Legal Disclaimer

By using this document, in addition to any agreements you have with Intel, you accept the terms set forth below.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm.

Intel, Intel Quark, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

Wind River is a trademark of Wind River Systems, Inc.

*Other names and brands may be claimed as the property of others.

Copyright © 2015, Intel Corporation. All rights reserved.
# Contents

## 1 Introduction

1.1 References

## 2 Watchdog Timer Identification and Driver Installation

2.1 Watchdog Timer Identification

2.2 Watchdog Timer Driver Installation

2.3 The Watchdog Timer Daemon

2.4 Custom Watchdog Test Code

## 3 Custom Watchdog Timer Code

3.1 IOCTL Calls

3.2 Timeout Reset

3.3 Pinging the Watchdog Timer

3.4 Changing the Watchdog Timeout Interval

## 4 Advanced Watchdog Timer Calls

4.1 Advanced IOCTL Capabilities and Magic Close

4.2 Using Watchdog Timer for Debugging Information

4.3 Preventing Watchdog Timer Disable

## 5 Putting the Custom Watchdog Daemon Together

5.1 Code That Uses Watchdog IOCTL Calls

5.2 Code without Watchdog IOCTL Calls

## Figures

- Figure 1. Watchdog Configuration File
- Figure 2. Watchdog Timer Daemon in Log
- Figure 3. Timeout Reset Output
- Figure 4. Watchdog Timer Ping Output
- Figure 5. Changing Timeout Interval Output
- Figure 6. Using watchdog_info Output
Tables

Table 1. Watchdog Timer Types ............................................................................................................................................... 7
Table 2. IOCTL Calls .................................................................................................................................................................. 14
Table 3. Struct Contents for watchdog_info ..................................................................................................................... 21
Table 4. Watchdog Timer IOCTL Calls ................................................................................................................................ 24
Table 5. Watchdog Debug Register Bitmasks .................................................................................................................. 25

Sample Code

Source Code 1. Timeout Reset ............................................................................................................................................... 14
Source Code 2. Pinging Watchdog Timer .......................................................................................................................... 16
Source Code 3. Changing Timeout Interval ...................................................................................................................... 19
Source Code 4. Using watchdog_info .................................................................................................................................. 21
Source Code 5. Reading Watchdog Debug Register ..................................................................................................... 25
Source Code 7. Watchdog Test Code Template without Watchdog IOCTL Calls .............................................................. 29
## Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>August 2015</td>
<td>002</td>
<td>Graphics and code samples replaced with new versions for clarity (colored code sample text)</td>
</tr>
<tr>
<td>July 2015</td>
<td>001</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
1 Introduction

A watchdog timer is a hardware monitoring tool that prompts system recovery after a failure, which is detected by a simple pinging mechanism. When the watchdog timer is not pinged after a certain amount of time, it saves system information to aid with debugging and then attempts to correct the problem by rebooting the system.

The watchdog timer can help with recovery from problems including but not limited to:

- Problems with operating system task scheduling
- Deadlocks from faulty interrupt handling
- Faulty interrupt disabling or masking
- Incorrect hardware configurations or hardware errors
- General software failure

Without a watchdog timer, undetected system malfunctions in IoT gateways could result in the loss of an entire system, expensive damages, and/or high recovery costs.

1.1 References

<table>
<thead>
<tr>
<th>Document</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>watchdog.conf man page</td>
<td><a href="http://linux.die.net/man/5/watchdog.conf">http://linux.die.net/man/5/watchdog.conf</a></td>
</tr>
</tbody>
</table>
2 Watchdog Timer Identification and Driver Installation

2.1 Watchdog Timer Identification

It is important to understand the differences between the three types of watchdog timers described in the following table. These are presented in the order of preference from best to worst:

### Table 1. Watchdog Timer Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Conceptual Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Watchdog Timer</td>
<td>An external or “system level” watchdog timer. This is a standalone circuit with its own clock source and power source, external to the SoC (System on Chip). This type of watchdog timer is the most robust in terms of ability to properly reset the system during a system error. External watchdog timers are strongly recommended for IoT applications that are safety-critical and/or mission-critical.</td>
<td><img src="image" alt="Conceptual Diagram" /></td>
</tr>
<tr>
<td>Internal Watchdog Timer</td>
<td>An internal watchdog timer is embedded into the SoC. Internal watchdog timers usually use the internal SoC’s clock. Internal watchdog timers are used in system-on-chip devices (SoCs). While more cost- and space-efficient, internal watchdog timer registers and clocks are embedded on the SoC and are vulnerable to being disabled by hardware errors.</td>
<td><img src="image" alt="Conceptual Diagram" /></td>
</tr>
</tbody>
</table>
See the system hardware specifications, datasheets, or similar documents from your system vendor to obtain the default watchdog timer implementation in your system and information about installing an external watchdog timer.

