Table of Contents

Mount Everest

libmtev version 1.10.2

Originally developed as a part of the Reconnoiter alerting system, libmtev evolved as a framework well-suited for building high performing server applications in C.

libmtev Application Configuration

Amongst other things, libmtev provides a robust configuration system that is based on simple files and XML. Unlike other XML-based systems libmtev forgoes the XML religion and uses a non-validated (no DTD, no relax-ng) "fast and loose" approach to XML configurations. The configuration system allows for powerful application-defined semantics by leveraging XPath for querying the configuration, but provides simple APIs for retrieving configuration settings. The only requirement is that the root node be named for the application.

Several of libmtev's shipped subsystems including the eventer, logging, clustering, network listeners and the module system rely on the configuration system. Various compenents are good at getting their parts from the config and ignore stuff they don't understand or know about making the system trivially extensible to support large, custom and complex application configurations if required.

The configuration file supports includes and a directory-based backing store (for configurations that are too large and/or update too often) to enhance simple XML files. The configuration can be updated at runtime and the new modified config written back to the original location allowing for persistent runtime-updateable configuration.

While not required, it is considered best practice to inherit attributes from parent nodes. This is accomplished via XPath in all of the existing subsystems. As the XML configuration system allows for arbitrary node names it allows operators to build configuration files that make sense for their deployments.

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<example1 lockfile="/var/run/example1.lock">
</example1>
```

lockfile

If the `lockfile` attribute is specified on the root node, libmtev will require and lock the specified file to prevent multiple invocations of the application running at once.

require_env

Via the `mtev_conf_env_off(mtev_conf_section_t node, const char *attr)` function, applications may choose to ignore certain nodes. The default `attr` (when NULL is specified) is `require_env` .

libmtev itself applies this, in its default form, to listeners, capabilities, logs, and modules.

Environmental controls support existence checking, equality checking and PCRE matching. Negation is accomplished by leading the expression with an exclamation mark: `!` .

- **`<var>`**

  Example: `"FOO"`

  Action: require that the "FOO" environment variable be set in order for the given node to be considered active.

- **`<var>=<val>`**

  Example: `"!FOO=42"`

  Action: require that the "FOO" environment variable **must not** (note the leading `!`) be set and equal to 42 in order for the given node to be considered active.

- **`<var>~<regex>`**

Example: `"FOO~^(?i)disabled_"`

Action: require that the "FOO" environment variable be set and begin with the case-insensitive string "disabled_" in order for the given node to be considered active.

Unlike other attribute inheritance within mtev_conf, the `mtev_conf_env_off` function will apply all ancestral `require_env` attributes during enforcement (including the node in question.) This allows nesting of more complex "stacked" requirements.

Inheritance

Unless otherwise specified, elements and attributes are inherited from all ancestors. As is typical with inheritance, the youngest/deepest value overrides values of ancestors. If the value requested is a set of key/value pairs, the top-most ancestor is queried, then its child is queried, merging values, replacing conflicts and so on until the youngest/deepest node is reached.

The C API `mtev_conf_get_hash` and `mtev_conf_get_namespaced_hash` implement this functionality; as long as those functions are used to convert configuration stanzas into hashes for internal use, the developer and the operator get all the advantages of sophisticated configuration reuse.

When attributes are extracted, developers should use the the XPath practice of `ancestor-or-self::node()/@name` for an attribute named `name` .

Simple (implicit) inheritance

simple.conf (snippet)

```
<a foo="bar">
  <config>
    <key1>val a 1</key1>
    <key2>val a 2</key2>
  </config>
  <b quux="baz">
    <config>
      <key1>val b 1</key1>
      <key3>val b 3</key3>
    </config>
    <c foo="meme" />
  </b>
</a>
```

When looking at the "foo" attribute we see the following values at nodes:

```
* at `a`, foo="bar"
* at `b`, foo="bar"
* at `c`, foo="meme"
```

When looking at the "quux" attribute we see the following values at nodes:

```
* at `a`, foo=(none)
* at `b`, foo="baz"
* at `c`, foo="baz"
```

When looking at the key/value set "config" we see the following values at nodes:

```
* at `a`, `{ key1: "val a 1", key2: "val a 2" }`
* at `b`, `{ key1: "val b 1", key2: "val a 2", key3: "val b 3" }`
* at `c`, `{ key1: "val b 1", key2: "val a 2", key3: "val b 3" }`
```

This inheritance model allows for "non-repetitive" configuration approaches: "express yourself once and reuse."

Complex (explicit) Inheritance

Sometimes it is useful to define a configuration key/value set for reuse, but the strict parent-child inheritance model is awkward. Under these circumstances, the explicit inheritance often solves the issue at hand. With explicit inheritance, a configuration can inherit from another named node elsewhere in the configuration.

The `<config>` stanzas (and others) can be identified, as is typical in XML, with the `id` attribute: `<config id="sample">. Additionally, any config may explicitly specify that it inherits from a named config by specifying the `inherit` attribute.

Any `<config>` , A, which has the `inherit` attribute will first inherit from its most direct parent, then supplement/replace those key/values with the configuration whose `id` attribute matches the `inherit` attribute of A, and finally supplement/replace those key/values with key/values directly beneath A. The entire tree is searched for a node whose `id` matches A's `inherit` attribute.

complex.conf (snippet)

```
<config name="A">
  <key1>a</key1>
  <key2>b</key2>
  <key3>c</key3>
  <config name="C" inherit="bob">
    <key1>AAA</key1>
    <key4>DDD</key4>
  </config>
</config>
<x>
  <y>
    <z>
      <config name="B" id="bob">
        <key2>bobB</key2>
        <key5>bobE</key5>
      </config>
    </z>
  </y>
</x>
```

The config named "A" contains:

```
* key1 = a
* key2 = b
* key3 = c
```

The config named "C" contains:

```
* key1 = AAA
* key2 = bobB
* key3 = c
* key4 = DDD
* key5 = bobE
```

It should be noted hat all config's that include the one named "B" above follows this same inheritance model.

Including configuration

For a variety of reasons, it can be desirable to include one configuration file from another. libmtev allows two forms of file include: normal and snippet.

Normal Includes

application.conf

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<root>
  <include file="child.conf"/>
</root>
```

child.conf

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<child>
  <foo></foo>
  <bar></bar>
</child>
```

Under normal includes, you can include a complete XML document at the point of the include. The `<include>` node is preserved, not replaced. The root node, `<child>` in this case, is "absorbed" and it's children are placed directly under the `<include>` node.

Any changes made to the runtime configuration within the `<child>` tree will be writte back to `child.conf` .

The tree above would look like

```
    root
      - include
        - foo
        - bar
```

Snippet Includes

Snippet includes act like normal includes except that the included file is treated as an XML snippet and thus there is no root node to "absorb." The elements in the snippet are placed directy under include.

application.conf

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<root>
  <include file="snippet.conf" snippet="true"/>
</root>
```

snippet.conf

```
<foo></foo>
<bar></bar>
```

The tree above would look like

```
root
  - include
    - foo
    - bar
```

Shatter

When the runtime configuration changes, the system will serialize those changes back to the containing XML files. For large and rapidly changing configurations this can be an overwhelming load on the system. libmtev provides a feature called "shatter" that allows any node (other than the root) to be annotated with a `backingstore` attribute that indicates a directory to which underlying nodes should be written out. Nodes and attributes are stored in separate files and when the configuraion is subsequently modified, only changed nodes need be added, deleted or updated.

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<root>
  <superbigtree backingstore="/path/to/etc/superbig"/>
</root>
```

Watchdog

The watchdog subsystem in libmtev provides a facility for hang protection, crash recovery, and crash reporting.

When "things go wrong" the parent (monitor) process will trace the monitored child using `{gilder} $pid $reason > {trace_dir}/$appname.$pid.trc` , ensure it is dead, confirm that it should restart it and then launch a new child.

example1.conf

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<example1 lockfile="/var/tmp/example.lock">
  <watchdog glider="/opt/local/bin/bt"
            tracedir="/var/traces/example"/>
</example1>
```

The following watchdog attributes are supported:

- **tracedir**

  A directory to deposit trace files. Trace files contain the output of the `glider` command. File names are of the format `{appname}.{pid}.trc` .

- **glider**

  The full path to an executable to invoke when a monitor process crashes or is killed due to inactivity. It is invoked with two arguments: process id and reason (one of "crash", "watchdog", or "unknown").

- **save_trace_output**

  Choose whether to store any output that the glider may produce on stdout. Its value is "true" or "false", and the default is "true" if not specified. If true, the glider output is saved as described in the `tracedir` attribute above. Some gliders may produce their own output files, in which case the stdout stream is unnecessary and one may choose to ignore it by setting this attribute to "false".

- **retries**

  The maximum number of restart attempts to be made over `span` seconds. Default: 5 retries over 60 seconds.

- **span**

  The number of seconds over which restarts are rate limited. Default: 5 retries over 60 seconds.

Managed Applications

The watchdog subsystem has the ability to manage sub-process that need to remain running. These are often called sidecar applications.

Applications listend in the `managed` section of the config will be started when running in managed mode and restarted upon failure. If the application exits normally with an exit code of 0, it is restarted immediately. In all other cases, an expoentential backoff is used in the restart sequence so as not to overwhelme the system.

Applications are found using the XPath `/example1/managed//application|/example1/include/managed//application` to pull all nested application declarations from underneath top-level `managed` sections (even when included).

example1.conf

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<example1 lockfile="/var/tmp/example.lock">
  <managed>
    <application exec="/opt/foo/sidecar1"
                 user="nobody" group="nogroup" dir="/var/run/sidecar1"
                 stderr="app/sidecar/stdout" stdout="app/sidecar/stderr" environment="true">
      <arg>-l</arg>
      <arg>localhost</arg>
      <env>TESTVAR=TESTVAL</env>
    </application>
  </managed>
</example1>
```

The following application attributes are supported:

- **exec**

  An executable to run, either a fully qualified path or name that is searched for in the PATH environment.

- **arg0**

  Specifies the first argument to `execve` is the program itself. If this option is not specified, the value of `exec` is used as arg0.

- **user**

  Specifies a user to `setuid` to. If this is not specified, it will default to the user specified to the application (usually via a `-u` argument) which is passed into `mtev_main()` .

- **group**

  Specifies a group to `setgid` to. If this is not specified, it will default to the group specified to the application (usually via a `-g` argument) which is passed into `mtev_main()` .

- **dir**

  If specified, the managed application will `chdir` to this directory prior to `exec` .

- **environment**

  Specified whether the parents environment should be copied into the manage application. The default is true.

- **stdout**

A name of a log facility to channel stdout (FD 1) to. The default is `error` .

- **stderr**

  A name of a log facility to channel stderr (FD 2) to. The default is `error` .

Arguments are passed to the application in order by specifying `<arg>` elements within the `<application>` configuration.

Environment variables can be added using `<env>` elements within the `<application>` configuration. If no `=` sign is found in the value of the specified `<env>` text, then value is used as a key and that environment variable value is extracted from the parents environment and pushed into the applications environment. This is useful in combination with `environment="false"` to pass through on specific environment variables.

Eventer

The libmtev eventer can be configured through a top-level `<eventer>` configuration block.

