e.g., NOWAYOUT)
- Support watchdog ioctl operation (via standard Linux VFS subsystem)
- Open /dev/watchdog = watchdog device
- Close watchdog device (called when /dev/watchdog is closed).
- Register watchdog character device (including support for legacy /dev/watchdog node as a miscdevice).
- Unregister watchdog character device
- Register /dev/watchdog
- Unregister /dev/watchdog
- Set last HW keepalive time for the watchdog device (must be called immediately after watchdog registration)
- Allocate a range of chardev nodes to be used for watchdog devices

3. Potential failure modes – TBD:
    - Watchdog fails to “bark” on timeout or fault.
    - Watchdog “barks” without timeout or fault.
    - Device is reset and skips defined fail state for corrective action.
    - Watchdog “barks” but timeout interval is inappropriately set for the given use case. ** Integrator
    - Timeout setting has been corrupted.
    - In a multitasking environment, the watchdog timer is not properly locked for correct synchronization between the tasks. “Kicking” the watchdog is not mutually exclusive. ** Failures of locking infrastructure, as well as integrator responsibility – out of scope.
    - In a multitasking environment, software watchdog does not handle expected software failures such as infinite loops, or deadlocks between two or more tasks - ** Failures of locking infrastructure, as well as integrator responsibility – out of scope.
    - In a multitasking environment, task priorities are not managed so that the watchdog cannot avoid livelock by higher priority task(s) - ** Failures of locking infrastructure, as well as integrator responsibility – out of scope.
    - Failure in selftests - ?? (test of basic watchdog functionality)