### 2.2 Watchdog Timer Driver Installation

For any Wind River® Linux-based Intel® IOT Gateway, if the watchdog driver is installed, the `/dev/watchdog` device node file should exist on the Gateway.

To ascertain the existence of this file, use the `find` command:

```
Find /dev/watchdog
```

If the file exists, then `/dev/watchdog` displays on the screen.

If this file is not present `No such file or directory` is displayed. In this case, see the system specifications to identify the watchdog timer hardware and work with the system vendor to install the appropriate watchdog driver.

### 2.3 The Watchdog Timer Daemon

All watchdog timer drivers come with a watchdog daemon pre-installed. A daemon is an independent process that runs in the background. In its most basic form, the daemon pings the watchdog timer device within an infinite loop. The watchdog keeps the system alive as long as it is responsive.

In addition to testing for a system stall, the daemon can perform a reboot under one of three conditions:

- User-specified system temperature threshold surpassed
- User-specified system load threshold surpassed
- User-specified IP address does not respond to a ping
These capabilities can be toggled in the `watchdog.conf` file located in the `/etc/watchdog.conf` directory.

By uncommenting lines and changing variable values, the watchdog daemon's behavior can be easily changed. A comprehensive list of options and their purposes are available on the Linux man pages under `watchdog.conf`. See [http://linux.die.net/man/5/watchdog.conf](http://linux.die.net/man/5/watchdog.conf)

An example `watchdog.conf` file is shown below. In this example the lines to uncomment are boxed in blue.
Figure 1. Watchdog Configuration File

```bash
root@WR-IntelligentDevice:~# vim /etc/watchdog.conf
# ping = 172.31.14.1
# ping = 172.26.1.255
# interface = eth0
# file = /var/log/messages
# change = 1407

# Uncomment to enable test. Setting one of these values to '0'
# These values will hopefully never reboot your machine during
# (if your machine is really hung, the loadavg will go much
max-load-1 = 50
max-load-5 = 0
max-load-15 = 0

# Note that this is the number of pages!
# To get the real size, check how large the pagesize is on your
# min-memory = 1
# repair-binary =
# repair-timeout =
# test-binary =
# test-timeout =

watchdog-device = /dev/watchdog

# Default values are compiled into the binary
# temperature-device =
# max-temperature = 120
# Default values are compiled into the binary
# admin = root
# interval = 1
# logtick = 1
# log-dir = /var/log/watchdog

watchdog-timeout = 5

# This greatly decreases the chance that watchdog won't be seen
# your machine is really loaded
realtime = yes
priority = 1

# Check if syslogd is still running by enabling the following
# pidfile = /var/run/syslogd.pid
```
A watchdog daemon that uses this configuration file resets the system if the average load over one minute exceeds 50% of max CPU load. This is indicated by max-load lines.

- The watchdog-device variable is set to the watchdog device file corresponding to the watchdog timer driver as a way to let the daemon know which device to ping.
- The timeout interval, watchdog-timeout (the reboot countdown), is set to 5 seconds.
- The watchdog timer daemon is given real-time priority by realtime and priority.

Verify that the watchdog daemon has started automatically by using the UNIX grep command:

```
grep watchdog /var/log/syslog
```

The response is a display similar to what is shown below in Figure 2. If the daemon was not started, no messages would be displayed.

**Figure 2. Watchdog Timer Daemon in Log**

To manually start the watchdog daemon, use the terminal command:

```
watchdog -c /etc/watchdog.conf
```

-c indicates that the daemon should be started with the indicated configuration file (/etc/watchdog.conf). If the daemon is successfully started, there will be no terminal output.

There are many ways to automatically start the watchdog daemon. Two of these are:

- Use the update-rc.d command with an appropriate init script.
- Edit the rc.local file located in /etc/rc.local and insert the command `watchdog -c /etc/watchdog.conf` above the `exit 0` line.
Caution: Manually test the watchdog daemon before being allowing it to start automatically. A watchdog daemon that does not behave as intended can cause the system to reboot endlessly.