```
<application>
  <eventer [ implementation="..." ]>
    <key>value</key>
    ...
  </eventer>
</application>
```

The `implementation` attribute is optional and must be supported on the platform; it is recommended that one omit this from configurataions. Valid values are `epoll` , `kqueue` , and `ports` .

The keys and values supported are:

- **debugging**

  If this key is present and the value is anything other than "0", eventer debugging facilities will be enabled. This can be slow and should not be used in production.

- **show_loop_callbacks_threshold**

  Specify a millsecond threshold for logging of "slow" callbacks in eventer loops. The default is 0 (all are logged), -1 disables logging. Logging is sent to a log stream called `debug/eventer/callbacks/loop/<loopname>` .

- **show_jobq_callbacks_threshold**

  Like `show_loop_callbacks_threshold` , but for callbacks run in asynchronous job queues. Logging is sent to a log stream called `debug/eventer/callbacks/jobq/<queuename>` .

- **rlim_nofiles**

  Specified the number of file descriptors desired. libmtev will attempt to up the operating system limits to allow the application to open this many files. The specified value must be at least 256. If not specified, a default value of 1048576 will be used.

- **default_queue_threads**

  This specified the number of threads that should be used to manage the default asynchronous job queue. If not specified, a value of 10 is used.

- **concurrency**

  The number of event loop threads to start. This effects the concurrency of the default event loop and sets the default of any non-default event loops. If not specified or specified as 0, the library will attempt to make a reasonable guess as to a good concurrency level based on the number of CPU cores in the system.

- **loop_<name>**

  This establishes a new named event loop and sets its concurrency and watchdog timeout to the provided values. The format is: `concurrency[,timeout_in_seconds]`

  If a concurrency value of 0 is provided, then the named event loop will use the default concurrency specified by the `concurrency` key. Floating point notation can be used to specify subsecond or partial second timeouts. If unspecified or specified as 0, the timeout will default to the global setting (which defaults to 5.0 unless overriden by the application).

- **jobq_<name>**

  This establishes a new named jobq and sets parameters around concurrency and memory safety. The format of the value is: `concurrency[,min[,max[,safety[,backlog[,lifo]]]]]` . Concurrency, min, max, and backlog are all unsigned integers. Concurrency must be greater than zero. If minimum is omitted or zero, no minimum is set. If max is omitted, it is set to min. If max is zero, there is no maximum. Safety can be one of `none` , `cs` , or `gc` (default). For more information om memory settings see eventer_jobq.h and mtev_memory.h. Backlog sets the advisory queue length backlog limit for the queue. The `lifo` setting instructs the jobq to process jobs last-in-first-out. The values for this field is either `lifo` or `fifo` .

  > Note that this merely creates the jobq. One must find and use it programmatically within the software. It is designed to have a code-free way of allowing operators to adjust parameters around jobqs used within an application.

- **default_jobq_ordering**

  Specifies the default job consumption order for jobqs that don't explicitly declare an ordering requirement. Valid values are `lifo` and `fifo` . If unspecified, the default for this setting is FIFO.

- **default_ca_chain**

  Specified the path to a file containing a PEM encoded set of certificate authorities to trust by default.

- **ssl_ctx_cache_expiry**

  Sets the default number of seconds after which cached SSL contexts will be released. The default is 5 seconds.

- **ssl_dhparam1024_file & ssl_dhparam2048_file**

  Sets the filename to cache generated DH params for SSL connections. If the keys are omitted, no cahce will be used and new parameters will (likely needlessly) be regenerated during startup. If specified and empty, DH params will not be generated and this be unavailable to the SSL subsystem; this will prevent forward secrecy.

Logging

The logging system within libmtev represents a directed acyclic graph of input-output chains. Each node in the graph has a unique name and is called a "log_stream." Log_streams without a `type` attribute have output to downstream nodes ("outlets"). Nodes with a `type` attribute have additional output characteristics (like outputting to a file).

Upon startup, the system will establish several built-in log_streams, only one of which has a type. The "stderr" log_stream has a type of `file` and an output filedescriptor of 2. Other log_stream are setup and and configured to have the "stderr" log_stream as their outlet. These log_streams are called: "error", "notice", "debug." The correspond to the global logging symbols in the C API: `mtev_stderr` , `mtev_error` , `mtev_notice` , and `mtev_debug` , respectively. For more information on logging via the API, see the development section of this documentation relaated to logging. The "debug" log_stream is disabled by default.

Logs are hierarchical in nomenclature as a convenience. If, in your code, you request a log named "error/foo" and no such log exists in the configuration, a new untyped log will be created and its outlet will be set to "error". This is recursive, so "debug/myapp/facility1" will (unless configured otherwise) outlet to "debug/myapp" which will outlet to "debug." This makes it very simple to semantically separate logs into new error and debugging facilities without worrying about them being lost, while providing the flexibility to configure where things go if other outcomes are desired.

All logging configuration exists within the top-level XML node `<logs>` . Individual log_streams are declared using `<log>` stanzas and outlets are declared using `<outlet>` stanzas. A log_stream uses all `<outlet>` stanzas that are its direct child or direct child of any ancestor node.

application.conf

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<application>
  <logs>
    <log name="internal" type="memory" path="10000,1000000" require_env="MEMLOG"/>
    <log name="logfile" type="file" path="/var/log/app.log"
         rotate_bytes="10000000" retain_bytes="50000000" timestamps="on"/>
    <log name="http/access" type="jlog" path="/var/log/app-http.feed(*)"/>
    <console_output>
      <outlet name="stderr"/>
      <outlet name="internal"/>
      <outlet name="logfile"/>
      <log name="error"/>
    </console_output>
    <components>
      <error>
        <outlet name="error"/>
        <log name="error/example"/>
        <log name="error/sample"/>
      </error>
      <debug>
        <outlet name="debug"/>
        <log name="debug/example" disabled="true"/>
      </debug>
    </components>
  </logs>
</application>
```

Let's walk through this sample file to understand what's going on. First there are seven `<log>` stanzas establishing log_streams named "internal", "logfile", "http/access", "error", "error/example", "error/sample", and "debug/example".

Starting at the end, the "debug/example" log_stream is declared in a disabled state via the `disabled="true"` attribute. If walk it and its anscestors we find one `<outlet name="debug"/>` child. This log_stream has no type, so any messages sent into this log are only output to its outlet "debug." You'll notice that no log_stream named "debug" is declared. We rely on the built-in "debug" log which is setup to output to "stderr". Also, because we did not declare a "debug" log_stream with `disabled="false"` , the default state remains disabled.

The "error/example" and "error/sample" log_stream are similarly configured to output to the "error" log_stream as its outlet. But, we've declared the "error" log_stream in this configuration so that we can manipulate its outlets. The `<components>` , `<error>` , and `<debug>` nodes have no special meaning by name; they are simply used as descriptive hierarchical containers to allow us to share outlet configuration and to logically isolate our intentions.

The "error" log_stream already exists as a built-in log_stream. The declaration here is used to set outlets to the three log_streams named: "stderr", "internal", and "logfile". As before, we use an arbitrarily named node to contiain the declaration logically; this time called `<console_output>` .

The "internal" log_stream is of type `memory` which uses an in-memory ring buffer to store recent log lines. We have a limit ot 10000 log lines and 1000000 bytes. It is also only active if the environment variable INMEM is set.

The "logfile" log_stream is of type `file` and will auto-rotate files as they hit 10 million bytes and delete old log files as the cumulative space consumed exceeds 50 million bytes. Timestamps are turned on for this log_stream.

The "http/access" log_stream is of type `jlog` which is create a Jlog journaled log for external consumption.

Generic Attributes

- **require_env**

  This optionally requires conditions around an environment variable. See `require_env` .

- **debug**

  If "on"/"true", additional debugging information (like thread ID) is injected into logged lines.

- **facility**

  If "on"/"true", the name of the log is injected into logged lines.

- **timestamps**

  If "on"/"true", timestamps are injected into logged lines.

- **disabled**

  If "on"/"true", the stream is disabled and attempts to log to the facility will result in a single branch instruction.

- **format**

  Can be set to `plain` , `flatbuffer` , or `json` . The default is `plain` . This option impacts logs that write output (those with the `type` field set).

Log Types

memory

The memory log_stream type establishes an internal ring buffer in memory. There are APIs (including REST endpoints) to retrieve the contents of this ring buffer. Additionally, if the process crashes one can examine the contents of the ring buffer with a debugger.

- **path**

The `path` attribute takes two numbers comma separated. The first number is the maximum number of log lines to be retained. The second number is the maximum number of bytes to be retained. The implementation will not exceed either limit.

file

The file log_stream type is used to drive writing to ordinary files using the POSIX API. It provides both time-based and size-based retention management capabilities.

- **path**

  The path is the filename to which log data should be written.

- **rotate_seconds**

  Specifies how many seconds of log data should be written into a file before it is moved aside and a new file is started. (used with `retain_seconds` and not with `rotate_bytes` or `retain_bytes`).

- **retain_seconds**

  Specified the number of seconds of data to be retained. If all log data in a rotated file are older than this value, the file will be removed. (used with `rotate_seconds` and not with `rotate_bytes` or `retain_bytes`).

- **rotate_bytes**

  Specifies how many bytes of log data should be written into a file before it is moved aside and a new file is started. (used with `retain_bytes` and not with `rotate_seconds` or `retain_seconds`).

- **retain_bytes**

  Specified the number of bytes of data to be retained. If all log data in a in all rotated files exceed this value, the oldest file will be removed. (used with `rotate_bytes` and not with `rotate_seconds` or `retain_seconds`).

jlog

The jlog log_stream type implements an log output to the Jlog multi-file journalled logging format. Jlog is a segmented write-ahead log that is fast and efficient and supports multiple subscribers with independently maintained process checkpoints.

- **path**

  The path is the Jlog directory to be used. It may optionally be ended with a parenthesized subscriber name. If a name (other than "*") is provided, a subscriber of that name will be added to the Jlog on creation.

jlog config

The jlog can be further configured using a `<config>` stanza supporting the following options:

- **segment_size**

  Set advisory segment size for the jlog in bytes. By default jlog uses 4Mb. There is a hard-coded limit of 1Gb. If the specified value is out of range, not changes are made to the existing jlog.

