LynxOS-178 Health Monitor Application Writer's Guide

LynxOS-178

DOC-2209-00

Table of Contents

Preface

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to filenames and commands are case-sensitive and should be typed accurately.

Kind of Text	Examples
Body text; *italicized* for emphasis, new terms, and book titles	Refer to the *LynxOS-178 Health Monitor Application Writer's Guide*
Environment variables, filenames, functions, methods, options, parameter names, path names, commands, and computer data	`ls -l myprog.c /dev/null`

Commands that need to be highlighted within body text or commands that must be typed as is by the user are **`bolded`**.	`login:` **`myname`**
	`#` **`cd /usr/home`**

Text that represents a variable, such as a filename or a value that must be entered by the user, is *italicized*.	`cat` *`<filename>`*
	`mv` *`<file1> <file2>`*

Blocks of text that appear on the display screen after entering instructions or commands	`Loading file /tftpboot/shell.kdi into 0x4000`
	`.....................................`
	`File loaded. Size is 1314816`
	`© 2015 Lynx Software Technologies, Inc. All rights reserved.`

Keyboard options, button names, and menu sequences	**`Enter, Ctrl-C`**

Technical Support

Lynx Software Technologies handles support requests from current support subscribers. For questions regarding Lynx Software Technologies products, evaluation CDs, or to become a support subscriber; our knowledgeable sales staff will be pleased to help you. Please visit us at:
http://www.lynx.com/training-support/contact-support/

How to Submit a Support Request

When you are ready to submit a support request, please include *all* of the following information:

- First name, last name, your job title
- Phone number, e-mail address
- Company name, address
- Product version number
- Target platform (for example, PowerPC)
- Board Support Package (BSP), Current Service Pack Revision, Development Host OS version
- Detailed description of the problem that you are experiencing:
- Is there a requirement for a US Citizen or Green Card holder to work on this issue?
- Priority of the problem - Critical, High, Medium, or Low?

Where to Submit a Support Request

Support, Europe	tech_europe@lynx.com +33 1 30 85 93 96
Support, worldwide except Europe	support@lynx.com +1 800-327-5969 or +1 408-979-3940 +81 33 449 3131 [for Japan]
Training and Courses	USA: training-usa@lynx.com Europe: training-europe@lynx.com USA: +1 408-979-4353 Europe: +33 1 30 85 06 00

CHAPTER 1 Introduction

The Health Monitor (HM) is a component within the LynxOS-178 Operating System which is in accordance with DO-178C Level A.

Virtual Machine Configuration Table

The Virtual Machine Configuration Table (VCT) is a file that contains configuration information for the target system running LynxOS-178. The contents of the VCT should be thought of as a well-defined set of descriptors that configures the LynxOS-178 Operating System. It supports the ability to create partitions (also known as virtual machines (VMs)) and configure each partition to match its design as determined by the user.

The VCT is organized into four areas. The first area contains information relating to the target system or all VMs. The second area contains a table of information specific to each VM. This section can be referred to as the virtual machine configuration profile — and is used to allocate system resources to the application software. The third area contains a table of information relating to each dynamic device driver to install. The last area contains information relating to each configurable file system to mount.

Based on the values of the descriptors of the VCT, the LynxOS-178 Kernel manages all of the loaded applications that are running under each partition (VM). The VCT file must be placed in both the `/usr/local/etc` and `/usr/etc` directories.

For complete details on the VCT, refer to the *LynxOS-178 User's Guide*. The following descriptors are typically of interest to Health Monitor application writers:

- VctCrc
- ActionOnVmErr
- ActionOnSigillExcp
- ActionOnSigfpeExcp
- ActionOnSigsegvExcp
- ActionOnSigbusExcp

Example VCT Entries for Health Monitor

```
<VM0>
   ActionOnVmErr=0;
   ActionOnSigillExcp=Override;
   ActionOnSigfpeExcp=Override;
   ActionOnSigsegvExcp=Override;
   ActionOnSigbusExcp=Override;
</VM0>
<VM1>
   ActionOnVmErr=3;
   ActionOnSigillExcp=Default;
   ActionOnSigfpeExcp=Default;
   ActionOnSigsegvExcp=Override;
   ActionOnSigbusExcp=Default;
</VM1>
```

The example above configures the system with two partitions; VM0 and VM1. By design, VM0 is a unique virtual machine with special privileges. These privileges are similar to the root privileges in a UNIX system. For example, VM0 can override protections set in other VMs and can reboot the computer. In addition, VM0 monitors the state of the processes and threads contained within the other VMs.

The example sets the following:

- It sets applications running under the VM0 partition to:
  - Reset the VM on every occurrence of a VM Fatal Error.
  - Allow the VM to handle all synchronous exceptions.
- It sets applications running under the VM1 partition to:
  - Halt the VM after three occurrences of a VM Fatal Error.
  - Allow the VM to handle the SIGSEGV synchronous exception only.

Note that VM1's configuration profile does not follow the heuristics of the descriptors described earlier in this section. This example configuration, however, can be useful when debugging LynxOS-178 application(s). The VCT is highly versatile that the design configuration is entirely up to the user; caution is strongly advised.

Please note that the example does not contain a complete table. A complete sample VCT file, `vct.sample`, can be found in the `/etc` directory of the LynxOS-178 installation.

CHAPTER 2

Health Monitor Application Program Interface

The Health Monitor device driver provides local methods, in the form of an Application Program Interface (API) to communicate with the Health Monitor application. The API enables applications to log application errors, and register for and strobe a software watchdog monitor. The header file, `hm_app_api.h`, which defines the application interfaces of the HM lives in the `/usr/include/hm` directory of the LynxOS-178 installation.

The Health Monitor software structure and partitioning are illustrated in Figure 3-1.

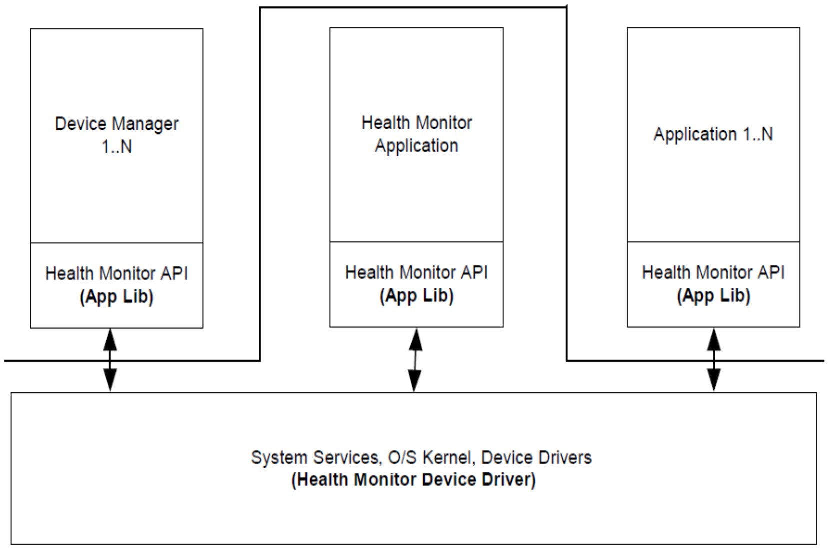
Device Manager 1..N
Health Monitor API (App Lib)
Health Monitor Application
Health Monitor API (App Lib)
Application 1..N
Health Monitor API (App Lib)
System Services, O/S Kernel, Device Drivers
(Health Monitor Device Driver)

Figure 2-1: Health Monitor Software Structure and Partitioning

Communicating Application Health

The communication of health is in the form of raising and clearing errors. There are two different sources for applications to raise and clear errors with the HM application. One is to use the API provided by the Health Monitor device driver. The other is to cause processor exceptions. Each error is made up of an error code, the severity, and a description.

