PowerAPI_Error_Logging

Table of Contents

Overview

The PowerAPI implementation contains an embedded logging system for use with application debugging, failure analysis, and general tracing.

The logging system supports automatic log file rotation to prevent overfilling the file system, and is built around ring buffers and background write-to-file to maximize both debugging usefulness and library performance.

Logs are both multi-thread and multi-process safe.

Logging is turned on by default, and will generate traces in the log in the event of errors.

Default Log Parameters

Application Log		
Parameter	**Default Value**	**Environment Variable**
log file path	/var/opt/cray/powerapi/log/powerapi.log	PMLOG_FILE_PATH
maximum file size	1 MiB	PMLOG_MAX_FILE_SIZE
maximum file count	5	PMLOG_MAX_FILE_COUNT
ring buffer size	256 KiB	PMLOG_RING_SIZE
ring buffer count	2	PMLOG_NUM_RINGS
stderr debug level	0	PMLOG_DEBUG_LEVEL
stderr trace level	0	PMLOG_TRACE_LEVEL
Powerapid Service Daemon Log		
log file path	/var/opt/cray/powerapi/log/powerapid.log	*none*

All log parameters can be modified as indicated by an application environment variable, or by direct calls to the logging API functions.

Multiple processes and threads can safely interleave log messages to the same file without corruption.

Automatic log rotation occurs when the log file size exceeds (approximately) the maximum file size. No attempt is made to resolve race conditions between threads or processes, so the actual file size may grow slightly larger than the specified maximum before rotation occurs. Rotation moves logfile to logfile.1, logfile.1 to logfile.2, and so forth, up to the maximum log file count (N). logfile.N is discarded.

Setting any of the environment variables to zero is the same as unsetting them, which causes the default values to be used.

If PMLOG_MAX_FILE_SIZE is set to -1, automatic log rotation is prevented, though log rotation can still be forced using direct calls to the logging API.

If PMLOG_MAX_FILE_COUNT is set to -1 (or 1), then only one log file is allowed: the PMLOG_MAX_FILE_SIZE value is ignored, and log rotation is completely prevented.

Note that disabling log rotation will allow logs to grow without limit. Any desired log rotation must be performed using external utilities.

If PMLOG_RING_SIZE is set to any non-zero value less than 4096, the ring buffer size will be 4096 bytes.

If PMLOG_NUM_RINGS is set to -1, ring buffers are disabled entirely.

The ring buffers are described in the section on **Normal Log Behavior**.

The stderr debug and trace levels are described in the section on **Stderr Log Behavior**.

Log Output Format

Example:

```
2017/03/23-14:17:58.850386 pwrcmd 42093 42093 TRC1 [PWR_GetMajorVersion:42] [ENTER]
2017/03/23-14:17:58.850387 pwrcmd 42093 42093 TRC1 [PWR_GetMajorVersion:44] [EXIT]
major = 1
2017/03/23-14:17:58.850391 pwrcmd 42093 42093 TRC1 [PWR_GetMinorVersion:51] [ENTER]
2017/03/23-14:17:58.850392 pwrcmd 42093 42093 TRC1 [PWR_GetMinorVersion:53] [EXIT]
minor = 4
2017/03/23-14:17:58.877985 pwrcmd 42093 42094 LOGR [_thr_rotate_logs:1956] Rotating
log files
```

Fields (space-separated):

Field	Content
1	Timestamp (YYYY/MM/DD-hh:mm:ss.usec)
2	Application name (argv[0] basename)
3	Process ID
4	Thread ID
5	Log message type
6	Function and line number
7+	Free-form text message

Log message types:

Type Name	Sync	Write-thru	Flush	Console	Meaning
CONS	yes	no	no	yes	Console message, delivered only to the system console
LOGR	no	yes	no	no	Internal logging message (e.g. starting a log rotation)
MESG	no	yes	no	no	Informational message
CRIT	no	no	yes	yes	Last message output before the library calls exit()
FAIL	no	no	yes	yes	Error occurred, associated with library call failure return

WARN	no	no	yes	yes	Recovered or ignored error with no library call failure return
DBG1	no	no	no	no	Debugging message
DBG2	no	no	no	no	Developer debugging message
TRC1	no	no	no	no	PowerAPI library call entry/exit
TRC2	no	no	no	no	Internal library call entry/exit
TRC3	no	no	no	no	Deep-internal library call entry/exit

Normal Log Behavior

Logging behavior is significantly different from traditional logging.

All message types except the console message are asynchronous: they are buffered and delivered to a background thread for writing to the output device, allowing the application to continue immediately after buffering.

Messages are treated as either write-through, or ring-buffered. A write-through message is delivered directly through the write thread to the log file. Only the LOGR messages (generated by the write thread itself) and the MESG types are written through. All others are buffered in the ring buffer. In general, new entries in the ring buffer overwrite the oldest entries in the ring buffer, so if the rings are never flushed, no output will appear in the log file. When the ring is flushed, ALL messages in ring are flushed to the log file, culminating in the message that triggered the flush. This also clears the ring buffer.

Consequently, if no errors occur (no CRIT, FAIL, or WARN message types are issued), no information will appear in the log file. If an error occurs, ALL messages stored in the ring buffer – including all debugging and trace information leading up to the error – will be flushed to the log file. Note also that the error message itself is also delivered to the console, but without the debugging trace leading up to the error.

There is no "log level" setting, and no filtering of messages, under the presumption that in stable code, errors will be rare. Because the message type is encoded in the logged message, filtering can be done after the fact on the log file itself.

Progress monitoring or telemetry should be performed exclusively with MESG type messages, which is always delivered directly to the log file. Note that MESG entries may appear out of time-order in the logs, since they are immediately delivered to the log: if a subsequent error occurs, it may flush the ring, which may contain debug messages generated *before* the MESG was logged, but which are actually flushed to the log *after* th e MESG file was written.

The size and count of ring buffers can be modified using environment variables (described above) or direct logging API calls, if required. In general, the ring size should be at least large enough to cover all debug messages that might lead up to a failure. The number of ring buffers can be increased under specialized conditions: basically, the more rings you allocate, the more consecutive errors can occur before logging will begin to slow the application waiting for log file system writes to complete. If you have continuous errors, the ring buffers will be continuously flushing, and the application will be forced to slow to file system speeds, regardless of the number of rings. So generally, two is sufficient.

If the ring buffer count is specified as -1, ring buffers are disabled. Errors and console messages will still be delivered to the console, but will not be logged to the log file. Only LOGR and MESG messages will be logged in the log file.

Stderr Log Behavior

For active development, it can be beneficial to see continuous logging to the terminal window. This can be enabled in a more traditional manner by specifying independent debug and trace levels, using environment variables or direct calls to logging API functions.

PMLOG_DEBUG_LEVEL can be set to 1 to continuously display all messages except DBG2 (and TRC* messages) to stderr as the application runs, or to 2 to include DBG2 messages as well.

PMLOG_TRACE_LEVEL can be set to 1 to continuously display all messages except TRC2 and TRC3 (and DBG* messages) to stderr as the application runs, or to 2 to include TRC2 messages as well, or to 3 to include TRC2 and TRC3 messages as well.

This trace can be logged to a file by redirecting stderr. One advantage of doing this is that it will be unaffected by log rotation.

Application Programming Interface

(TBD)