- **precommit**

  Specify a precommit buffer size in bytes. The default is 0 and the maximum is 8Mb. If the specified value is too large, 8Mb is used.

Built-in logging facilities

libmtev uses its own logging, so applications have ample error and debugging information exposed out of the box. There are three four built-in logging facilities that serve as the base for most others: `stderr` , `error` , `notice` , and `debug` . By default, `debug` is disabled and both `error` and `notice` outlet `stderr` .

The following log streams are used within libmtev:

`debug`

Generic debug logging, by default all `debug/*` logs flow through here as an outlet.

`debug/amqp`

Debugging output from the amqp module.

`debug/cluster`

Debugging output from mtev clustering.

`debug/conf`

Debugging output from the configuration system.

`debug/consul`

Debugging output from the consul module.

`debug/consul/curl`

Debugging output from curl operations in the consul module.

`debug/dwarf`

Debugging information from the internal dwarf analyzer.

`debug/eventer`

Debugging information from the eventer subsystem.

`debug/fq`

Debugging information from the fq module.

`debug/http`

Debugging information from the http service framework.

`debug/http2`

Debugging information from the http/2 service framework.

`debug/http_observer`

Debugging information from the http_observer module.

`debug/listener`

Debugging information from the listener subsystem.

`debug/lua`

Debugging information from the lua modules lua_general and lua_web.

`debug/memory`

Debugging information from the memory subsystem, specfically around safe memory reclamation.

`debug/rest`

Debugging information from the rest subsystem (sitting atop the http and http/2 service frameworks).

`debug/reverse`

Debugging information from the reverse connection subsystem.

`debug/time`

Debugging information from the time subsystem, particularly around timings and thread affinity.

`debug/websocket_client`

Debugging information from the websocker integration atop the http service framework.

`debug/xml`

Generic debug capture from the libxml2 framework.

`debug/zipkin_fq`

Debugging information from the zipkin_fq module.

`debug/zipkin_jaeger`

Debugging information from the zipkin_jaeger module.

`error`

Generic error logging, by default all `error/*` logs flow through here as an outlet.

`error/amqp`

Error logging for the amqp module.

`error/cluster`

Error logging for clustering operations.

`error/conf`

Error logging for the configuration subsystem.

`error/consul`

Error logging for the consul module.

`error/eventer`

Error logging for the eventer subsystem.

`error/fq`

Error logging for the fq module.

`error/http_observer`

Error logging for the http_observer module.

`error/listener`

Error logging for the listener subsystem.

`error/lua`

Error logging for the lua modules lua_general and lua_web.

`error/rest`

Error logging for the rest subsystem (sitting atop the http and http/2 service frameworks).

`error/reverse`

Error logging for the reverse connection subsystem.

`error/websocket_client`

Error logging for the websocker integration atop the http service framework.

`error/zipkin_jaeger`

Error logging for the zipkin_jaeger module.

`notice`

Informational output.

`http/access`

HTTP access logs (close to the Apache2 common log format).

`http/io`

Debugging information for I/O performed in the http and http/2 service frameworks.

`stderr`

A logging facility that writes to file descriptor 2.

Listeners

Network listeners and their services are specified via configuration.

application.conf (snippet)

```
<listeners>
  <sslconfig>
    <optional_no_ca>false</optional_no_ca>
    <certificate_file>/path/to/server.crt</certificate_file>
    <key_file>/path/to/server.key</key_file>
    <ca_chain>/path/to/ca.crt</ca_chain>
    <layer>tlsv1:all,!sslv2,!sslv3,cipher_server_preference</layer>
    <ciphers>EECDH+AES128+AESGCM:EDH+AES128+AESGCM:!DSS</ciphers>
  </sslconfig>
  <consoles type="mtev_console" require_env="MTEV_CONTROL">
    <listener address="127.0.0.1" port="32322">
      <config>
        <line_protocol>telnet</line_protocol>
      </config>
    </listener>
  </consoles>
  <web type="control_dispatch" address="*">
    <config>
      <idle_timeout>30000</idle_timeout>
      <document_root>/path/to/docroot</document_root>
    </config>
    <listener port="80" />
    <listener port="443" ssl="on" />
  </web>
</listeners>
```

This example demonstrates many powerful concepts of the libmtev configuration system. There are three listener stanzas nested above and we'll walk through each. The first is the `<listener address="127.0.0.1" port="32322">`. With this, you can telnet to 127.0.0.1 port 32322 and talk with your libmtev application. The console is extensible so you can add application-specific command, control, and interrogation capabilities.

This listener has a `<config>` stanza underneath it that sets `line_protocol` to `telnet`. `line_protocol` is a configuration option for listeners of type `mtev_console`. You'll note that the listener's `type` attribute was actually set in a parent node. Most systems in libmtev will recusively merge from ancestors down to the a specimen node and use that result. Here `type` is simply an attribute, so merging is just replacing. This node also has an `sslconfig`, but it doesn't use it, so we'll ignore that for now. The `require_env` attribute requires the `MTEV_CONTROL` environment variable to be set for this listener to be active; if unspecified, it is active.

The next two listener stanzas are for port 80 and 443. They are in a `web` node that has both `type` and `address` attributes set (those are inherited by the listeners). The `config` node (child of `web`) and the `sslconfig` node (child of `listeners`) are also inherited into the `listener` nodes. The `config` is arbitrary and passed into the listener. The `sslconfig` is passed into the ssl subsystem and is uniform across all listener types.

The following attributes are supported for listeners:

- **type**

  The type of listener simply references a named eventer callback in the system (one registered with `eventer_name_callback(...)`. libmtev support four built-in listener types: `http_rest_api`, `mtev_wire_rest_api/1.0`, `control_dispatch`, and `mtev_console`. Applications can arbitrarily extend the system by naming callbacks.

- **require_env**

This optionally requires conditions around an environment variable. See `require_env` .

- **address**

  The address is either a filesystem path (AF_UNIX), an IPv4 address or an IPv6 address. The type is intuited from the input string. If the special string `*` or `inet:*` is used, then the IPv4 `in_addr_any` address is used for listening. IF `inet6:*` is used, then the IPv6 `in_addr_any` address is used for listening.

- **port**

  Specifies the port on which to listen. This has no meaning for AF_UNIX-based addresses.

- **ssl**

  If the value here is `on` , then the socket passes through SSL negotiation before handed to the underlying system driving the specified listener type.

- **fanout**

  If the value here is `on` , the new events created for accepted connections will be fanned out across threads in the event pool owning the listening socket (usually the default event pool). A different pool can be selected by additionally supplying `fanout_pool` .

- **fanout_pool**

  If `fanout` is `on` , this will select a named pool on which to distribute new connection events. The value of this attribute should be the name of an event pool. If not pool exists with the specified name, the pool containing the listening event will be used.

- **accept_thread**

  If `accept_thread` is `on` , a new dedicated thread will be spawned to handle accepting new connections in a blocking fashion.

- **no_delay**

  If `no_delay` is `off` or `false` , then `TCP_NODELAY` will not be activated on the accepted socket. The default is `on` .

- **idle_timeout**

  Specifies a time in milliseconds afterwhich if the connection remains idle (no read or write traffic) it will be terminated. The protocol driver must cooperate programmatically to inform the system of such activity; the `mtev_console` and `http` protocols do this.

Each listener can access the `config` passed to it; see type-specific documentation for other config keys.

sslconfig

The ssl config allow specification of many aspects of how SSL is negotiated with connecting clients. SSL config supports the follwing keys:

- **layer**

  This specifies the SSL protocol options we present and is the form `<protocol>[:<option>,[<option>[,...]]]` . Options may be negated with an antecedent `!` . Tokens are matched case-insensitively.

  Protocols supported (depending on openssl): `SSLv2` , `SSLv3` , `TLSv1` , `TLSv1.1` , `TLSv1.2` .

  Options supported (depending on openssl): `SSLv2` , `SSLv3` , `TLSv1` , `TLSv1.1` , `TLSv1.2` , `cipher_server_preference`

The default layer string is `tlsv1:all,!sslv2,!sslv3`

- **certificate_file**

  Specifies the path to a PEM encoded certificate file.

- **key_file**

  Specifies the path to a PEM encoded key file. It must not be encrypted with a password.

- **ca_chain**

  Specifies the CA chaing file (PEM encoded) that should be used to validate client supplied certificates.

- **crl**

  Specifies a PEM encoded certificate revocation list file. If not specified, no revocation is enforced.

- **ciphers**

  Specifies which ciphers should be supported, expressed in the OpenSSL cipher list format. Check the OpenSSL manual for more details. If not specified, the default ciphers supported by the OpenSSL library are used.

- **npn**

  Specifies which NPN (next-protocol-negotiation) to offer. If omitted, `h2` is used and the http2 protocol is exposed. Specifying `none` will disable this NPN registration.

Dynamically Loadable Modules

The libmtev library supports loading dynamically loadable modules that can provide optional features to an appliction or change the behavior of exisitng code via hooks.

There are two types of modules in the core libmtev sytems "generics" and "loaders." Loaders know how to load generics. The only built-in module is the "C" loader which knows how to load architecture-appropriate shared objects.

There are several modules that ship with libmtev (though engineers can build more as a part of their application design). These modules are described in the modules section of the manual.

Modules are all configured under the top-level `<modules>` node in the configuration.

application.conf

```
<?xml version="1.0" encoding="utf8" standalone="yes"?>
<application>
  <modules directory="../modules">
    <generic image="zipkin_fq" name="zipkin_fq" require_env="FQ">
    </generic>
    <generic image="lua_mtev" name="lua_general">
      <config>
        <directory>../modules/lua-support/?.lua;./lua-examples/?.lua;{package.path}</directory>
        <cpath>../modules/mtev_lua/?.so;{package.cpath}</cpath>
        <lua_module>luatest</lua_module>
        <lua_function>onethread</lua_function>
      </config>
    </generic>
  </modules>
</application>
```

The `<modules>` node takes an optional `directory` attribute that specified where dynamic modules should be found on the filesystem. If omitted, the directory in which modules were installed as a part of your libmtev install will be used. Typically, this attribute is omitted unless you are developing new modules. The attribute acts as a search path and both ':' and ';' can be used as separators between directory entries. If the module cannot be loaded from any of the specified directories, the loader will attempt a fallback to the installation's default module directory.

Like other parts of the sytem `<config>` blocks are ancestrally merged.

The `<generic>` stanzas instruct the system to load a module. The above config loads the `zipkin_fq` module from the `zipkin_fq` binary image (`zipkin_fq.so` on ELF systems and `zipkin_fq.bundle` on mach-o systems like Mac OS X) with no configuration if an only if the FQ environment variable is set. It also loads the `lua_general` module from the `lua_mtev` binary image with a configuration.