Error Code

Error codes uniquely identify an error. Error codes must be unique within the VM. This element facilitates the communication and storage of errors. To ensure uniqueness, applications need to share the error codes when running under the same VM. For example, if applications A and B are running in the same VM, the applications need to coordinate the allocation of error codes to ensure uniqueness. If both applications define application error code 1, there would be no way to determine if the error was for application A or B.

The range for a valid application error code for each VM is 0 to 255, inclusive. Table 2-1 shows an example layout of error codes.

Table 2-1: Sample Error Code Layout

	ERROR CODE	VM1—IOC	VM2—APP A & APP B	System VM
Application Error Codes are unique per VM.	1	IOC error 1	APP A error 1	HM error 1
	2	IOC error 2	APP A error 2	HM error 2
	3	IOC error 3	APP B error 1	DL error 1
	...			
	APP_ERR_MAX			

Error Severity

The severity of an error determines the level of recovery action performed by the Health Monitor device driver. The error severity types are patterned after the Minimum Operational Performance Standards (MOPS) for Avionics Computer Resource (ACR), although the Health Monitor does not include process level errors. For applications, the two severity types defined from least severe to most severe are as follows:

Warnings

Warnings generate no recovery action by the Health Monitor. The information will be written to the real time error status record.

VM Fatal

Logging a VM fatal error will kill all processes in the virtual machine the fatal error was reported against. Applications can report VM fatal errors only against themselves. The “init process” will restart the VM or let it remain halted according to the VCT configuration profile entry for the VM.

Note that the Health Monitor application itself can report a VM fatal error. The other VMs should not be allowed to run while the Health Monitor is being reset or potentially halted.

Error Description

The description is a text string to support isolation of the error within the module (for example, to a Shop Replacement Unit (SRU) or functional circuit).

The maximum valid error description string message is 150 ASCII characters, plus the NULL terminator. The NULL error description string is considered valid. The ASCII carriage return, line feed and tab characters cannot be used in the string.

Below is an example of an error table:

```
error_record error_table[MAX_ERRORS]={
   {0, WARNING,  "Error 0"},
   {1, WARNING,  "Error 1"},
   {2, WARNING,  "Error 2"},
   {3, VM_FATAL, "Error 3"},
   {4, VM_FATAL, "Error 4"}
};
```

Using the HM API to Log Application Health

An application must first register itself to log errors with the Health Monitor device driver. When raising an error, the error code will be set in the error status report and the error description is written to the maintenance file. Clearing an error will cause the error to be cleared in the error status report and marked as intermittent in the maintenance file.

The maximum number of queued error records for logging by the Health Monitor application is 5 per virtual machine. Warning, clear, and VM fatal error records are counted for the queue. There will exist only one VM fatal error per virtual machine in the queue for the HM application to log. When a virtual machine is restarted, during a service of a VM fatal error, the VM's application and system real-time status for all error codes will be set to a value of zero.

Error Logging API

`int hm_register(void)`

Name

```
hm_register
```

Synopsis

```
#include <hm/hm_app_api.h>
int hm_register(void)
```

Description

Registers a process with the Health Monitor. None of the other Health Monitor APIs can be used until the process has registered.

Return Value

`HM_API_SUCCESS` on success

`HM_API_FAILURE` on failure

Failure modes:

- System was unable to allocate memory for the device driver static structure.
- The environment variable for the device node path was not found.

```
int hm_raise_application_error(
        const HM_ERROR_CODE_TYPE error_code,
        const ERROR_SEVERITY_TYPE error_severity,
        const ERROR_STRING_TYPE description)
```

Name

```
hm_raise_application_error
```

Synopsis

```
#include <hm/hm_app_api.h>
int hm_raise_application(const HM_ERROR_CODE_TYPE
        error_code,
        const ERROR_SEVERITY_TYPE
```

```
       error_severity,
       const ERROR_STRING_TYPE
       description)
```

Description

This function provides an interface for applications to report errors to the Health Monitor.

Arguments

`error_code` - VM-unique error code

`error_severity`- WARNING: Logs the error in the maintenance file and sets the error in the real-time reporting.

VM_FATAL: The VM will be reset or halted depending on the fatal recovery action for the VM in the VCT, the error is logged in the maintenance file, and the error code is set in the real-0time reporting.

`description` - Detailed description of the error. Any values included in the string should be converted to ASCII text. TAB, CR, and LF characters are not allowed in the description string.

Return Value

`HM_API_SUCCESS` on success

`HM_API_FAILURE` on failure

Failure modes:

- Process has not registered with the Health Monitor
- String exceeds max length or is not NULL terminated
- Invalid characters in error string
- Outside the range of valid severities the applications can report
- Outside the range of valid application error codes

`HM_QUEUE_FULL` if VM queue is full.

```
int hm_clear_application_error(const HM_ERROR_CODE_TYPE
                                     error_code)
```

Name

`hm_clear_application_error`

Synopsis

```
#include <hm/hm_app_api.h>
int hm_clear_application_error(const HM_ERROR_CODE_TYPE
       error_code)
```

Description

This function provides applications a method to clear an error that has been previously set.

Arguments

`error_code` - VM-unique error code to clear

Return Value

`HM_API_SUCCESS` on success

`HM_API_FAILURE` on failure

Failure modes:

- Process has not registered with the Health Monitor.
- Outside the range of valid application error codes.

```
int hm_raise_super_error(
                  const HM_ERROR_CODE_TYPE error_code,
                  const ERROR_SEVERITY_TYPE error_severity,
                  const ERROR_STRING_TYPE description,
                  const boot_mode_t mode)
```

Name

`hm_raise_super_error`

Synopsis

```
#include <hm/hm_app_api.h>
int hm_raise_super_error(
      const HM_ERROR_CODE_TYPE error_code,
      const ERROR_SEVERITY_TYPE error_severity,
      const ERROR_STRING_TYPE description,
      const boot_mode_t mode)
```

Description

This function raises the supervisor error. When the supervisor error happens, the Health Monitor logs the supervisor error record in the maintenance file and then reboots the target.

Invocation of `hm_raise_super_error()` is available on VM0 only. Invoked on non-VM0 the function returns a failure.

Arguments

`error_code` - VM-unique error code

`error_severity` -

- `WARNING` - this is a warning.

- `VM_FATAL` - this is a fatal error.

`description`- Detailed description of the error. Any values included in the string should be converted to ASCII text. TAB, CR, and LF characters are not allowed in the description string.

`mode` - The operation system will reboot into this mode.

Return Value

It never returns on a success.