2.4 Custom Watchdog Test Code

The `watchdog.conf` file allows custom watchdog timer test code by setting the `test-binary` variable in `watchdog.conf` to the location of a piece of executable code.

Custom watchdog daemons can use the pinging criteria of your choice. Such test code is advantageous in applications where proper data transmission, sensor readings, and/or hardware connectivity are crucial for proper system functioning. By using custom watchdog timer code, the watchdog daemon can prevent the watchdog timer from being pinged until the specified test criteria are passed.

The daemon can be programmed to take different corrective actions if the watchdog timer approaches a timeout reset. This allows fine control over the data logged immediately before the watchdog timer expires.

If the standard watchdog daemon is sufficient for your application, skip to Section 4.3, Preventing Watchdog Timer Disable. Otherwise, continue to the next section learn how to create custom watchdog timer code and insert it into the watchdog timer daemon.
The watchdog timer code examples in this document were performed on an Intel® IoT Gateway DK100 platform running Linux* version 3.4.91, Wind River 5.0.1.26 with standard Intelligent Device Platform XT 2.0.4.26. This Gateway is based on Intel® Quark™ SOC architecture. The Gateway was accessed through a serial connection via PuTTY* on Microsoft Windows* 8.1, 64-bit.

The configuration methods and code also apply to Intel® Atom™-based gateways when the driver has been integrated and enabled.

The example code was created in C and was cross compiled using the Wind River Toolchain for Intel® Quark™ systems.

These examples demonstrate accessing and configuring a watchdog timer so that custom watchdog timer test code can be understood and easily created. These examples are performed outside of a daemon (user-space) for clarity.

**Caution:** Prematurely terminating the example code in this chapter will cause the watchdog timer to keep running and will eventually cause a system reset.

Watchdog timers are enabled as soon as the file at `/dev/watchdog` is opened. Watchdog timers must be disabled by writing the terminate character \( V \) to the `/dev/watchdog` file and then closing it.

### 3.1 IOCTL Calls

**NOTE:** Although unlikely, a self-installed external watchdog timer might not have IOCTL capabilities. In this case, the code for this chapter will not work. See Section 4.3, Preventing Watchdog Timer Disable and then continue to Chapter 5, Putting the Custom Watchdog Daemon Together.

The example code used in this document uses IOCTL (input/output control) calls which are standard API functions provided by the open source watchdog driver. IOCTL calls enable userspace applications to communicate with device drivers. These communicate with the device.

Watchdog timer IOCTL calls take the following parameters:

- The file descriptor of the watchdog device node (`/dev/watchdog`).
- A specific request ID. These IDs are constant 32-bit values in the `linux/watchdog.h` file which comes with the watchdog driver.
- A pointer to a variable that gets written to or read from as appropriate for the specified request ID.

The following table shows the IOCTL calls that are used in this chapter's example code:
Table 2. IOCTL Calls

<table>
<thead>
<tr>
<th>IOCTL Call</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioctl(fd, WDIOC_GETTIMEOUT, &amp;interval)</td>
<td>Read the watchdog timer’s timeout interval value and write it to the “interval” variable. This value is an unsigned integer presenting time units of seconds.</td>
</tr>
<tr>
<td>ioctl(fd, WDIOC_SETTIMEOUT, &amp;desired_interval)</td>
<td>Overwrite the current watchdog timer’s timeout interval value with the value stored in the “desired_interval” variable. Just as before, this value is an unsigned integer representing seconds.</td>
</tr>
</tbody>
</table>

### 3.2 Timeout Reset

All watchdog timers have a timeout interval value. This value represents the number of seconds that the watchdog timer will wait to be pinged before forcing a system reboot.

The following source code shows how to enable the watchdog timer, read its default timeout interval. Then it demonstrate a system reset resulting from not pinging the watchdog hardware. If this code resets your system, the watchdog timer driver has been successfully installed.