Generic Options

- **require_env**

  This optionally requires conditions around an environment variable. See `require_env` .

- **image**

  The name of the shared object (or bundle on Mac OS X) that will by dynamically loaded into the system.

- **name**

The name of the symbol that will be used to identify the module or loader within the image.

Loader Modules

Loader modules know how to load other types of modules. The only loader that ships with libmtev is the C loader. However new loaders can be implemented by libmtev consumers.

Generic Modules

Generic modules can change the way the system behaves by interfacing with various hooks within the software stack.

amqp

The amqp module consumes and publishes message via AMQP (RabbitMQ).

- **loader**: C
- **image**: amqp.so

Module Configuration

- `poll_limit` (optional) [default: `10000`]

  allowed: `/^\d+$/`

  Maximum number of messages to handle in a single callback.

Examples

Loading the amqp module.

```
<root>
  <modules>
    <module image="amqp" name="amqp"/>
      <config>
        <poll_limit>1000</poll_limit>
      </config>
  </modules>
  <network>
    <mq type="amqp">
      <host>localhost</host>
      <port>8765</port>
      <user>user</user>
      <pass>pass</pass>
      <exchange>exchange</exchange>
      <routingkey>foo.*</routingkey>
    </mq>
    <mq type="amqp">
      <host>localhost</host>
      <port>8765</port>
      <user>user</user>
      <pass>pass</pass>
      <exchange>exchange</exchange>
      <routingkey>bob.#</routingkey>
    </mq>
  </network>
</root>
```

consul

A service registration and config integration for Consul agent.

- **loader**: C
- **image**: consul.so

Module Configuration

- `boot_state` (optional) [default: `passsing`]

  allowed: `/^(?:passing|warning|critical)$/`

  Set the initial state of service registration.

- `kv_prefix` (optional)

  allowed: `/^.*$/`

  Set an option directory prefix for loading keys from consul's KV store.

- `bearer_token` (optional)

  allowed: `/^.*$/`

  Set a bearer token for interactions with consul (to satisfy Consul ACLs).

Examples

Loading the http_observer module.

```
<app>
  <modules>
    <generic image="consul" name="consul"/>
  </modules>
  <consul>
    <service>
      <myservice id="{app}-{node}" port="12123">
        <check deregister_after="10m" interval="5s" HTTP="/url"/>
        <weights passing="10" warning="1"/>
        <tag>foo</tag>
        <tag>bar:baz</tag>
        <meta>
          <key>value</key>
        </meta>
      </myservice>
    </service>
  </consul>
</app>
```

fq

The fq module consumed and publishes message via fq.

- **loader**: C
- **image**: fq.so

Module Configuration

- **`poll_limit`** (optional) [default: `10000`]

  allowed: `/^\d+$/`

  Maximum number of messages to handle in a single callback.

Examples

Loading the fq module.

```
<root>
  <modules>
    <module image="fq" name="fq">
      <config>
        <poll_limit>1000</poll_limit>
      </config>
    </module>
  </modules>
  <network>
    <mq type="fq">
      <host>localhost</host>
      <port>8765</port>
      <user>user</user>
      <pass>pass</pass>
      <exchange>exchange</exchange>
      <program>prefix:"in."</program>
    </mq>
    <mq type="fq">
      <host>localhost</host>
      <port>8765</port>
      <user>user</user>
      <pass>pass</pass>
      <exchange>exchange</exchange>
      <program>prefix:"in2."</program>
    </mq>
  </network>
</root>
```

http_hmac_cookie

The http_hmac_cookie provides a safe way to persent authentication information in http sessions.

- **loader**: C
- **image**: http_hmac_cookie.so

Module Configuration

- `key` (optional)

  allowed: `/^.+$/`

  A hex encoded key, if none is specified a random key will be generated.

- `max_age` (optional) [default: `86400`]

  allowed: `/^\d+$/`

  The number of seconds the cookie will be valid for (default 1 day).

- `domain` (optional)

  allowed: `/^.+$/`

  The "Domain" for Set-Cookie. If not specified, it will use the subdomain of the Host header if the Host header is at least three units deep. (e.g. foo.com will not do anything, bar.foo.com will set Domain=foo.com).

- `user_(\S+)` (option)

  allowed: `/^.*$/`

  A user and password allowed. This should be used for testing only.

Examples

Loading the http_hmac_cookie module.

```
<noit>
  <modules>
    <generic image="http_hmac_cookie" name="http_hmac_cookie"/>
  </modules>
</noit>
```

http_observer

The http_observer module observers and exposes HTTP request/response information.

- **loader**: C
- **image**: http_observer.so

Module Configuration

- `max_count` (optional) [default: `10000`]

  allowed: `/^\d+$/`

  The max number of http requests to track.

- `max_age` (optional) [default: `30`]

  allowed: `/^\d+$/`

  The max time to retain completed requests.

Examples

Loading the http_observer module.

```
<noit>
  <modules>
    <generic image="http_observer" name="http_observer"/>
  </modules>
</noit>
```

lua_general

The lua_general module allows running of arbitrary lua code at startup.

- **loader**: C
- **image**: lua_mtev.so

Module Configuration

- `directory` (optional) [default: `/install/prefix/libexec/mtev/lua/?.lua`]

  allowed: `/^.+$/`

  This is the lua load path. See the lua manual for more details on meaning and syntax.

- `cpath` (optional)

  allowed: `/^.+$/`

  This is the lua DSO load path. See the lua manual for more details on meaning and syntax.

- `lua_module` (required)

  allowed: `/^.+$/`

  The lua module to load.

- `lua_function` (required)

  allowed: `/^.+$/`

  The lua function to run in the module.

- `Cpreload` (optional)

  allowed: `/^.*$/`

  Specify a set of luaopen_(.+) calls to make immediately after the context is created. This is useful if you're writing a standalone interpreter or other program that needs to extend lua (without shipping another lua module) before you start.

- `concurrent` (optional) [default: `false`]

  allowed: `/^(?:true|on|false|off)$/`

  Specify if the function should be invoked in each concurrent eventer thread.

- `gc_full` (optional) [default: `1000`]

  allowed: `/^^(?:0|[1-9]\d*)$$/`

  Specify how many yield/resume iterations may happen before a full garbage collection cycle is performed (0 means never).

- `gc_step` (optional) [default: `0`]

  allowed: `/^^(?:0|[1-9]\d*)$$/`

  Specify the parameter to normal lua_gc LUA_GCSTEP calls.

- `gc_stepmul` (optional) [default: `1`]

  allowed: `/^^(?:[1-9]\d*)$$/`

  Set the lua gc step multiplier.

Set the lua gc step multiplier.

- `gc_pause` (optional) [default: `200`]

  allowed: `/^^(?:[1-9]\d*)$$/`

  Set the lua gc pause percentage.

- `interrupt_time` (optional)

  allowed: `/^^\d+(?:\.\d+)?$$/`

  Specify the maximum time a lua operation may execute in a single eventer callback.

Examples

Loading the lua general module an run somefunction from the somemodule module.

```
      <noit>
        <modules>
          <module image="lua_mtev" name="lua_general">
            <config>
              <directory>/some/other/path/?.lua</directory>
              <lua_module>somemodule</lua_module>
              <lua_function>somefunction</lua_function>
            </config>
          </module>
        </modules>
      </noit>
```

lua_web

The lua_web module allows lua to drive http requests.

- **loader**: C
- **image**: lua_mtev.so

Module Configuration

- **`directory`** (optional) [default: `/install/prefix/libexec/mtev/lua/?.lua`]

  allowed: `/^.+$/`

  This is the lua load path. See the lua manual for more details on meaning and syntax.

- **`cpath`** (optional)

  allowed: `/^.+$/`

  This is the lua DSO load path. See the lua manual for more details on meaning and syntax.

- **`dispatch`** (required)

  allowed: `/^.+$/`

  The lua module to load.

- **`loop_assign_(.*)`** (optional)

  allowed: `/^(.+)$/`

  Optionally assigned a `mount_[name]` to a given `eventer_pool_t`. The name must match a `mount_[name]` stanza. The value is the name of the eventer loop pool desired.

- **`mount_(.*)`** (optional)

  allowed: `/^([^:]+):([^:]+):([^:]+)(?::(.+))?$/`

  module:method:mount[:expr]. The name `mount_[name]` simply must be unique and thus allows for multiple separate lua web services to be mounted in a single instance. Module is the name of the lua module the system will require, the function named "handler" will be called. Method is the HTTP method to serve (e.g. GET). The mount is the uri "directory" that will be handled by this `mount_[name]` stanza. Expr is a PCRE that further restricts the URIs handled.

- **`gc_full`** (optional) [default: `1000`]

  allowed: `/^^(?:0|[1-9]\d*)$$/`

  Specify how many yield/resume iterations may happen before a full garbage collection cycle is performed (0 means never).

- **`gc_step`** (optional) [default: `0`]

  allowed: `/^^(?:0|[1-9]\d*)$$/`

  Specify the parameter to normal lua_gc LUA_GCSTEP calls.

- **`gc_stepmul`** (optional) [default: `1`]

  allowed: `/^^(?:[1-9]\d*)$$/`

  Set the lua gc step multiplier.

- `gc_pause` (optional) [default: `200`]

  allowed: `/^^(?:[1-9]\d*)$$/`

  Set the lua gc pause percentage.

- `interrupt_time` (optional)

  allowed: `/^^\d+(?:\.\d+)?$$/`

  Specify the maximum time a lua operation may execute in a single eventer callback.

Examples

Loading the lua web module connection webmodule to the http services.

```
<noit>
  <modules>
    <module image="lua_mtev" name="lua_web">
      <config>
        <directory>/some/other/path/?.lua</directory>
        <dispatch>webmodule</dispatch>
        <mount_foo>foo:GET:/foo</mount_foo>
      </config>
    </module>
  </modules>
</noit>
```

zipkin_fq

The zipkin_fq module publishes Zipkin traces via Fq.

- **loader**: C
- **image**: zipkin_fq.so

Module Configuration

- `host` (optional) [default: `127.0.0.1`]

  allowed: `/^.+$/`

  The Fq host.

- `port` (optional) [default: `8765`]

  allowed: `/^\d+$/`

  The Fq port.

- `user` (optional) [default: `mtev`]

  allowed: `/^.+$/`

  The Fq user.

- `pass` (optional) [default: `mtev`]

  allowed: `/^.+$/`

  The Fq pass.

- `exchange` (optional) [default: `logging`]

  allowed: `/^.+$/`

  The Fq exchange.