`HM_API_FAILURE` on failure

Failure modes:

- Process has not registered with the Health Monitor
- String exceeds max length or is not NULL terminated
- Invalid characters in error string
- Outside the range of valid severities the applications can report
- Outside the range of valid application error codes
- Invoke on non-VM0

Using Exceptions to Communicate Application Health

The other way an application can communicate to the Health Monitor application is by throwing a processor exception.

Exceptions, either asynchronous or synchronous, are implemented as UNIX signals. Synchronous signals are so named because they happen in immediate response to something the process/thread does. Asynchronous signals can happen at any time and include timer or receipt of a message on a message queue.

Processor exceptions are categorized as synchronous signals since they result directly from something the process has done itself. The following signals are defined as synchronous:

- SIGILL- Illegal instruction execution attempted.
- SIGSEGV - Memory Access exception.
- SIGBUS - Bus Error.
- SIGFPE - Floating point error (Divide by Zero, Overflow, Underflow, Not a number).

When an application, also a process, raises a processor exception, the exception handler will dump exception data in a known location. It uses a semaphore to signal the virtual machine's HM kernel thread to wake up. The kernel thread checks to see if the process has registered a signal handler and, whether or not it is allowed to handle the signal. This allowance is defined in the VCT. If both conditions are met, the kernel thread will send a signal to the process and return to sleep. The application will again begin executing, but in the signal handler for the exception signal. If there was no definition in the VCT on how the application should handle the exception signal, then the default handler is run and the process is terminated.

An application is either expecting to have an exception or it is not. If it is an expected exception, the application must register a signal handler to catch that signal. This can be done using the POSIX `sigaction` function. If the exception is unexpected, then the kernel thread will halt the entire VM immediately and log the exception with the Health Monitor.

Below is an example of how to register a signal handler:

```
/* registering for real-time signal handler */
sa.sa_sigaction = wd_strobe_function;
sa.sa_flags = SA_SIGINFO;
sigemptyset(&sa.sa_mask);
sigaction(SIGRTMIN, &sa, NULL);
```

Note that if an application throws a processor exception and the VCT was configured not to allow the application to handle the exception signal, it is a VM Fatal error. This behavior is also exhibited when the VCT is configured to allow applications to handle exception signals, but no signal handler is defined within the application.

Signal Exceptions API

`int hm_set_sigaction(const SET_VM_EXCPS_IOCTL_TYPE *dexcp)`

Name

`hm_set_sigaction`

Description

This function provides applications a method to configure the exception handling. The following fields of the `dexcp` structure need to be filled:

`dexcp.vm_id = <vmid>`- the VM identifier

`dexcp.override_sigs = <sig_mask>` - the signal mask of the type `sigset_t`.

Return value

`HM_API_SUCCESS` on success

`HM_API_FAILURE` on failure

Failure modes:

- Process has not registered with the Health Monitor

Registering for the Software Watchdog Monitor

The Health Monitor API includes a software watchdog function to provide applications the mechanism to use a watchdog monitor. An application can register for a software watchdog monitor and strobe the watchdog. First, however, the application must register with the HM device driver. The Health Monitor application will monitor the application's watchdog. The watchdog monitor only counts strobes. So, if the application fails to strobe the watchdog at the required rate, the Health Monitor will perform the desired action.

Each VM can only have five watchdog monitors. The range for a valid strobe rate is 1 Hz to 100 Hz, inclusive.

Software Watchdog API

```
SW_MONITOR_ID_TYPE hm_register_sw_watchdog(
    const STROBE_RATE_TYPE strobes_per_second,
    const FATAL_FLAG_TYPE fatal_flag)
```

Name

`hm_register_sw_watchdog`

Synopsis

```
#include <hm/hm_app_api.h>
SW_MONITOR_ID_TYPE hm_register_sw_watchdog(
    const STROBE_RATE_TYPE strobes_per_second,
    const FATAL_FLAG_TYPE fatal_flag)
```

Description

This function provides applications the ability to register for a software watchdog. The caller specifies the minimum `strobes_per_second` and whether or not it is fatal if it fails to meet the minimum `strobes_per_second`. The function returns a `sw_monitor_id`, which will be used when strobing the monitor.

Arguments

`strobes_per_second`- Minimum rate at which this monitor can be strobed.

`fatal_flag`- Flag that indicates if a failure of this monitor is reported as `VM_FATAL` or `WARNING`.

Return Value

`monitor_id` on success

`HM_API_FAILURE` on failure

Failure modes:

- Process has not registered with the Health Monitor.
- Strobe rate requested is not supported.
- No more software watchdogs available for the VM.

```
int hm_strobe_sw_watchdog(
    const SW_MONITOR_ID_TYPE sw_monitor_id)
```

Name

`hm_strobe_sw_watchdog`

Synopsis

```
#include <hm/hm_app_api.h>
int hm_strobe_sw_watchdog(
  const SW_MONITOR_ID_TYPE sw_monitor_id)
```

Description

This function provides applications the ability to strobe the software watchdog monitor.

Arguments

`sw_monitor_id`- Watchdog ID to strobe.

Return Value

`HM_API_SUCCESS` on success

`HM_API_FAILURE` on failure

Failure modes:

- Process has not registered with the Health Monitor.
- Invalid `sw_monitor_id`.

CHAPTER 3

Health Monitor Device Interface

This chapter describes the interface of the Health Monitor Device Driver that is the kernel-side component of the Health Monitor. It is the internal Health Monitor interface that is used by the HM API library.

A regular application desiring to interface with the Health Monitor should never use this interface. Instead, it should use the interface provided by the HM API library as described in Chapter 2, "Health Monitor Application Program Interface". The Health Monitor Driver interface is described for reference.

Health Monitor Driver Nodes

There are two types of the HM Device Driver nodes: privileged and unprivileged. The names of the privileged and unprivileged devices are `/dev/hmdd_su` and `/dev/hmdd`, respectively. The Health Monitor driver permits to use the privileged HM device from VM0 only. The unprivileged HM device can be accessed by any VM.

The HM driver allows performing the full set of operations on the privileged device. However these operations are restricted on the unprivileged HM device. Refer to Table 3-1 for details.

Table 3-1: Supported Operations on the Privileged or Unprivileged Devices

Functions	Description	Privileged Device	Unprivileged Device
`open()`	Open the HM device	Yes	Yes
`close()`	Close the HM device	Yes	Yes
`read()`	Get an error from HM error queue	Yes	No
`write()`	Put an error to the HM error queue	Yes	Yes
`ioctl(REGISTER_SW_WATCHDOG)`	Register software watchdog	Yes	Yes
`ioctl(STROBE_SW_WATCHDOG`	Strobe software watchdog	Yes	Yes

Table 3-1: Supported Operations on the Privileged or Unprivileged Devices (Continued)

`ioctl(TEST_SW_WATCHDOGS)`	Test software watchdogs	Yes	No
`ioctl(RAISE_SYSTEM_ERROR)`	Raise system error	Yes	No
`ioctl(CLEAR_SYSTEM_ERROR)`	Clear system error	Yes	No
`ioctl(GET_VM_RT_STATUS )`	Get real-time status of the VM errors	Yes	No
`ioctl(GET_VM_RTS_SUMMARY)`	Get the VM real-time error status summary	Yes	No
`ioctl(INFORM_VM_RESTART)`	Inform about VM restarting	Yes	No
`ioctl(SET_VM_EXCEPTIONS)`	Specify the signal exceptions	Yes	No

The Health Monitor Functions

Open the Health Monitor Pseudo Device

int open(const char *path, int mode)

To start using the HM device driver, an application must open the Health Monitor pseudo device. For this purpose, the application needs to invoke the `open()` function to open either the privileged or unprivileged HM device with the required options set.