#### Source Code 1. Timeout Reset

```c
#include <linux/watchdog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

// Global variable to hold the initial time of program execution
static time_t initial_time;

// Returns the elapsed time since this program began execution
double get_elapsed_time()
{
    time_t current_time;
    time(&current_time);
    return difftime(current_time, initial_time);
}

//Main function
int main(void)
{
    //Record initial time
    time(&initial_time);

    // Open watchdog file to turn on watchdog timer
    printf("Opening watchdog file at /dev/watchdog ... ");
    int fd = open("/dev/watchdog", O_WRONLY);
    int ret = 0;
    if (fd == -1) {
        // Error handling
    }

    // Read the timeout interval
    printf("Read timeout interval: ");
    printf("%d", get_elapsed_time());
    printf(" seconds. ");
    return 0;
}
```

...
```c
perror("watchdog file could not be opened");
exit(EXIT_FAILURE);

// Read the timeout interval from the watchdog timer hardware
unsigned int interval;
printf("Reading watchdog timer reset interval ... ");
if (ioctl(fd, WDIOC_GETTIMEOUT, &interval) == -1) {
    perror("watchdog timer interval could not be read\n");
    exit(EXIT_FAILURE);
}
printf("SUCCESS!\n");

// Count off until reset
printf("The timeout interval is %u s. System should reset in
" "this time since we are not pinging the WDT ...
", interval);
while (get_elapsed_time() < 2 * interval)
{
    printf("Time elapsed: %.f s\n", get_elapsed_time() );
    sleep(1);
}

// Code should not get here since the system should reset
printf("Watchdog timer not working: system should have restarted.\n");
return -1;
```

The screen looks like this immediately before the system reset:

**Figure 3. Timeout Reset Output**
3.3 Pinging the Watchdog Timer

The following example code builds off the above source code. This time it will ping the
watchdog timer every second. No system reset occurs. Once the timeout interval
duration plus 5 seconds has passed, the watchdog timer is disabled and the program
exits normally.

There are two ways to ping the watchdog timer:

- Writing to the /dev/watchdog file. This method is used below and writes the
  null character \0.
- Use the IOCTL call Keep Alive which writes a value to /dev/watchdog. The
  code used here uses the non-IOCTL pinging method for portability.

**Caution:** Early termination results in the watchdog timer counting down and resetting your
system. Let the code run to completion so that the watchdog timer can be disabled.

Source Code 2. Pinging Watchdog Timer

```c
#include <linux/watchdog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

// Global variable to hold initial time of program execution
static time_t initial_time;

// Returns the elapsed time since this program began execution
double get_elapsed_time()
{
    time_t current_time;
    time(&current_time);
    return difftime(current_time, initial_time);
}

// Prevent watchdog timer from restarting the system
void ping_watchdog(fd)
{
    write(fd, "\0", 1);
    printf("/dev/watchdog written to (timer has been pinged)\n");
}

// Main function
int main(void)
{
    // Record initial time
    time(&initial_time);

    // Open the file /dev/watchdog to turn on watchdog timer
    printf("Opening watchdog file at /dev/watchdog ...");
    int fd = open("/dev/watchdog", O_WRONLY);
    int ret = 0;
    if (fd == -1) {
```
```c
perror("watchdog file could not be opened\n");
exit(EXIT_FAILURE);

} printf("SUCCESS!\n");

// Read the timeout interval from the watchdog timer hardware
unsigned int timeout_interval;
printf("Reading watchdog timer reset interval ... ");
if (ioctl(fd, WDIOC_GETTIMEOUT, &timeout_interval) == -1) {
    perror("watchdog timer interval could not be read\n");
    exit(EXIT_FAILURE);
}
printf("SUCCESS!\n");
unsigned int test_time = timeout_interval + 5;

// Ping the watchdog timer hardware every second to prevent reset
printf("Begin watchdog timer ping test. If the watchdog timer is not pinged\n" "for %u seconds, the system will reset.\n", timeout_interval);
while (get_elapsed_time() < test_time) {
    sleep(1); // Wait 1 second
    ping_watchdog(fd);
    printf("Time elapsed: %.f seconds\n\n", get_elapsed_time() );
}

// Write "magic" terminate character and close /dev/watchdog to turn off watchdog
printf("TEST SUCCESS! Now disabling watchdog timer.\n");
write(fd, "V", 1);
close(fd);
return 0;
```
3.4 Changing the Watchdog Timeout Interval

The source code in this section demonstrates how to specify the watchdog timer’s timeout interval. This means how long the device can go without being pinged before forcing a system reset.

This program takes the desired timeout interval as interactive input and then sets the watchdog timeout interval. The updated timeout interval is then printed for verification. The desired timeout value must be an unsigned integer that represents time in seconds. The default timeout value is usually 60. The minimum is usually 1, and the maximum is usually a few hundred seconds.