- `route_prefix` (optional) [default: `scribe.zipkin.`]

  allowed: `/^.+$/`

  The routing prefix to which the traceid is appended.

Examples

Loading the zipkin_fq module.

```
<noit>
  <modules>
    <generic image="zipkin_fq" name="zipkin_fq"/>
  </modules>
</noit>
```

zipkin_jaeger

The zipkin_jaeger module publishes Zipkin traces to Jaeger.

- **loader**: C
- **image**: zipkin_jaeger.so

Module Configuration

- `host` (optional) [default: `127.0.0.1`]

  allowed: `/^.+$/`

  The jaeger collector host.

- `port` (optional) [default: `9411`]

  allowed: `/^\d+$/`

  The jaeger collector port.

- `period` (optional) [default: `500`]

  allowed: `/^\d+$/`

  The submission frequency in ms.

- `max_batch` (optional) [default: `500`]

  allowed: `/^\d+$/`

  The submission max batch size.

- `backlog` (optional) [default: `5000`]

  allowed: `/^\d+$/`

  The max backlog before spans are dropped.

- `retries` (optional) [default: `0`]

  allowed: `/^\d+$/`

  The number of HTTP retries upon failure..

Examples

Loading the zipkin_jaeger module.

```
      <noit>
        <modules>
          <generic image="zipkin_jaeger" name="zipkin_jaeger"/>
        </modules>
      </noit>
```

Development

This section of the manual will discuss the ins and outs of building applications using the libmtev APIs. There are many APIs in the system that can be used in an isolated fashion but they are built atop each other.

Most parts of the system rely heavily on the facilities provided in utils/. The eventer/ takes care not to require the configuration API directly. Most other APIs in the system have intricate interdependencies. Unless your use-cases for libmtev are very sophisticated, you need not worry about these subtleties and can simple use the APIs you need when you need them.

Many subsystems require explicit initialization before they are used and some subsystem initializations require prior initialization of dependent subsystems. This can cause a bit of boilerplate in your startup sequence which will seem unnecessary at first, but when your application becomes sufficiently complex you will appreciate it dearly.

For development reference purposes, libmtev source code contains a functional example application that does very little but can be used as a template for new applications.

Important Notes

In this section we will call out important notes that affect all developers that will integrate libmtev. These notes will be reinforced in other more specific sections, but often deal with the interaction between components.

The Eventer and the Watchdog and You

The eventer is the core operational concept of libmtev; it is its heart. Unlike some simple event loops out there, libmtev uses multi-threaded event loops (called eventer_pools). By default, there is one eventer_pool in the system, but more can be configured. In addition to the traditional event loop concept, asynchronous job queues ("eventer_jobq") are available for operations having the potential to block the normal event loops. These queues are named and can have different concurrencies based on work load.

So you have multiple asynch queues, each with configurable and run-time-adjustable concurrency plus multiple pools each with multiple threads running a "traditional" event loop.

In evented systems, it is important that you don't "block" the loop. It is so important, in fact, that the watchdog exists to ensure that you don't mistakenly do so. The watchdog is responsible for making sure that the program does not stall in the event loop. This also means that your event loops are responsible for issuing a heartbeat such that the watchdog knows you have not stalled. While this is C, and you can do almost anything, if you attempt to disable the heartbeats, you're doing it wrong and things will break unexpectedly and often in ways that will adversely affect production applications. Don't do that. You can set different watchdog timeouts per eventer_pool.

Multi-thread Safety

Memory Management

Multi-threaded apps can be hard, specifically in the area of memory management. The `safe_` memory management routines in `mtev_memory.h` are there to help, but they are not a silver bullet. They wrap libck epoch memory reclamation and make it such that memory touched *inside* an event callback will not be freed until the callback returns. Again, not a silver bullet.

Configuration

The `mtev_conf_` subsystem is based on libxml2 and has certain nuances to its thread-safety. In order to interoperate with the configuration system you must acquire and release sections of the config in either read (concurrent) or write (single access) mode. If you make any changes to the XML structure you *must* acquire a section in write mode or undefinied behavior may ensue. All locks are recusively safe and a write-lock will serve as a read-lock, but a read-lock will not upgrade to a write-lock.

Keep Related Events Together

The event system is thread-safe, but that doesn't mean you can't do bad things. Specifically, one should only manipulate events in the current event loop. Case in point, if you have a read/write event and a timeout event that can shutdown the read/write event, the two events should exist in the same event loop thread. Complications arise if the timeout fires and attempts to manipulate the read/write event if the read/write event is currently in its callback. It can be done safely, but it is complicated to get right and one should just make life simple if possible.

mtev_main

Every C application starts with `main()`. In order to make life easier, libmtev provides an `mtev_main` that facilitates configuration loading, logging setup, privilege separation and watchdog orchestration.

simple.c

```c
#include <mtev_main.h>
#include <mtev_memory.h>

#define APPNAME "simple"
static char *config_file = "/path/to/simple.conf";
static int debug = 0;
static int foreground = 0;
static char *glider = NULL;
static char *droptouser = NULL;
static char *droptogroup = NULL;
static mtev_lock_op_t lockop = MTEV_LOCK_OP_LOCK;
static int usage(const char *execname) {
  /* relevant usage output */
  return 2; /* returning 2 will avoid a watchdog restart */
}
static void parse_cli_args(int argc, char **argv) {
  /* left as an excercise to the reader */
}
int child_main() {
  /* implement your application here */
  /* typically: init things and start the event loop */
  return 0;
}
int main(int argc, char **argv) {
  parse_cli_args(argc, argv);
  if(!config_file) exit(usage(argv[0]));

  mtev_memory_init();
  mtev_main(APPNAME, config_file, debug, foreground,
            lockop, glider, droptouser, droptogroup,
            child_main);
  return 0;
}
```

The arguments to `mtev_main` are what is important here.

- **APPNAME**

  This is the name of your app, and must match the name of the root node in your XML configuration document.

- **config_file**

  The path to the config file. It is highly recommended that libmtev applications tie this to the `-c` CLI flag.

- **debug**

  If 0, the "debug" log_stream remains disabled. If non-zero, the "debug" log_stream is enabled. It is highly recommended that libmtev applications tie this to the `-d` CLI flag.

- **foreground**

If 0, the application will run in the background and be monitored by the watchdog subsystem. This is a reasonable behavior for almost all libmtev applications. If set to 1, the application will run in the foreground and not be monitored by the watchdog subsystem. It is highly recommended that libmtev applications tie this behavior to a single `-D` CLI argument. If set to 2, the application will be run under the watchdog subsystem, but the watchdog monitoring process will remain in the foreground. It is highly recommended that libmtev applications tie this behavior to repeated `-D -D` CLI arguments. When applications are run in the foreground, file descriptors 0, 1 and 2 remain pointing to the stdin, stdout and and stderr of the invoking context. If, the application is run in the background, all are replaced with file descriptors pointing to `/dev/null`.

- **lockop**

  This argument controls how `mtev_main` will respect the lockfile specified at the configuration root. If `MTEV_LOCK_OP_NONE` is used, then locking will be skipped; only do this if you know what you are doing as it could lead to application inconsistency. If `MTEV_LOCK_OP_LOCK` is used, the lockfile will be locked prior to starting; if locking fails, the application will exit immediately. If `MTEV_LOCK_OP_WAIT` is used, the application will wait for the lockfile to be available and then lock it before starting the application.

- **glider**

  An optional override for the `gilder` attribute of the watchdog configuration.

- **droptouser & droptogroup**

  If the application is run as the root user and these are specified, this informs `mtev_main` and its initialization routines that you intend to drop privileges and ensures that initialization processes that must be performed as the specified user and group are done so in that context.

Caution: Security Warning

The developer is responsible for dropping privileges during the initialization sequence in `child_main` via the `mtev_conf_security_init(...)` API call.

- **child_main**

  This is the surrogate main where the application should be initialized and run. Note that it should not return. Typically this function will end with an `event_loop()` which is non-returning.

The Eventer

The eventer is designed to perform micro tasks without the overhead of a context switch. The underlying goal is to support millions of "seemingly" concurrent heavy tasks by modifying the tasks to be reactive to state changes, make small, non-blocking progress, and yielding control back to the event loop.

Not all work can be done in a non-blocking fashion (e.g. disk reads/writes, and intense computational work). For this, the eventer provides work queues that allow for blocking operations.

Events (`eventer_t`) have a `callback` and a `closure` at their heart. The rest of the fields dictate when, why and possibly where the callback will be invoked with the provided closure.

Event types are dictated by the `mask` set in the `eventer_t` object. There are four basic event types available for use:

- File Descriptor Activity
- Recurrent Events
- Timed Events
- Asynchronous Events

File Descriptor Activity

```
#define EVENTER_READ             0x01
#define EVENTER_WRITE            0x02
#define EVENTER_EXCEPTION        0x04
```

File descriptor based events ("fd events") can react to any of three conditions (by bitwise OR):

- `EVENTER_READ` : the availability of data to read
- `EVENTER_WRITE` : the availability of buffer space so a write may succeed
- `EVENTER_EXCEPTION` : an error condition has occured on the file descriptor

Note that under most circumstances, the file descriptor should be a socket. In typical POSIX systems, these fd events don't fire as expected on files.

The return value of callbacks for fd events represents the new `mask` that should be used (for subsequent callback invocations). If 0 is returned, the event will be removed from the system and `eventer_free` will be invoked.

Operations on the file descriptor

While the `fd` field of the `eventer_t` is a normal file descriptor and normal POSIX operations can be performed on it (such as read(2), write(2), etc.). These operations are all abstracted away behind the `eventer_read` , `eventer_write` , etc. convenience functions.

The opset can be changed to support SSL operations and, in such operations, some non-blocking operations can require some non-obvious events to make progress. Namely, an SSL read may fail with `EAGAIN` , but require an `EVENTER_WRITE` event to continue due to a renegotiation.

```
static int
my_callback(eventer_t e, int mask, void *closure, struct timeval *now) {
  char buff[1024];
  size_t len;
  int mask = 0;

  len = eventer_read(e, buff, sizeof(buff), &mask);
  if (len < 0) {
    if (errno == EAGAIN) return mask|EVENTER_EXCEPTION;
    eventer_remove_fde(e);
    eventer_close(e, &mask);
    return 0;
  }

  /* use buff */

  return EVENTER_READ|EVENTER_EXCEPTION;
}
```

Order of operations (and other) notes:

- Always set an fd as non-blocking after creation using `eventer_set_fd_nonblocking(int fd)`
- Remember that `connect(2)` can block and often sets `errno = EINPROGRESS` and not `EAGAIN`
- Always remove an fd event from the eventer before closing it.
- To suspend an fd event (A) while an asynch event (B) is run:
    - In A's callback
        - `eventer_remove_fde(A)`
        - `eventer_ref(A)`
        - pass A as a part of B's closure
        - `eventer_add(B)`
        - `return 0`
    - in B's callback with `mask = EVENTER_ASYNCH_WORK`
        - `eventer_trigger(A, EVENTER_READ|EVENTER_WRITE)`

Recurrent Events

```
#define EVENTER_RECURRENT        0x80
```

Recurrent events are registered to run **every** time through an event loop. Use these with care. They should be extremely light-weight.