The unprivileged HM device can be opened on any VM while the privileged HM device opens on VM0 only.

The HM pseudo devices can be opened for reading only or writing only or for both reading and writing modes.

Close The Health Monitor Pseudo Device

```
int close(int fildes)
```

An application shall close the HM pseudo devices if it is not required any more using the `close()` system call.

Error Logging API

`ssize_t read(int fildes, void *buf, size_t nbyte)`

The function gets an error record from the HM error queue.

An application can read an error record from the HM error queues by invoking the `read()` system call. The priority of the errors is as follows. The `read()` method returns the first fatal error occurred in any VM. If there are no fatal errors, it returns the first warning occurred in any VM. If there are no warnings then `read()` returns immediately.

The error or warning read is deleted from the HM error queue.

Invocation

```
read(hmfd, &erec, sizeof(ERROR_RECORD_TYPE));
```

Arguments

`hmfd`- the file descriptor of the HM device

`erec` - the error record. This is a structure of the type `ERROR_RECORD_TYPE`.

Description

The function puts the first fatal error or warning into the `erec` error record stored in the HM error queue and returns the number of bytes that it actually read or 0, if the error does not exist.

The read operation is supported by the privileged Health Monitor pseudo device only. For the unprivileged HM device `read()` returns a failure with the `ENODEV`error number.

Put An Error Record to the HM Error Queue

ssize_t write(int fildes, const void *buf, size_t nbyte)

An application can write an error record to the HM error queues by means of the `write()` function.

Invocation

```
write(hmfd, &erec, sizeof(ERROR_RECORD_TYPE)
```

Arguments

`hmfd`- the file descriptor of the HM device

`erec` - the error record. This is a structure of the type `ERROR_RECORD_TYPE`.

Description

The `write()` function puts the erec error record to the proper HM error queue. The following fields of the erec structure need to be filled:

- `erec.code = <code>` - VM-unique error number
- `erec.severity = <severity>`

There are three possible values of severity:

  - `WARNING` - Add the error to the HM warning error queue and set the error in the real-time reporting
  - `VM_FATAL` - Add the error to the HM fatal error queue and set the error in the real-time reporting. Also, it resets or halts the VM depending on the fatal recovery action for the VM in the VCT.
  - `CLEAR_ERROR` - Clear the error previously set.

- `erec.string = <description>`

  This is a detailed description of the error. Any values included in the string should be converted to ASCII text. TAB, CR and LF characters are not allowed in the description string.

  The write operation is supported on both privileged and unprivileged Health Monitor pseudo devices.

  The HM driver marks all errors registered using the `write()` function as application errors.

Software Watchdog Monitor

`ioctl(REGISTER_SW_WATCHDOG)`

This call registers a software watchdog.

An application can register a software watchdog by means of the `REGISTER_SW_WATCHDOG` command. Each VM can have a maximum of five watchdog monitors.

Invocation

```
ioctl(hmfd, REGISTER_SW_WATCHDOG, (char *)&wdog_io)
```

Arguments

`hmfd` - the file descriptor of the HM device

`wdog_io` - the structure of the `REG_WATCHDOG_IOCTL_TYPE` type

Description

The `REGISTER_SW_WATCHDOG` command registers the software watchdog described by the wdog_io structure. The following fields of the `wdog_io` structure need to be filled:

- `wdog_io.rate = <rate>`

  This field defines the minimum number of strobes per second. The range of the valid strobe rate is 1 to 100 inclusive.

- `wdog_io.fatal_flag = <flag>`

  There are two possible values of `fatal_flag`: true or false. If `fatal_flag` is set then a failure of the software will be considered as a fatal error. Otherwise, a software watchdog failure will be considered as a warning.

- `wdog_io.id = <pointer_id>`

  This is the pointer where `ioctl()` stores the software watchdog identifier on success. This `wdog_io.id` identifier needs to be used in the subsequent `ioctl()` calls.

The `ioctl(REGISTER_SW_WATCHDOG)` command is supported on both privileged and unprivileged Health Monitor pseudo devices.

`ioctl(STROBE_SW_WATCHDOG)`

This call strobes the software watchdog.

The application can strobe the software watchdog by means of the `STROBE_SW_WATCHDOG` command.

Invocation

```
ioctl(hmfd, STROBE_SW_WATCHDOG, (char *)&swid)
```

Arguments

`hmfd` - the file descriptor of the HM device

`swid` - the software watchdog identifier of the type `SW_MONITOR_ID_TYPE`. The `swid` identifier returned by the `ioctl(REGISTER_SW_WATCHDOG)` command.

Description

The `STROBE_SW_WATCHDOG` command strobes the software watchdog. The application needs to strobe the watchdog at the required rate that was specified by the `ioctl(REGISTER_SW_WATCHDOG)` command.

The `ioctl(STROBE_SW_WATCHDOG)` command is supported on both privileged and unprivileged Health Monitor pseudo devices.

`ioctl(TEST_SW_WATCHDOGS)`

This call tests software watchdogs.

An application can monitor all the software watchdogs present in the VM via the `TEST_SW_WATCHDOGS` command.

Invocation

```
ioctl(hmfd, TEST_SW_WATCHDOGS, (char *)&vmid)
```

Arguments

`hmfd` - the file descriptor of the HM device.

`vmid` - the identifier of the VM where it monitors the software watchdogs

Description

The `TEST_SW_WATCHDOGS` command monitors all the software watchdogs present in the VM identified by `vmid`. If the `TEST_SW_WATCHDOGS` command detects that an application fails to strobe the software watchdog at the required rate then it performs the action that was specified when the watchdog was registered (the `REGISTER_SW_WATCHDOG` command). Specifically, it delivers the fatal or warning error for this software watchdog.

The `ioctl(TEST_SW_WATCHDOGS)` command is supported on the privileged Health Monitor pseudo device only. If it is invoked on the unprivileged device then `ioctl()` returns a failure with the `ENODEV` error number.

Register System Errors

`ioctl(RAISE_SYSTEM_ERROR)`

This call raises system error.

An application can raise the supervisor error by means of the `RAISE_SYSTEM_ERROR` command.