Although unlikely, old watchdog timer devices might not support timeout interval configuration. If running the code below leads to an IOCTL error, run the sample code in Chapter 4 to verify absence of the SETTIMEOUT capability.
Caution: Early termination of this code or delayed user input will result in the watchdog timer resetting your system. Enter your new timeout interval in a reasonable amount of time so that the watchdog timer can be disabled.

Source Code 3. Changing Timeout Interval

```c
#include <linux/watchdog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

int main()
{
    // Open the file /dev/watchdog in order to configure watchdog timer
    printf("Opening watchdog file at /dev/watchdog ...");
    int fd = open("/dev/watchdog", O_WRONLY);
    int ret = 0;
    if (fd == -1) {
        perror("watchdog file could not be opened\n");
        exit(EXIT_FAILURE);
    }
    printf("SUCCESS!\n");

    // Read the timeout interval from the watchdog timer hardware
    unsigned int timeout_interval;
    printf("Reading watchdog timer reset interval ...");
    if (ioctl(fd, WDIOC_GETTIMEOUT, &timeout_interval) == -1) {
        perror("watchdog timer interval could not be read\n");
        exit(EXIT_FAILURE);
    }
    printf("SUCCESS!\n\n");

    // Display old timeout interval and set the new desired timeout interval
    unsigned int desired_interval;
    printf("Old timeout interval: %d\n" "Enter new timeout interval now\n\n", timeout_interval);
    scanf("%u", &desired_interval);
    if (ioctl(fd, WDIOC_SETTIMEOUT, &desired_interval) < 0) {
        perror("watchdog timeout interval could not be set\n");
        exit(EXIT_FAILURE);
    }

    // Show new timeout interval and turn off watchdog timer
    if (ioctl(fd, WDIOC_GETTIMEOUT, &timeout_interval) == -1) {
        perror("new watchdog timer interval could not be read\n");
        exit(EXIT_FAILURE);
    }
    printf("Watchdog timeout interval is now %d seconds!\n" "Disabling watchdog timer.\n", timeout_interval);
    write(fd, "V", 1);
    close(fd);
    return 0;
}
```
Figure 5. Changing Timeout Interval Output

```
root@WR-IntelligentDevice:# ./configure
Opening watchdog file at /dev/watchdog ... SUCCESS!
Reading watchdog timer reset interval ... SUCCESS!
Old timeout interval: 3
Enter new timeout interval now
5
Watchdog timeout interval is now 5 seconds!
Disabling watchdog timer.
root@WR-IntelligentDevice:#
```
Advanced Watchdog Timer Calls

The previous chapter demonstrated a simple mechanism in which a watchdog timer was enabled and then a NULL character was echoed after opening the watchdog device file. This kept the watchdog timer alive.

This chapter introduces code that increases the robustness of the watchdog daemon. This chapter also shows how to read from a debugging register on the watchdog timer device to get information about the last system reset.

The code used in this chapter uses the IOCTL call:
```
ioctl(fd, WUDIOC_GETSUPPORT, &my_info)
```
This call return information packaged into a `watchdog_info` struct, arbitrarily named `my_info`. This information can be used to gain information about the capabilities of the watchdog device installed.

The contents of this `watchdog_info` struct are:

<table>
<thead>
<tr>
<th>Struct Variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>my_info.identity</code></td>
<td>Contains the watchdog driver name represented as a string.</td>
</tr>
<tr>
<td><code>my_info.firmware_version</code></td>
<td>Contains the firmware version represented as an unsigned integer.</td>
</tr>
<tr>
<td><code>my_info.options</code></td>
<td>Contains a 32 bit value where each bit indicates if the watchdog driver supports a certain capability.</td>
</tr>
</tbody>
</table>

Examine and run the source code in this chapter to understand `my_info.options`. Bits that represent the different driver capabilities are read by performing a bitwise AND operation on the 32-bit `options` value with watchdog driver bitmasks. These bitmasks come with the `linux/watchdog.h` header file and cover all bits that represent driver capabilities.

These bitmasks are used in the code provided in this chapter. Older drivers might recognize only a subset of these capabilities. Therefore, some of the bitmasks used might not exist in your header file. Replace such bitmasks with 0 (zero).