The return value from recurrent events should be `EVENTER_RECURRENT`.

recurrent_example.c (snippet)

```
  eventer_t e = eventer_alloc_recurrent(super_often, NULL;
  eventer_add(e);
```

Timed Events

```
#define EVENTER_TIMER            0x08
```

By setting the `mask` to `EVENTER_TIMER` and the `whence` to a desired time, an event will perform its callback at some point in the future.

The return value of callbacks for timer events should always be 0.

timers.c (snippet)

```
  eventer_t e;
  struct timeval whence;
  mtev_gettimeofday(&whence, NULL);
  whence.tv_sec += 5;
  e = eventer_alloc_timer(something_to_do_in_five_seconds, NULL, &whence);
  eventer_add(e);

  /* equivalent via helpers */
  struct timeval when;
  mtev_gettimeofday(&when, NULL);
  when.tv_sec += 5;
  eventer_add_at(something_to_do_in_five_seconds, NULL, when);

  /* or */
  struct timeval diff = { .tv_sec = 5 };
  eventer_add_in(something_to_do_in_five_seconds, NULL, diff);

  /* or */
  eventer_add_in_s_us(something_to_do_in_five_seconds, NULL, 5, 0);
```

Asynchronous Events

Asynchronous events within libmtev run within the context of an `eventer_jobq_t`. Programmatically, or via configuration, these job queues can be created. The context of the job queue can control how many threads are running and how many events may be queued using an advisory limit. Job queue concurrency can be controlled at run-time.

Job queues have several attributes related to concurrency: min, floor, desired, and max.

The min and the max describe "functional correctness" constraints on the jobq. For example, if it would "malfunction" if more than one job ran concurrently, then the max should be set to one. Setting the concurrency (or desired concurrency) for a jobq sets a target thread count to be achieved if there is work to be done. Floor sets a minimum thread count to maintain even if there is no work to be done. Min and max should not be changed at run-time and provide the domain boundaries for desired.

```
#define EVENTER_ASYNCH_WORK      0x10
#define EVENTER_ASYNCH_CLEANUP   0x20
#define EVENTER_ASYNCH           (EVENTER_ASYNCH_WORK | EVENTER_ASYNCH_CLEANUP)
```

asynch_example.c

```
static int
asynch_test(eventer_t e, int mask, void *c, struct timeval *now) {
  mtevL(mtev_error, "thread %p -> %04x\n", pthread_self(), mask);
  return 0;
}

void child_main() {

  ...

  mtevL(mtev_error, "thread %p -> add\n", pthread_self());
  eventer_t e = eventer_alloc_asynch(asynch_test, NULL);
  eventer_add(e);

  eventer_loop();
}
```

asynch_example.c (output)

```
[2016-12-04 12:01:50.216744] [error] thread 0x7fffcf7093c0 -> add
[2016-12-04 12:01:50.216776] [error] thread 0x70000086d000 -> 0010
[2016-12-04 12:01:50.216815] [error] thread 0x70000086d000 -> 0020
[2016-12-04 12:01:50.216863] [error] thread 0x7fffcf7093c0 -> 0030
```

It is important to note the life-cycle of an asynchronous event:

1. The event is added from some thread A (usually an event loop thread)
2. A jobq thread B will invoke the `callback` with `mask = EVENTER_ASYNCH_WORK` if `whence` is set and in the future.
3. Thread B will invoke the `callback` with `mask = EVENTER_ASYNCH_CLEANUP`
4. The event will return to thread A and `callback` will be invoked with `mask = EVENTER_ASYNCH`

Choosing Threads

By default new events are created on the "current" event loop thread. This has the effect of causing all new connections from a listener to gang on a single event loop thread. If an event is added from a non-event-loop thread, it will be assigned to thread 1.

The `thr_owner` field of the `eventer_t` structure describes which event loop thread owns the event. This can be changed using the `eventer_choose_owner_pool` and `eventer_choose_owner` functions.

To take an event and move it to a random thread within its current eventer pool:

move_event1.c (snippet)

```
static int
my_acceptor(eventer_t e, int mask, void *c, struct timeval *now) {
  eventer_pool_t *my_pool = eventer_get_pool_for_event(e);
  int newfd, newmask;

  newfd = eventer_accept(e, &addr, &addrlen, &newmask);
  if(newfd < 0) return newmask | EVENTER_EXCEPTION;
  if(eventer_set_fd_nonblocking(newfd)) {
    close(newfd);
    return EVENTER_READ|EVENTER_EXCEPTION;
  }
  int mask = EVENTER_READ|EVENTER_WRITE|EVENTER_EXCEPTION;
  newe = eventer_alloc_fd(my_handler, NULL, newfd, mask);
  eventer_set_owner(newe, eventer_choose_owner_pool(my_pool, rand()));
  eventer_add(newe);

  return EVENTER_READ | EVENTER_WRITE | EVENTER_EXCEPTION;
}
```

In the above example, a new connection is received and the new event that is created is assigned to a random thread within the same pool as the parent (listening) event.

move_event2.c (snippet)

```
  ...
  if (want_move) {
    eventer_pool_t *my_pool = eventer_get_pool_for_event(e);
    eventer_set_owner(e, eventer_choose_owner_pool(my_pool, rand()));
    return EVENTER_READ|EVENTER_WRITE|EVENTER_EXCEPTION;
  }
```

To switch an event from one thread to another, simply reassign the `thr_owner` and then return immediately with the desired mask. The eventer will reschedule the event on the requested thread. Be careful to not ping-pong back and forth without making proress!

Complex Interactions

The Eventer (ACO)

Non-blocking programming can be a real mind bender. While using the classic event system is fast and extremely powerful, it can be difficult to cope with context. When there isn't data available and you need to be called back later, you must track that context yourself through closures and if you consider complex protocols requiring asynch operations your resulting state machine can be a monster.

This is where ACO comes in. ACO stands for Arkenstone Co-routines; see libaco. This system provides a novel cooperative co-routine approach that allows the "call me back later" to happen in a seemingly blocking coding style.

How it works.

In order to use ACO, you must start an ACO procedure. This procedure runs a function that should not return, but instead call `aco_exit()` . Within this function, the execution is in the context of a "green thread."

Once in this function, the stack pointer is switched to an alternate co-routine "shared stack" (one per operating system thread). Each co-routine on a given operating system thread will have a small "save stack". When the thread performs an operation that would normally require using a callback within the classic event system, it yields control back to the event loop. When an event triggers that would allow this co-routine to continue, it is resumed. If the co-routine wasn't the last one on the "shared stack" then it is swapped in ("shared stack" copied to the current owner's "save stack" and the resuming thread's "save stack" copied into the "shared stack").

This stack swapping means that a coroutine's "save stack" can be right-sized and occupy very little space. The copying can be expensive for larger stacks so it is also important that you keep your stack usage small. Turning on debug logging will report stack sizes of co-routines as they are resumed. Additionally, your compiler can help you: `-Wstack-size=1024` for example.

Co-routines do not "manage" asynchronous events, but can place calls to them.

Relationship with the classic event system.

You can create and schedule classic events from anywhere in the system including a co-routine. Special interactions with events will yield/resume automatically are performed through the `eventer_aco` family of functions. Not all classic events make sense to interact with.

- Recurrent events have no meaning in co-routines. Co-routines aren't recurrent.
- Timed events make little sense in non-callback-oriented coding. `eventer_aco_sleep` should be everything you need.
- Asynchronous events are blocking already, so co-routines make no sense. However, waiting for the completion of an asynchronous event is quite useful. This is what the `eventer_aco_simple_asynch` and `eventer_aco_run_asynch` families of functions do.
- File-descriptor-based events are the obvious deep integration. Each of the `eventer_aco_` variants to read, write, accept, and close events have optional timeouts as parameters instead of a mask.

If a normal `eventer_read` (or write, etc.) function is called upon an `eventer_aco_t` object (breaking the type safety), it will be treated as if it is a call to its `eventer_aco_` counterpart with no timeout.

Examples

Starting a simple co-routine

```
static void my_coroutine(void) {
  struct timeval *sleep_time = eventer_aco_arg();
  while(1) {
    mtev_aco_sleep(sleep_time);
    mtevL(mtev_error, "Waking up to do something\n");
  }
  aco_exit();
}

void calling_function(void) {
  struct timeval *sleep_time = malloc(sizeof(*sleep_time));
  sleep_time->tv_sec = 2;
  sleep_time->tv_usec = 0;
  eventer_aco_start(my_coroutine, sleep_time);
}
```

The `calling_function` can be call any time after `eventer_init()` . While the `my_coroutine` function looks blocking, it is actually cooperating with all classic events and other aco events on the same eventer thread.