Invocation

```
ioctl(hmfd, RAISE_SYSTEM_ERROR, &err)
```

Arguments

`hmfd` - the file descriptor of the HM device `err` - the supervisor error record, This is a structure of the type `SUPERVISOR_RECORD_TYPE`

Description

The `RAISE_SYSTEM_ERROR` command writes the supervisor error record to the persistent storage (NVRAM for example), if it is supported by the target platform. And then it reboots the target. The following fields of the `err` structure need to be filled:

- `err.rec.code = <code>` -VM-unique error number
- `err.rec.category = <category>`

  There are two possible values of category:

  `APPLICATION` - application errors reported by an application

  `SYSTEM` - system errors reported by the system software
- `err.rec.severity = <severity>`

  There are two possible values of severity:

  `WARNING` - the HM warning

  `VM_FATAL` - the HM fatal error
- `err.rec.string = <description>`

  This is a detailed description of the error. Any values included in the string should be converted to ASCII text. TAB, CR and LF characters are not allowed in the description string.
- `err.mode = <mode>` - the processor reboots into this mode
- `err.start = <start>` - the processor startup type that is used for the reboot

The `ioctl(RAISE_SYSTEM_ERROR)` command is supported on the privileged Health Monitor pseudo device only. If it is invoked on the unprivileged device then `ioctl()` returns a failure with the `ENODEV` error number.

`ioctl(CLEAR_SYSTEM_ERROR)`

This call clears a system error.

An application can clear the system error by means of the `CLEAR_SYSTEM_ERROR` command.

Invocation

```
ioctl(hmfd, CLEAR_SYSTEM_ERROR, &erec)
```

Arguments

`hmfd` - the file descriptor of the HM device

`erec` - the system error record, This is a structure of the type `ERROR_RECORD_TYPE`

Description

The `CLEAR_SYSTEM_ERROR` command clears the system error record that was previously added to the HM system error queue. The real-time status of the system error that is present in the system will be set to false. The following fields of the erec structure need to be filled:

- `erec.code = <code>`- VM-unique error number
- `erec.vm_id = <vmid>`- The VM identifier

  The `<code>` system error will be cleared on the `<vmid>` VM by the `CLEAR_SYSTEM_ERROR`command.

  The `ioctl(CLEAR_SYSTEM_ERROR)` command is supported on the privileged Health Monitor pseudo device only. If it is invoked on the unprivileged device then `ioctl()` returns a failure with the `ENODEV` error number.

Errors Status Information

`ioctl(GET_VM_RT_STATUS)`

This call gets real-time status of the VM errors.

An application can get the real-time status of all the errors in the VM.

Invocation

```
ioctl(hmfd, GET_VM_RT_STATUS, &rts)
```

Arguments

`hmfd` - the file descriptor of the HM device.

`rts` - the real-time status record, This is a structure of the type `RTS_IOCTL_TYPE`.

Description

The `GET_VM_RT_STATUS` command returns the real-time status of all the system or application errors in the specified VM. If the real-time status of the error set to true then the error is present in the VM and accordingly the error is not present If the status set to false.

The following fields of the `rts` structure need to be filled:

- `rts.vm_id = <vmid>`- The VM identifier
- `rts.status_type = <type>`-The types of getting errors. There are two possible error types:
  - `APPLICATION`-application errors reported by an application
  - `SYSTEM` –system errors reported by the system software
- `rts.rts_buf_ptr` - the real-time status of the errors.

  This is an `rts_buf_ptr`pointer to the union of the type `RT_STATUS_TYPE`. This union contains the array of the real-time status for the system or application errors. The `GET_VM_RT_STATUS` command will write the real-time status by the `rts_buf_ptr` pointer. The error is present in the VM, if the corresponding element of array set to true. Otherwise the error is not present.

  The `ioctl(GET_VM_RT_STATUS)`command is supported on the privileged Health Monitor pseudo device only. If it is invoked on the unprivileged device then `ioctl()` returns a failure with the `ENODEV` error number.

`ioctl(GET_VM_RTS_SUMMARY)`

This call gets the VM real-time error status summary.

An application can get the real-time error status summary for the VM via the `GET_VM_RTS_SUMMARY`command

Invocation

```
ioctl(hmfd, GET_VM_RTS_SUMMARY, &rsumm)
```

Arguments

`hmfd`- the file descriptor of the HM device

`rsumm`- the real-time status summary record, This is a structure of the type `RTS_SUMMARY_IOCTL_TYPE`

Description

The `GET_VM_RTS_SUMMARY` command returns the real-time error status summary for the specified VM. The summary includes information about:

- Types of errors (warning or error) that occur in the VM
- Errors are not present at all
- The VM restarts now

  The following fields of the `rsumm`structure need to be filled:

- `rsumm.vm_id = <vmid>`- The VM identifier
- `rsumm.type = <type>`-The types of errors summary. There are two possible types:
  - `APPLICATION` - application errors reported by an application
  - `SYSTEM` - system errors reported by the system software
- `rsumm.value = <value>`- returned summary value of the type `RTS_SUMMARY_TYPE`.

  There are four possible real-time status summary values:

  - `NO_ERRORS_REPORTED` - there are no errors in the VM.
  - `WARNING_REPORTED` - there are warnings in the VM
  - `VM_RESTARTING` – the fatal errors in the VM
  - `VM_HALTED` – the VM halts now

The `GET_VM_RTS_SUMMARY` command will return the VM status summary in the `rsumm.value`field.

The `ioctl(GET_VM_RTS_SUMMARY)` command is supported on the privileged Health Monitor pseudo device only. If it is invoked on the unprivileged device then `ioctl()` returns a failure with the `ENODEV` error number.

`ioctl(INFORM_VM_RESTART)`

This call informs about VM restarting.

An application can inform the Health Monitor that the VM reboots at this moment by means of the `INFORM_VM_RESTART`command.

Invocation

```
ioctl(hmfd, INFORM_VM_RESTART, &rest)
```

Arguments

`hmfd` - the file descriptor of the HM device

`rest`- the VM restart status, This is a structure of the type `RESTART_IOCTL_TYPE`

Description

The following fields of the rest structure need to be filled:

- `rest.vm_id = <vmid>` - the VM identifier
- `rest.restart_vm = <status>` - the restart status of the VM. If <status> is true then the VM restarts. Otherwise the VM halts.

The `INFORM_VM_RESTART` command informs the Health Monitor about VM restart. If the VM reboots (`restart_vm` is true) then all errors are cleared by `INFORM_VM_RESTART`.

The `ioctl(INFORM_VM_RESTART)` command is supported on the privileged Health Monitor pseudo device only. If it is invoked on the unprivileged device then `ioctl()` returns a failure with the `ENODEV` error number.

Signal Exceptions

`ioctl(SET_VM_EXCEPTIONS)`

This call specifies the signal exceptions.

An application can specify the signals that can be handled by means of the `SET_VM_EXCEPTIONS` command.

Invocation

```
ioctl(hmfd, SET_VM_EXCEPTIONS, &rexp)
```

Arguments

`hmfd` - the file descriptor of the HM device

`rexp` - setting of exceptions. This is a structure of the type `SET_VM_EXCPS_IOCTL_TYPE`.

Description

The `SET_VM_EXCEPTIONS` command will set the signal exceptions that are handled.

The following fields of the `rexp` structure need to be filled:

- `rexp.vm_id = <vmid>` - the VM identifier
- `rexp.override_sigs = <sig_mask>` - the signal mask of the type `sigset_t`.

The `ioctl(SET_VM_EXCEPTIONS)` command is supported on the privileged Health Monitor pseudo device only. For the unprivileged device `ioctl()` returns failure with the `ENODEV` error number.