Source Code 4. Using watchdog_info
```
#include <linux/watchdog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

// Global watchdog_info struct (see chapter 4: Preface)
struct watchdog_info my_info;
```
// Perform bitwise AND using watchdog bitmasks

unsigned int test_support(unsigned int bitmask)
{
    return (my_info.options & bitmask);
}

// Main function
int main(void)
{
    // Open the file /dev/watchdog to read watchdog timer properties
    printf("Opening watchdog file at /dev/watchdog ... ");
    int fd = open("/dev/watchdog", O_WRONLY);
    int ret = 0;
    if (fd == -1) {
        perror("watchdog file could not be opened\n");
        exit(EXIT_FAILURE);
    }
    printf("SUCCESS!\n");

    // Read and print driver info and capabilities
    printf("Attempting to get watchdog timer info/capabilities ... ");
    if (ioctl(fd, WDIOC_GETSUPPORT, &my_info) == -1) {
        printf("FAILED\n");
    } else {
        printf("SUCCESS!\n\n" "Driver type: %s \n" "Firmware version: %u\n" "Can detect Overheat Reset:.................%u\n" "Can detect Fan Fault Reset:................%u\n" "Can detect External Relay 1 Reset:........%u\n" "Can detect External Relay 2 Reset:........%u\n" "Can detect Power Under Reset:.............%u\n" "Can detect Card Reset:....................%u\n" "Can detect Power Overvoltage Reset:.......%u\n" "Capability to Set Timeout Interval:........%u\n" "Capability to Set Pre-Timeout:.............%u\n" "Capability to ping by IOCTL Call:.........%u\n" "Magic Close Feature:........................%u\n", my_info.identity, my_info.firmware_version, test_support(WDIOF_OVERHEAT), test_support(WDIOF_FANFAULT), test_support(WDIOF_EXTERN1), test_support(WDIOF_EXTERN2), test_support(WDIOF_POWERUNDER), test_support(WDIOF_CARDRESET), test_support(WDIOF_POWEROVER), test_support(WDIOF_SETTIMEOUT), test_support(WDIOF_PRETIMEOUT), test_support(WDIOF_KEEPALIVEPING), test_support(WDIOF_MAGICCLOSE));
    }

    // Write terminate character and close /dev/watchdog to turn off watchdog timer
    printf("End of test. Now disabling watchdog timer.\n");
    write(fd, "V", 1);
    close(fd);
}
In this case, the only supported capabilities are:

- **Read/set a timeout interval.** See Section 3.4, Changing the Watchdog Timeout Interval
- **Ping via IOCTL call.** See Section 3.3, Pinging the Watchdog Timer
- **The Magic Close Feature.** See Table 4, Watchdog Timer IOCTL Calls

The next two sections describe the capabilities displayed above and other capabilities that were not testable via the `watchdog_info` structure.

### 4.1 Advanced IOCTL Capabilities and Magic Close

Different watchdog timers will support varying IOCTL calls that allow advanced watchdog timer functionality. These IOCTL calls are:
### Table 4. Watchdog Timer IOCTL Calls

<table>
<thead>
<tr>
<th>Functionality</th>
<th>IOCTL Call(s)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Support</td>
<td><code>ioctl(fd, WDIOC_GETSUPPORT, &amp;my_info);</code></td>
<td>Write the driver name, firmware version, and driver capabilities to the variable <code>my_info</code>, a <code>watchdog_info</code> struct that was designed to hold this data.</td>
</tr>
<tr>
<td>Get Boot Status</td>
<td><code>ioctl(fd, WDIOC_GETBOOTSTATUS, &amp;status);</code>&lt;br&gt;or for some drivers: <code>ioctl(fd, WDIOC_GETSTATUS, &amp;status);</code></td>
<td>Get debugging information about the last system reset. The value written to <code>status</code> indicates how the last system reset occurred. To interpret this value, it should undergo a bitwise AND with one of the bitmasks described in Section 4.2, Using Watchdog Timer to Get Debugging Information.</td>
</tr>
<tr>
<td>Pre-Timeout</td>
<td><strong>Read:</strong>&lt;br&gt;<code>ioctl(fd, WDIOC_GETPRETIMEOUT, &amp;current_pretimeout);</code>&lt;br&gt;<strong>Set:</strong>&lt;br&gt;<code>ioctl(fd, WDIOC_SETPRETIMEOUT, &amp;desired_pretimeout);</code></td>
<td>A pre-timeout value allows the watchdog timer to trigger an interrupt before the true timeout is reached so that the panic information and core dumps can be recorded before the system is forcibly reset. These values are unsigned integers that represent seconds.</td>
</tr>
<tr>
<td>Get Time Left</td>
<td><code>ioctl(fd, WDIOC_GETTIMELEFT, &amp;time_remaining);</code></td>
<td>Returns the time remaining before the system will restart in the absence of a watchdog timer ping.</td>
</tr>
<tr>
<td>Get Temperature</td>
<td><code>ioctl(fd, WDIOC_GETTEMP, &amp;temperature)</code></td>
<td>Allows you to get the system temperature so a force restart can be performed if a temperature anomaly is detected. The returned temperature is an unsigned integer representing degrees Fahrenheit.</td>
</tr>
<tr>
<td>Magic Close</td>
<td>N/A&lt;br&gt;This is an inherent functionality.</td>
<td>If the driver supports this functionality, the watchdog timer will not be disabled until <code>V</code> is written to the <code>/dev/watchdog</code> file. This Magic Close feature is assumed in all code written in this document for portability.</td>
</tr>
</tbody>
</table>
4.2 Using Watchdog Timer for Debugging Information