Moving an event to aco

```
static void my_aco_process(void);

static int
classic_callback(eventer_t e, int mask, void *c, struct timeval *now) {
  // First we must remove the event from the eventer system.
  eventer_remove_fde(e)

  // When we return 0 from this function, the event will be freed,
  // so we need a copy.
  eventer_t copy = eventer_alloc_copy(e);

  // This event needs to be converted to an aco event, but we don't
  // have a coroutine yet.
  eventer_aco_start(my_aco_process, copy);

  return 0;
}

static void
my_aco_process(void) {
  eventer_t classic_e = eventer_aco_arg();

  // We're now in an aco context so we can convert this to an aco event.
  eventer_aco_t e = eventer_set_eventer_aco(classic_e);

  // Use eventer_aco_* functions to interact with e.

  aco_exit();
}
```

Making an aco listener

The listener subsystem is outfitted to dispatch to aco threads. In this case, all of the event duplication and conversion is done for you. Simply register a named function as an aco function with the listener subsystem.

```
static void listen_to_me(void) {
  eventer_aco_t e = eventer_aco_arg();
  struct timeval tensec = { .tv_sec = 10 };
  while(1) {
    int rv;
    char buff[128];
    rv = eventer_aco_read(e, buff, sizeof(buff), &tensec);
    if(rv == -1) {
      if(errno == ETIME) {
        eventer_aco_write(e, "bye!\n", 5, NULL);
      }
      break;
    }
    if(rv >=4 && !strncasecmp(buff, "quit", 4)) {
      eventer_aco_write(e, "quitter!\n", 9, NULL);
      break;
    }
    if(eventer_aco_write(e, "thanks!\n", 9, NULL) < 0)
      break;
  }

  eventer_aco_close(e);
  eventer_aco_free(e);
  // our caller aco_exit()s, but it won't hurt anything if we do it first
  // aco_exit();
}

static int child_main(void) {
  ...
  eventer_init();
  mtev_listener_register_aco_function("listen_to_me", listen_to_me);
  mtev_listener_init(APPNAME);
  eventer_loop();
  return 0;
}
```

Using aco with REST

To use aco with REST handlers, simply use `mtev_rest_mountpoint_set_aco` . The handler has the same signature, but it will be serviced within an aco co-routine. This is particularly useful for calling asynch events. As the `eventer_aco_simple_asynch` family of functions will run code asynchronously in a job queue and appear to block until it is complete.

```
static void asynch_hello(void *closure) {
  mtev_http_rest_closure_t *restc = closure;
  mtev_http_session_ctx *ctx = restc->http_ctx;
  sleep(1);
  mtev_http_response_append_str(ctx, "Hello world.\n");
}

static int
hello_handler(mtev_http_rest_closure_t *restc,
              int npats, char **pats) {
  mtev_http_session_ctx *ctx = restc->http_ctx;
  mtev_http_response_ok(ctx, "text/plain");
  eventer_aco_simple_asynch(asynch_hello, restc);
  mtev_http_response_end(ctx);
  return 0;
}

static int child_main(void) {
  ...
  eventer_init();
  mtev_http_rest_init();
  mtev_listener_init("myapp");
  ...

  mtev_rest_mountpoint_t *rule = mtev_http_rest_new_rule(
    "GET", "/", "^hello$", hello_handler
  );
  mtev_rest_mountpoint_set_aco(rule, mtev_true);

  eventer_loop();

}
```

REST

The default network listening framework allows for simple registration of REST-based services via HTTP service.

app.conf (snippet)

```
<app>
  <listener type="http_rest_api" address="127.0.0.1" port="80">
    <config>
      <acl>internal</acl>
      <document_root>/path/to/docroot</document_root>
    </config>
  </listener>
  <rest>
    <acl type="deny" listener_acl="^internal$">
      <rule type="allow" url="."/>
    </acl>
    <acl type="deny"></acl>
  </rest>
</app>
```

More information about listener configuration can be found in the listener configuration section.

The above config establishes a REST-capable listener listening on `localhost` port 80. The "ACL" config option is set to `internal` which can be used as a filter for ACL rules in the `rest/acl` config section. The `document_root` is set, which will enable serving of static files on URLs that do not otherwise match routing rules. The ACL rule for `internal` set that all urls are allowed followed by a blanket deny rule. Other listeners that might not specify an `acl` option would not see the first ACL allowing any URL, but still see the blanket deny rule.

Registering a REST handler.

myhandler.c (snippet)

```
#include <mtev_rest.h>

static int
my_rest_handler(mtev_http_rest_closure_t *restc,
                int npats, char **pats) {
  mtev_http_session_ctx *ctx = restc->http_ctx;
  mtev_http_response_ok(ctx, "text/plain");
  mtev_http_response_appendf(ctx, "myhandler: %s\n", pats[0]);
  mtev_http_response_end(ctx);
  return 0;
}

void child_main() {
 ...

  mtev_http_rest_register_auth(
    "GET", "/", "^myhandler/(.+)$", my_rest_handler,
           mtev_http_rest_client_cert_auth
  );
  mtev_http_rest_register_auth(
    "GET", "/", "^(.*)$", mtev_rest_simple_file_handler,
           mtev_http_rest_client_cert_auth
  );
  eventer_loop();
  return 0;
}
```

Handling asynchronous work.

In order to complete some complex action in response to an inbound REST request, it might be necessary to schedule some asynchronous work and complete the response later. This is possible, but requires a bit of juggling. The basic idea is:

- from your handler:
  - copy and remove the connection's event from the eventer
  - set the rest's fastpath to a completion routine
  - schedule asynchronous work
  - return 0;
- from the async job's completion:
  - trigger the connection's event

sleepy.c (snippet)

```
#include <mtev_rest.h>

static int handler_work(eventer_t e, int mask, void *closure,
                        struct timeval *now) {
  mtev_http_rest_closure_t *restc = closure;
  if(mask == EVENTER_ASYNCH_WORK) {
    sleep(5);
  }
  if(mask == EVENTER_ASYNCH) {
    mtev_http_session_resume_after_float(restc->http_ctx);
  }
  return 0;
}

static int handler_complete(mtev_http_rest_closure_t *restc,
                            int npats, char **pats) {
  mtev_http_session_ctx *ctx = restc->http_ctx;
  mtev_http_response_ok(ctx, "text/plain");
  mtev_http_response_append_str(ctx, "Hello world\n");
  mtev_http_response_end(ctx);
  return 0;
}

static int handler(mtev_http_rest_closure_t *restc,
                   int npats, char **pats) {
  eventer_t conne, worke;
  mtev_http_session_ctx *ctx = restc->http_ctx;

  /* remove the eventer */
  conne = mtev_http_connection_event_float(mtev_http_session_connection(ctx));
  if(conne) eventer_remove_fde(conne);

  /* set a completion routine */
  restc->fastpath = handler_complete;

  /* schedule our work */
  worke = eventer_alloc_asynch(handler_work, restc);
  eventer_add(worke);
  return 0;
}
```

Handling POST/PUT data

Reading data from the HTTP request is done by calling the `mtev_http_session_req_consume` function. This can be tedious, so unless you are doing something special it can be much easier to simply first invoke the `mtev_rest_complete_upload` convenience wrapper.

It must be called as the first action inside your REST callback handler. Any manipulation of the `restc` (closures in particular) will have undefined outcome.

```
static int handler(mtev_http_rest_closure_t *restc,
                   int npats, char **pats) {
  int mask;
  void *payload;
  int64_t payload_len;
  mtev_http_request *req = mtev_http_session_request(restc->http_ctx);

  if(!mtev_rest_complete_upload(restc, &mask)) return mask;
  payload = mtev_http_request_get_upload(req, &payload_len);

  ...
}
```

Logging

Logging within libmtev depends heavily upon logging configuration.

DTrace

The logging system is instruemented with DTrace, so despite any configuration settings, an operator can leverage DTrace to sniff logs (regardless of their outlets or disabled state) using the `libmtev*:::log` probe. See DTrace operations.

log_stream

The C logging API requires directing each log statements to an `mtev_log_stream_t`. There are four builtin log stream: `mtev_stderr`, `mtev_error`, `mtev_notice`, and `mtev_debug`. Their behavior can be modified via configuration or programmatically via the API.

Before using the log system it must be initialized via `mtev_log_init(int debug)`, but `mtev_main` handles this for you.

Startup sequence

Within your `child_main` that is called by `mtev_main`, you should reopen all logs and then enable rotation (config driven).

```
  if(mtev_conf_load(config_file) == -1) {
    mtevL(mtev_error, "Cannot load config: '%s'\n", config_file);
    exit(2);
  }
  mtev_log_reopen_all();
  mtev_log_go_asynch();
  if(eventer_init() == -1) mtevFatal(mtev_stderr, "Cannot initialize eventer\n");
  mtev_conf_log_init_rotate(APPNAME, mtev_false);
```

Because the watchdog subsystem can restart our process on crash, it is important to always (re)load the config and reopen all log files immediate inside of `child_main`. `mtev_log_go_asynch()` is optional and will make logging operations (aside from type `memory`) asychronous to the calling thread. This is important for high-performance systems where writting logs can interfere with latency and throughput objectives.

`mtev_conf_log_init_rotate` tells the configuration system to register maintenance for any logs configured to have rotation. It uses the eventer subsystem to performan maintenance and requires the eventer to be initialized beforehand.

Getting a mtev_log_stream_t

```
static mtev_log_stream_t my_awesome_log;
void some_init_function() {
  my_awesome_log = mtev_log_stream_find("awesome");
}
```

Getting a log stream by name will implicitly create a typeless log_stream with no outlets if no such name log already exists in the system.

Writing to a log

```
  struct timeval now;
  mtev_gettimeofday(&now, NULL);
  mtevLT(mtev_error, &now, "Avoids the internal mtev_gettimeofday call\n");
```

```
mtevL(my_awesome_log, "My %d %s-style format string.\n", 1, "sprintf");
```

The mtevLT and mtevL "functions" are actually vararg macros. This is done so that if a log is disabled, none the arguments are actually evaluated. If one of the parameters to these macros is an expensive function call, the call will be elided if the log is disabled.

> Note: writing to a log does not automatically append a line feed. You almost always want to include line feeds in your log lines explicitly.

Logging a programming error

```
if(disaster_strikes) {
  mtevFatal(mtev_error, "Disaster has struck, I give up.\n");
}
```

A special macro `mtevFatal(<stream>, <fmt>, ...)` is provided that will take three actions.

1. `mtev_log_go_synch()` will be called to ensure logging goes synchronous and the subsequent log message will be written.
2. `mtevL(...)` log the arguments.
3. `abort()`

Other logging APIs

mtev_log_go_asynch

```
void mtev_log_go_asynch();
```

All logging operations that can be performed asynchronously will be done asynchronously.

mtev_log_go_synch

```
void mtev_log_go_synch();
```

All logging operations will be performed synchronously with respect to the called upon return of this function.

mtev_log_reopen_all

```
void mtev_log_reopen_all();
```

Reopen all log files. Log types that do not implement reopen are unaffected.

mtev_log_reopen_type

```
void mtev_log_reopen_type(const char *type);
```

Repoen all log files of the specified type.

mtev_log_stream_get_flags

```
#define MTEV_LOG_STREAM_ENABLED
#define MTEV_LOG_STREAM_DEBUG
#define MTEV_LOG_STREAM_TIMESTAMPS
#define MTEV_LOG_STREAM_FACILITY
```

```
int mtev_log_stream_get_flags(mtev_log_stream_t);
```

Retrieve a bitmask of the enabled flags on a stream.

mtev_log_stream_set_flags

```
int mtev_log_stream_set_flags(mtev_log_stream_t, int newmask);
```

Set a bitmask of the enabled flags on a stream, returning the previous mask.

mtev_log_stream_get_type

```
const char *mtev_log_stream_get_type(mtev_log_stream_t);
```

Returns the type of the log stream.

mtev_log_stream_get_name

```
const char *mtev_log_stream_get_name(mtev_log_stream_t);
```

Returns the name of the log stream.

mtev_log_stream_get_path

```
const char *mtev_log_stream_get_path(mtev_log_stream_t);
```

Returns the path of the log stream.

mtev_log_stream_set_property

```
void mtev_log_stream_set_property(mtev_log_stream_t ls,
                                  const char *prop, const char *v);
```

Set an arbitrary property on a log stream.

mtev_log_stream_get_property

```
const char *mtev_log_stream_get_property(mtev_log_stream_t ls,
                                         const char *prop);
```

Retrieve an arbitrary property from a log stream.