CHAPTER 4 *Cyclic Redundancy Check*

Cyclic redundancy check (CRC) is an error detection method used to produce a check sequence, also known as a checksum, by performing polynomial division. A CRC is calculated and appended to the original stream before transmission or storage. Then it is verified by the receiver that no corruption is assumed to have occurred by using the same mathematical calculation. As a caveat, while CRCs are useful, they cannot be safely relied on for data integrity due to their linear structure of CRC polynomials.

The header file, `crc.h`, that defines the application interface of the CRC lives in the `/usr/include/crc` directory of the LynxOS-178 installation.

CRC API

`uint32_t crc_32_bytes(uint32_t seed, uint8_t *data, size_t size)`

Name

```
crc_32_bytes
```

Synopsis

```
#include <crc/crc.h>
uint32_t crc_32_bytes(uint32_t seed, uint8_t *data,
size_t size)
```

Description

This function computes a 32-bit CRC across the buffer specified by the data argument. Typically, the CRC32 calculation does a 1's complement on the final CRC value. This function does not perform the complement operation.

Arguments

`seed` - The previous CRC32 value or all 1's for first call

`data` - An array of bytes to be CRC-ed

`size` - the number of bytes to be CRC-ed

Return Value

The 32-bit CRC

uint16_tcrc_16_bytes(uint16_tseed,uint8_t*data, size_t size)

Name

`crc_16_bytes`

Synopsis

```
#include <crc/crc.h>
uint16_t crc_16_bytes(uint16_t seed, uint8_t *data,
size_t size)
```

Description

This function computes a 16-bit CRC across the buffer specified by the data argument.

Arguments

`seed`- The previous CRC16 value or all 1's for first call

`data`- An array of bytes to be CRC-ed

`size`- The number of bytes to be CRC-ed

Return Value

The 16-bit CRC

`int crc_16_stream(FILE *fp, size_t size, uint16_t *value)`

Name

`crc_16_stream`

Synopsis

```
#include <crc/crc.h>
int crc_16_stream(FILE *fp, size_t size, uint16_t
*value)
```

Description

Generate a CRC16 value for the file specified by the `fp`argument.

Arguments

`fp` - File stream to CRC

`size`- The number of bytes in the file being CRCed

`value`- Location to store the generated CRC16 value

Return Value

`CRC_SUCCESS` on success

`CRC_FAIL` on failure

`int crc_32_stream(FILE *fp, size_t size, uint32_t *value)`

Name

`crc_32_stream`

Synopsis

```
#include <crc/crc.h>
int crc_32_stream(FILE *fp, size_t size, uint32_t
*value)
```

Description

Generate a CRC32 value for the file specified by the `fp` argument. The CRC calculation begins at the current file position. Typically, the CRC32 calculation does a 1's complement on the final CRC value. This function does not perform the complement operation.

Arguments

`fp` - File to CRC

`size` - The number of bytes in the file being CRCed

`value` - Location to store the generated CRC32 value

Return Value

`CRC_SUCCESS` on success

`CRC_FAIL` on failure

APPENDIX A API QuickReference

Table A-1 provides an HM API quick reference.

Table A-1: provides an HM API quick reference

	Function	Parameter(s)	Returns
Registers a process with the Health Monitor:			
	`hm_register`	N/A	`HM_API_SUCCESS` `HM_API_FAILURE`
Reports errors to the Heath Monitor:			
	`hm_raise_application_error`	`const` `HM_ERROR_CODE_TYPE` `error_code` `const` `ERROR_SEVERITY_TYPE` `error_severity` `const` `ERROR_STRING_TYPE` `description`	`HM_API_SUCCESS` `HM_API_FAILUREHM_QUEUE_FULL`
Clears previous errors with the Health Monitor:			
	`hm_clear_application_error`	`const` `HM_ERROR_CODE_TYPE` `error_code`	`HM_API_SUCCESS` `HM_API_FAILURE`
Registers for a software watchdog monitor:			
	`hm_register_sw_watchdog`	`const` `STROBE_RATE_TYPE` `strobes_per_second` `const` `FATAL_FLAG_TYPE` `fatal_flag`	`SW_MONITOR_ID_TYPE monitor_id` `HM_API_FAILURE`

Table A-1: provides an HM API quick reference (Continued)

	Strobes the software watchdog monitor:		
	`hm_strobe_sw_watchdog`	`const SW_MONITOR_ID_TYPE sw_monitor_id`	`HM_API_SUCCESS` `HM_API_FAILURE`
Reports the super error to Health Monitor:			
	`hm_raise_super_error`	`const HM_ERROR_CODE_TYPE error_code` `const RROR_SEVERITY_TYPE error_severity` `const ERROR_STRING_TYPE description` `const boot_mode_t mode`	Never on success `HM_API_FAILURE`
Configures exception handling:			
	`hm_set_sigaction`	`const SET_VM_EXCPS_IOCTL_TYPE *dexcp`	`HM_API_SUCCESS HM_API_FAILURE`
Compute a 32-bit CRC:			
	`crc_32_bytes`	`uint32_t seed,uint8_t *data, size_t size`	`uint32_t 32_bit_crc`
Compute a 16-bit CRC:			
	`crc_16_bytes`	`uint16_t seed, uint8_t *data, size_t size`	`uint16_t 16_bit_crc`
Compute a 16-bit CRC for the passed in file:			
	`crc_16_stream`	`FILE * fp, size_t size, uint16_t *value`	`CRC_SUCCESS CRC_FAIL`
Compute a 32-bit CRC for the passed in file:			
	`crc_32_stream`	`FILE * fp, size_t size, uint32_t *value`	`CRC_SUCCESS CRC_FAIL`

Table A-2 provides correspondence between the HM API and the HM device driver interface.

Table A-2 Correspondence Between the HM API and the HM Device Driver Interface

The HM Character Driver Interface	The HM API
`open()`	`hm_register()`
`write(category = APPLICATION)`	`hm_raise_application_error()`
`write(category = APPLICATION, severity = CLEAR_ERROR)`	`hm_clear_application_error()`
`ioctl(REGISTER_SW_WATCHDOG )`	`hm_register_sw_watchdog()`
`ioctl(STROBE_SW_WATCHDOG)`	`hm_strobe_sw_watchdog()`
`ioctl(RAISE_SYSTEM_ERROR)`	`hm_raise_super_error()`
`ioctl(SET_VM_EXCEPTIONS)`	`hm_set_sigaction()`