Some watchdog timers will store information in a register about the last system reset which can be retrieved with the IOCTL call “Get Boot Status” (Table 4). To see what information your driver is capable of saving after a reset, run the source code provided by Figure 9, Using watchdog_info.

The return value of the “Get Boot Status” IOCTL call should undergo a bitwise AND with each of driver-support bitmasks in the table below to determine the cause of the last reset.

<table>
<thead>
<tr>
<th>Bitmask Name</th>
<th>Reset Type / Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDIOF_OVERHEAT</td>
<td>Overheat</td>
</tr>
<tr>
<td>WDIOF_FANFAULT</td>
<td>Fan Fault</td>
</tr>
<tr>
<td>WDIOF_EXTERN1</td>
<td>External Relay 1</td>
</tr>
<tr>
<td>WDIOF_EXTERN2</td>
<td>External Relay 2</td>
</tr>
<tr>
<td>WDIOF_POWERUNDER</td>
<td>Power Fault</td>
</tr>
<tr>
<td>WDIOF_CARDRESET</td>
<td>Card Reset</td>
</tr>
<tr>
<td>WDIOF_POWEROVER</td>
<td>Power Overvoltage</td>
</tr>
</tbody>
</table>

The following source code shows an example of how to ascertain if the last reboot was due to an overheat condition:

```c
unsigned int status;
ioctl (fd, WDIOC_GETBOOTSTATUS, &status);
if (status & WDIOF_OVERHEAT) {
    printf("Last reset was due to overheat/n");
}
```
4.3 Preventing Watchdog Timer Disable

For the most critical applications, the user might want to prevent the watchdog timer from being disabled at all. This configuration, supported by some watchdog timers, is achieved by setting the "Disable watchdog shutdown on close" configuration option to 'Y' when compiling the kernel.

By having this configuration enabled, a watchdog timer cannot be disabled by closing the /dev/watchdog file. The advantage of this configuration is the watchdog timer’s reliability is not compromised by errant or malicious software that somehow closes /dev/watchdog.
5 Putting the Custom Watchdog Daemon Together

Simple applications can use the standard watchdog timer daemon. For more complex applications, a watchdog timer daemon must perform actions of varying complexity that go beyond checking for a system stall, maximum temperature, or server ping response.

Such actions include taking corrective measures in response to a watchdog timer that is approaching its reset timeout. Another option is logging specific debugging data. Finally, the daemon could verify data transmission, sensor readings, and/or hardware connectivity.

When creating custom watchdog timer code, an executable must be created and have its location specified in the `watchdog.conf` file under the variable name `test-binary`. This code is inserted into the watchdog daemon's infinite loop to be executed at every loop iteration. The `test-timeout` variable in `watchdog.conf` must be set to 0 to give custom watchdog timer code the same priority as the built-in watchdog tests. See http://linux.die.net/man/5/watchdog.conf.

**Caution:** Code from the previous sections was run from the user-space. When custom watchdog timer code is in place, it will automatically run in the kernel-space and may reset the system inappropriately if not thoroughly debugged. Ensure that your code works as expected before allowing the daemon to automatically run the code.

Watchdog daemon test code is not limited to the C programming language. Bash scripts are fine as well.