APPENDIX B Example Source Code

Makefile

```
#include $(ENV_PREFIX)/ENVIRONMENT$(ENV_SUFFIX)

GCCOPTS= -c -MMD -Wall -ansi -g -D_USE_FIXED_PROTOTYPES__

all: hm_app

# Make sure to link in HM and Threads libraries
hm_app: example_hm_user_app.o
   gcc example_hm_user_app.o -o hm_app -lhm

example_hm_user_app.o: example_hm_user_app.c global_error_table.h
   gcc ${GCCOPTS} example_hm_user_app.c

clean:
   rm -f *.o
   rm -f *.d

spotless:
   make clean
   rm -f hm_app
   rm -f *~
```

global_error_table.h

```
/*******************************************************
PURPOSE
Sample code to show an example error table shared by
user applications running under the same
 VM.
*******************************************************/
#include <hm/hm_app_api.h>

#define MAX_ERRORS 5

typedef struct error_record {
   HM_ERROR_CODE_TYPE type;
   ERROR_SEVERITY_TYPE severity;
   ERROR_STRING_TYPE string;
} error_record;

error_record error_table[MAX_ERRORS]={
   {0, WARNING,  "Error 0"},
   {1, WARNING,  "Error 1"},
   {2, WARNING,  "Error 2"},
   {3, VM_FATAL, "Error 3"},
   {4, VM_FATAL, "Error 4"}
};
```

example_hm_user_app.c

```
/*****************************************************************
 * PURPOSE
 *   Sample code to show how to use the Health Monitor (HM) API:
 *     1) registering with the HM driver
 *     2) registering for a software watchdog monitor
 *     3) strobing the software watchdog monitor
 *     4) raising errors
 *     5) registering signal handler to catch processor exceptions

 ****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <arinc653/arinc653.h>
#include "global_error_table.h"
#include <time.h>
#include <unistd.h>

/* global vars for strobing the watchdog monitor */
STROBE_RATE_TYPE strobe_rate = 2; /* 2 seconds */
SW_MONITOR_ID_TYPE wd_mon_id;
struct sigaction sa;
struct itimerspec timer_attr;
struct sigevent notif;
timer_t timer_id;
int timer_counter = 4;

/* global vars for catching a synchronous SIGSEGV processor exception */
struct sigaction segv_action, old_action;
sigset_t mask;
short sigsegv_flag = 0;

/* Function prototypes */
int start_wd_timer(void);
void wd_strobe_function(int signo, siginfo_t *info, void *context);
void throw_sigsegv_excp(void);
void segv_handler(int signo);
void print_error(int flag);

int main( void )
{
 int success;

 /*setting env var for persistent storage and health*/

 if (0 != putenv("DEVNODEPATH=/dev")) {
   printf("Could not set DEVNODEPATH env var\n");
   return EXIT_FAILURE;
 }

 success = hm_register();

 if (success == HM_API_FAILURE) {
   perror("register HM driver");
   return EXIT_FAILURE;
 }

 printf("Sanity check: registered successfully with HM\n");

 /* register for a watchdog monitor */
 wd_mon_id = hm_register_sw_watchdog(strobe_rate, WARNING);
```

```
    if (!start_wd_timer())
      exit(EXIT_FAILURE);
    /* sleep to let to give the thread a chance to run */ while
    (timer_counter--) {

      if (timer_counter == 2){
       /* throw a processor exception */
       throw_sigsegv_excp();
       printf("Threw SIGSEGV exception.\n");
      }
      if (sigsegv_flag) {
       printf("Hit SIGSEGV exception.\n");
       sigsegv_flag = 0;
       /* create VM Fatal error */
       /* success = hm_raise_application_error(error_table[3].type,
       error_table[3].severity, error_table[3].string ); */
       /* create Warning error */
       success = hm_raise_application_error(error_table[2].type,
       error_table[2].severity, error_table[2].string);
       print_error(success);
      }
      pause();
    }

    return EXIT_SUCCESS;
}

  int start_wd_timer()
  {
   int flags = 0; /* use relative time */

   /* registering for real-time signal handler */
   sa.sa_sigaction = wd_strobe_function;
   sa.sa_flags = SA_SIGINFO;
   sigemptyset(&sa.sa_mask);
   sigaction(SIGRTMIN, &sa, NULL);

   /* init signal and data to send */
   notif.sigev_signo = SIGRTMIN;
   notif.sigev_notify = SIGEV_SIGNAL;

   /* create timer */
   if (timer_create(CLOCK_REALTIME, &notif, &timer_id))
     return 0; /* failure */

   /* set timer attributes */
   timer_attr.it_value.tv_sec = 2; /* delay until timer fires */
   timer_attr.it_value.tv_nsec = 0;
   timer_attr.it_interval.tv_sec = 2; /* interval for subsequent fires */
   timer_attr.it_interval.tv_nsec = 0;

   /* start timer */
   if (timer_settime(timer_id, flags, &timer_attr, NULL))

     return 0; /* failure */

   return 1; /* success */
  }

  void wd_strobe_function( int signo, siginfo_t *info, void *context )
  {
   switch (hm_strobe_sw_watchdog(wd_mon_id)) {
   case HM_API_SUCCESS:
      printf("Strobe...\n");
      break;
    default:
      perror("strobe watchdog");
      exit(EXIT_FAILURE);;
   }
 }

 void throw_sigsegv_excp()
 {
   pid_t pid = getpid();

   /* registering for standard signal handler */
   segv_action.sa_handler = segv_handler;
   sigemptyset(&segv_action.sa_mask);
   segv_action.sa_flags = 0;
   sigaction(SIGSEGV, &segv_action, &old_action);

   /* cause a sigsegv */
   kill(pid, SIGSEGV);

 }

 void segv_handler(int signo)
 {
   sigsegv_flag = 1;
 }

 void print_error(int flag)
 {
   switch (flag) {
   case HM_API_SUCCESS:
    printf("Success: raise app error\n");
    break;
   case HM_API_FAILURE:
    perror("fail raising error");
    exit(EXIT_FAILURE);
   case HM_QUEUE_FULL:
    perror("queue full when raising error");
    exit(EXIT_FAILURE);
   default:
    perror("unexpected error in raising error");
    exit(EXIT_FAILURE);
 }
}
```

APPENDIX C Error Code Tables

Table C-1, Table C-2, and Table C-3 provide information about the VM0 application error codes, the system error codes, and the generic VM system error codes.

Table C-1: VM0 Application Error Code Table

Error Logical Name	Error Code Number	Severity
System call failed	0	VM fatal
Initialization timeout	1	VM fatal
Incorrect configuration	2	VM fatal
File open error	3	Warning
File read error	4	Warning
xBit error	5	Warning
HW Watchdog thread cycle slip	6	Warning
Motherboard Monitor cycle slip	7	Warning
SW Watchdog Monitor cycle slip	8	Warning
RAM test cycle slip	9	Warning
Error data thread cycle slip	10	Warning
DRAM Channel A SEC error	11	Warning
DRAM Channel A DED error	12	Warning
DRAM Channel B SEC error	13	Warning
DRAM Channel D DED error	14	Warning
RAM test failed	15	VM fatal
Reserved for HM application	16 - 225	Reserved for HM application
cinit error: syscall return error / mkffs or ffsck exits abnormally / cinit cannot start/restart partition(s)	226	Warning
Reserved for the CINIT	227	

Table C-1: VM0 Application Error Code Table (Continued)

Invalid configuration (VCT parse error)	228	Module Fatal
Fatal reset limit exceeded	229	Module Fatal
Process terminations start (VM0)	230	VM Fatal
Process terminations start (VM1)	231	VM Fatal
Process terminations start (VM2)	232	VM Fatal
Process terminations start (VM3)	233	VM Fatal
Process terminations start (VM4)	234	VM Fatal
Process terminations start (VM5)	235	VM Fatal
Process terminations start (VM6)	236	VM Fatal
Process terminations start (VM7)	237	VM Fatal
Reserved for the CINIT	238 - 255	

Table C-2: System Error Code Table

Error Logical Name	Error Code Number	Severity
File integrity fail	0	VM Fatal
Reserved	1 - 3	
Unhandled SIGILL error	4	VM Fatal†
Reserved	5 - 7	
Unhandled SIGFPE error	8	VM Fatal†
Reserved	9	
Unhandled SIGBUS error	10	VM Fatal†
Unhandled SIGSEGV error	11	VM Fatal†
Reserved	12 - 14	
Software Watchdog monitor error (0)	15	Application- configurable
Software Watchdog monitor error (1)	16	Application- configurable
Software Watchdog monitor error (2)	17	Application- configurable

Table C-2: System Error Code Table (Continued)

Software Watchdog monitor error (3)	18	Application- configurable
Software Watchdog monitor error (4)	19	Application- configurable
Reserved for PMC EDAC	20 - 24	
Reserved for Persistent Storage	25 – 44	
Reserved for Read/Write File System	45 - 74	
Kernel: Kernel panic	75	Module Fatal
Kernel: Unsupported system call invoked	76	Warning
Kernel: Attempt to exceed the maximum number of processes on the system allocated to this VM (`NumOfProcessesLim`)	77	Warning
Kernel: Attempt to exceed the maximum number of threads on the system allocated to this VM (`NumOfThreadsLim`)	78	Warning
Kernel: Attempt to exceed the maximum number of POSIX timers available through `timer_create()` on the system allocated to this VM (`NumOfTimersLim`)	79	Warning
Kernel: Attempt to exceed the maximum number of file descriptors that can be in use at the same time (`FsNumOfOpenFdsPerVmLim`)	80	Warning
Kernel: Attempt to exceed the total amount of system RAM allocated to this VM (`SysRamMemLim`)	81	Warning
Kernel: Attempt to exceed the maximum number of the `inodes` allowed in the file system (`FsNumOfInodesLim`) - obsolete	82	Warning
Kernel: Attempt to exceed the maximum number of record locks allowed in the file system (`FsNumOfRecordLocksLim`) - obsolete	83	Warning
Kernel: Attempt to exceed the maximum number of different named message queues to this VM (`NumOfMsgQueuesLim`)	84	Warning

Table C-2: System Error Code Table (Continued)

Kernel: Attempt to exceed the maximum number of the POSIX named and unnamed semaphores to this VM (`NumOfSemaphoresLim`)	85	Warning
Kernel: Attempt to exceed the maximum number of pipes that can be opened in this VM (`NumOfPipesLim`)	86	Warning
Attempt to exceed the maximum number of shared objects available to this VM (`NumOfSharedMemObjsLim`)	87	Warning
Kernel: Attempt to exceed the maximum number of signals allowed to this VM (`NumOfSignalsLim`) - obsolete	88	Warning
Kernel: Attempt to exceed the maximum number of fast ADA semaphores for this VM	89	Warning
Kernel: Attempt to exceed the maximum number of kernel timers for this VM	90	Warning
Kernel: Attempt to exceed the maximum number of named servers allowed to this VM	91	Warning
Kernel: Attempt to exceed the maximum number of opened named servers allowed to this VM	92	Warning
Kernel: Attempt to exceed the memory limit allowed for creating named servers in this VM	93	Warning
Kernel: Attempt to exceed the maximum number of mounted file systems allowed to this VM	94	Warning
Kernel: Attempt to exceed the maximum number of UNIX stream sockets allowed to this VM	95	Warning
Kernel: Internal kernel error at the resource allocation	96	Warning
Reserved for Kernel	97 -104	
Reserved for Flash Device Driver	105 - 109	
Reserved for RAM Test Device Driver	110 - 114	

Table C-2: System Error Code Table (Continued)

Reserved for Hardware Monitor Device Driver	115 - 122	
Reserved for ASL Device Driver	123 - 142	
Reserved for BOOT	143 - 174	
Reserved for BSP	175 - 184	
Unused	185 - 199	
Ethernet: Not found any supported device	200	Warning
Ethernet: DRM operation failed	201	Warning
Ethernet: Link is down	202	Warning
Ethernet: Transmit timeout	203	Warning
Ethernet: Receive sequence error	204	Warning
Ethernet: Error in received frame (dropping)	205	Warning
Ethernet: Receiver overrun	206	Warning
UVME: VMEbus posted write error	207	Warning
UVME: PCIbus posted write error	208	Warning
UVME: Supported device not found	209	Warning
UVME: DRM operation failed	210	Warning
Reserved for program device drivers	211 - 219	
SATA: Supported device not found	220	Warning
SATA: Device identification error / timeout on device / device issued interrupt without clearing busy	221	Warning
SATA: Partition table read error	222	Warning
SATA: IO error occurred (recovered)	223	Warning
SATA: IO error occurred (could not recover, send RST command to the device)	224	Warning
SATA: IO error occurred (driver could not recover it)	225	Warning
SATA: IO error occurred (recovered by driver)	226	Warning
SATA: Device does not clear BSY flag (never cleared)	227	Warning

Table C-2: System Error Code Table (Continued)

SATA: DRDY never set	228	Warning
SATA: DRQ never set	229	Warning
SATA: DRM operation failed	230	Warning
Reserved for program device drivers	231 - 234	Warning
Motherboard PENTXM2: HI2PCI bridge: unknown error posted, NMI received	235	Warning
Motherboard PENTXM2: HI2PCI bridge: PERR# assertion detected	236	Warning
Motherboard PENTXM2: HI2PCI bridge: master data parity error detected	237	Warning
Motherboard PENTXM2: HI2PCI bridge: received target abort	238	Warning
Motherboard PENTXM2: HI2PCI bridge: received master abort	239	Warning
Motherboard PENTXM2: HI2PCI bridge: received system error	240	Warning
Motherboard PENTXM2: HI2PCI bridge: detected parity error	241	Warning
Reserved for Motherboard	242 - 244	
Motherboard PENTXM2: DRM operation failed	245	Warning
Motherboard PENTXM2: Mother board driver installation error	246	Warning
Motherboard PENTXM2: CPU thermal status (THRM_STS) is set	247	Warning
Reserved for Motherboard	248, 249	
Reserved for program device drivers	250 - 255	

Table C-3: Generic VM System Error Code Table

Error Logical Name	Error Code Number	Severity
Reserved	0 - 3	
Unhandled SIGILL error	4	Overridable
Reserved	5 - 7	
Unhandled SIGFPE error	8	Overridable
Reserved	9	
Unhandled SIGBUS error	10	Overridable
Unhandled SIGSEGV error	11	Overridable
Reserved	12 - 14	
Software Watchdog monitor error (0)	15	Application- configurable
Software Watchdog monitor error (1)	16	Application- configurable
Software Watchdog monitor error (2)	17	Application- configurable
Software Watchdog monitor error (3)	18	Application- configurable
Software Watchdog monitor error (4)	19	Application- configurable
`syslog()` messages	20	Warning
Unused	21 - 224	
Library subset error	225	VM Fatal
Unused	226 - 255	