For security and software reliability, the watchdog daemon code should be compiled within a BSP build, not cross compiled and transferred to the Gateway.

5.1 Code That Uses Watchdog IOCTL Calls

If your custom watchdog timer code uses IOCTL calls to the watchdog timer, it must create a file descriptor to the device file at `/dev/watchdog`. See the source code for Chapters 3 and 4. Therefore, the built-in watchdog timer's configuration file must be modified so the watchdog daemon does not create its own file descriptor. The watchdog device node can only be used by one process. The consequence of this are:

- The watchdog timer code must never terminate (run in an infinite loop) so that the watchdog file descriptor is not lost.
- Watchdog timer pinging must be performed within the custom code since it is the only process that can access the watchdog timer.
- The “watchdog-device” variable in `watchdog.conf` MUST be commented out so that the creation of a second file descriptor is not attempted.
Caution: If creation of a second file descriptor is attempted, the daemon will crash and the watchdog timer device will reset the system because it was not pinged. Therefore, any tests built into `watchdog.conf` that access the watchdog timer directly will not work.

A general template for custom watchdog timer code in C that uses IOCTL calls to the watchdog device is provided by the following example source code:

**Source Code 6. Customer Watchdog Timer Code Template with Watchdog IOCTL Calls**

```c
#include <linux/watchdog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define DESIRED_TIMEOUT 5

// Prevent watchdog timer from restarting the system
void ping_watchdog(fd) {
    write(fd, "\0", 1);
}

// Main function
int main(void) {
}
```

```c
    // Create file descriptor and enable watchdog
    printf("Opening watchdog file at /dev/watchdog ...");
    int fd = open("/dev/watchdog", O_WRONLY);
    int ret = 0;
    if (fd == -1) {
        printf(stderr,"watchdog file could not be opened\n");
        exit(EXIT_FAILURE);
    }

    // Read the timeout interval from the watchdog timer hardware
    unsigned int timeout_interval;
    if (ioctl(fd, WDIOC_GETTIMEOUT, &timeout_interval) < 0) {
        printf(stderr, "watchdog timer interval could not be read\n");
        exit(EXIT_FAILURE);
    }

    // Make sure the timeout interval is set to DESIRED_TIMEOUT.
    if (timeout_interval != DESIRED_TIMEOUT) {
        unsigned int new_interval = DESIRED_TIMEOUT;
        if (ioctl(fd, WDIOC_SETTIMEOUT, &new_interval) < 0) {
            printf(stderr,"watchdog timeout interval could not be set\n");
            exit(EXIT_FAILURE);
        }
    }

    // PERFORM CUSTOM TESTS
    while(1){
        // Hypothetical test 1
        while(getRAMusage() > 50){
            // stall
        }
        // Hypothetical test 2
    }
```
if (verifyHardwareConnection() == -1){
    // immediately reboot
    return -1;
}
// Ping the watchdog timer hardware indicating all tests passed
ping_watchdog(fd);
}

The source code demonstrates how \texttt{while} statements should be used for tests that may eventually pass if given enough time. In contrast, \texttt{if} statements are for tests that should immediately reboot the system upon failure.

## 5.2 Code without Watchdog IOCTL Calls

If your custom watchdog timer code does not make watchdog timer IOCTL calls, the resulting code will have the following properties:

- The custom test code can be used in tandem with any of the other tests within the \texttt{watchdog.conf} file.
- Test code must return an integer value.

If the test code succeeds, specify that it returns 0 (zero). Specify that it returns -1 to have the watchdog daemon immediately reset the system. For a full list of reserved return codes for the test script, see http://manpages.ubuntu.com/manpages/precise/man8/watchdog.8.html.

A general template for custom watchdog timer code in C that does not use IOCTL calls to the watchdog device is provided by the following sample code.

### Source Code 7. Watchdog Test Code Template without Watchdog IOCTL Calls

```c
// Main function
int main(void)
{
    // PERFORM CUSTOM WATCHDOG ROUTINES ...

    // Hypothetical test 1
    while (getRAMusage() > 50){
        // stall
    }

    // Hypothetical test 2
    if (verifyHardwareConnection() == -1){
        // immediately reboot
        return -1;
    }

    // Return 0 indicating all custom tests passed.
    return 0;
}
```
There are no interactions with the watchdog timer device in this code. All watchdog device interaction is performed by the daemon and is configured via the `watchdog.conf` file.