Adding custom logging types

Libmtev ships with four logging types: `memory` , `file` , `file_synch` , and `jlog` . The system is extensible and additional types can be added. Those types do not have the benefit of existence checking as the logging is initialized before dynamic modules are loaded and new logging types are typically added via dynamic modules. This chicken-and-egg issue requires us to load log_streams with unknown types and resolve them post-facto.

```
  typedef struct {
    mtev_boolean supports_async;
    int (*openop)(mtev_log_stream_t);
    int (*reopenop)(mtev_log_stream_t);
    int (*writeop)(mtev_log_stream_t, const struct timeval *whence, const void *, size_t);
    int (*writevop)(mtev_log_stream_t, const struct timeval *whence, const struct iovec *iov, int iovcnt);
    int (*closeop)(mtev_log_stream_t);
    size_t (*sizeop)(mtev_log_stream_t);
    int (*renameop)(mtev_log_stream_t, const char *);
    int (*cullop)(mtev_log_stream_t, int age, ssize_t bytes);
  } logops_t;

  void mtev_register_logops(const char *name, logops_t *ops);
```

By implementing the `logops_t` structure and registering it with a name, you can then delcare `<log>` stanzas with `type` of that name.

The following operations are optional to implement:

- reopenop
- writevop
- sizeop
- renameop
- cullop

Attaching contexts

Most log implementations require some context to be attached to the `mtev_log_stream_t`. Two functions are provided to attach and retrieve an arbitrary context to a log_stream. These functions should only be used by those implementing new logging types as the context will be some arbitrary domain-specific struct that is opaque to those outside the implementation.

```
  void *mtev_log_stream_get_ctx(mtev_log_stream_t);
  void mtev_log_stream_set_ctx(mtev_log_stream_t, void *);
```

Arbitrary Hooks

Building a callout API makes sense for common, structured features in software, but occasionally there is a need to provide somewhat arbitrary hook points after the software is designed and the mtev_hooks system is what satisfies this need.

The design goals here are somewhat specific in that we would like to allow for a large number of hook points at low cost when not instrumented. As such, a hash lookup of registered hooks would be considered too expensive. Additionally, we want to provide strong, compile-time type safety as it can be all too easy to hook something with a function using a slightly incorrect prototype that could result in disastrous corruption or crashes (or, perhaps worse, extremely subtle bugs that are punishing to troubleshoot).

The hooks system is simply a set of two macros; one allowing for the declaration of function prototypes for registering and invoking specific programmer-specific instrumentation points, and the other providing an implementation of the registration and invocation routines. Due to the nature of C, the macro calling conventions are less than elegant, but ultimately require no complicated implementation by the programmer.

Hook Declaration

Declaring hooks is done by calling the `MTEV_HOOK_PROTO` macro with the name of the hook (a term that composes a valid C function name), the arguments it expects, the type of closure (usually a `void *`), and some variations on those themes that provide CPP enough info to construct an implementation with no programmer "programming."

The declaration of a hook "foo" will result in two functions: `foo_hook_invoke` and `foo_hook_register` .

Declaring a hook "foo" (in a header)

This hook "foo" takes a `struct timeval *` as an argument in addition to its closure.

```
#include <mtev_hooks.h>

MTEV_HOOK_PROTO(foo, (struct timeval *now),
                void *, closure, (void *closure, struct timeval *now));
```

Implementing a hook "foo" (in a source file)

```
#include <mtev_hooks.h>

MTEV_HOOK_IMPL(foo, (struct timeval *now),
               void *, closure, (void *closure, struct timeval *now),
               (closure,now));
```

Hook Usage

Once the hook is implemented, it can be used by the application and instrumented by code at runtime. In the below example, we'll invoke the `foo` instrumentation and assuming no issues arise, we'll invoke the original `foo_work()` function.

Instrumenting a function conditionally

Before we instrument, suppose we have:

```
  /* preamble code */
  foo_work();
  /* postamble code */
```

Now we wish to allow programmers to add instrumentation immediately before this code that can conditionally prevent its execution, so we would modify the above code to look like:

```
/* preamble code */
struct timeval now;
mtev_gettimeofday(&now, NULL);
if(MTEV_HOOK_CONTINUE == foo_hook_invoke(&now)) {
     foo_work();
}
/* postamble code */
```

If the hook should not conditionally cause or prevent code to run, the `_invoke` function's return value can be ignored.

In order to register a function that allows the above execution on every other subsequent execution one would provide the following:

```
static my_sample_hook(void *closure, struct timeval *now) {
  static int alt = 0;
  return (alt++ % 2) ? MTEV_HOOK_CONTINUE : MTEV_HOOK_DONE;
}

void my_init_fuction() {
  foo_hook_register("sample", my_sample_hook, NULL);
}
```

The implementation of the hook can be elsewhere in the code, even in a dynamically loaded module. When the hook is registered (you must orchestrate the calling of `my_init_function`), the behavior of the `foo_work()` callsite will change and our hook will be called. Given the above implementation, the `struct timeval` will be ignored, but every other time we reach the call site, `foo_work()` will be skipped due to a `MTEV_HOOK_DONE` return value.

jemalloc

libmtev has some special support available if you are running with the jemalloc allocator loaded.

> Note that if your application isn't directly linked with jemalloc, libmtev will still notice its presence at run-time if loaded via `LD_PRELOAD` . So, the following can be used in conjunction with an operator's force loading of libjemalloc.so

malloc statistics.

Malloc statistics (in JSON) are available at the URL: `/mtev/memory.json`

Activating jemalloc profiling

To perform heap profiling you must have jemalloc heap profiling enabled via environment variable:

`export MALLOC_CONF="prof:true,prof_active:false"`

Or via the special `/etc/malloc.conf` string file:

`sudo ln -s 'prof:true,prof_active:false' /etc/malloc.conf`

You can then flip on heap profiling in your mtev app by curling:

`/mtev/heap_profile?active=true`

This will turn on profiling from that moment until you disable it via:

`/mtev/heap_profile?active=false`

In a variety of cases, it might be desirable to have profiling active from the point of application start. To do this set `prof_active:true` in the `MALLOC_CONF` .

Heap Profiling

To get periodic heap profile dumps from your running application do:

`curl yourmachine:yourport/mtev/heap_profile?trigger_dump=true > profile.prof`

This will spit back jeprof format heap information which can then be passed to the `jeprof` analysis program for further analysis. For example, to show allocations by source code line, but from a perspective outside of libmtev's use of SMR (`mtev_memory_`) and libck's hash tables (`ck_hs_`), one could run:

`jeprof --text --lines --exclude='(mtev_memory_|ck_hs_)' /path/to/your/executable profile.prof`

For more information on jemalloc heap profiling, see here: jemalloc heap profiling

Operations

Observability

If you can't see what your system is doing, then you clearly don't care that much if it is doing what it is supposed to be. There is really no excuse for unanswerable questions in a production environment. Instrumentation for observability is the starting support of responsible systems development, deployment, and operations.

Environment Variables

Other than the interactions environment variables can have on configuration, there are several standard environment variables that effect the operation of any libmtev application.

- **EVENTER_DEBUGGING**

  If set to 1 it will enable the `debug/eventer` log stream.

- **MTEV_DWARF**

  If set to 0, dwarf sections will not be analyzed to make human readable stacktraces.

- **MTEV_JIT_OFF**

  If set to 1 it will disable the JIT within the lua module.

- **MTEV_JIT_OPT**

  If numeric and positive, it be passed numerically to `jit.opt.start` in the lua module.

- **MTEV_RDTSC_REQUIRE_INVARIANT**

  If set to 0, this will disable the requirement for an invariant rdtsc for libmtev's "faster time" support.

- **MTEV_RDTSC_DISABLE**

  If set to 1, this will disable libmtev's "faster time" support.

- **MTEV_THREAD_BINDING_DISABLE**

  If set to 1, threads created via `mtev_thread` will not be bound to CPUs.

- **MTEV_DIAGNOSE_CRASH**

  If set to 0, libmtev's internal crash handling code will not be run. If set to 1, will run libmtev's internal crash handling code. If set to a file path for a script or external tool, this will be invoked on a crash with the thread id and process pid as parameters (pid only on non-linux). Use a wrapper script with execution rights and sudoers as needed to give sudo permissions or additional calling parameters when invoking the external tool.

- **MTEV_LOG_DEBUG**

  If numeric and non-zero, this turns on debug logging for the logging system. This should only be used by developers to debug the logging system itself. These logs all go to stderr.

- **MTEV_ASYNCH_CORE_DUMP**

  If zero, this disables "asynch core dumps." It forces the monitor process to wait for the child process to leave the process table before attempting to restart it upon crash.

- **MTEV_ALTSTACK_SIZE**

  A size (in bytes) for the altstack for handling crashes.

- **MTEV_WATCHDOG_TIMEOUT**

  A timeout observed by the parent monitor. If the child process does not heartbeat from each thread within this number of seconds, the monitor will terminate and restart the child. Non-integral numbers are allowed.

REST-accessible Observability Endpoints

Various components in the libmtev namespace can register REST endpoints via the `http_rest.h` APIs and expose a wealth of configuration and run-time mechanics to the caller.

application.conf

```
<root>
<listeners>
  <listener type="http_rest_api" address="*" port="8888" ssl="off">
    <config>
      <document_root>/path/to/docroot</document_root>
    </config>
  </listener>
</listeners>
<rest>
  <acl>
    <!-- you should consider tighter ACLs -->
    <rule type="allow" />
  </acl>
</rest>
</root>
]]></programlisting>
</example>
```

In addition to a configuration snippet similar to the above, the HTTP REST subsystem must be programmatically initialized via a call to `mtev_http_rest_init()` .

The REST listener is implemented on a non-compliant HTTP listener by subverting the four-byte control words "DELE", "MERG", "GET ", "HEAD", "POST", "PUT " and dropping them into a compliant HTTP state machine. While the session initiation is not strictly compliant with the HTTP specification it happily serves all known browsers and plays nicely with HTTP proxies as well.

If the listener "type" is `http_rest_api` , then only the REST handler is served on that listening socket. If the more general type of `control_dispatch` is used, then a full control channel is served and the REST services are superimposed on that. Listeners of type `control_dispatch` are even less HTTP compliant, but still serve all web browsers and proxies correctly.

The capabilities endpoint

It is highly recommended that you expose the libmtev capabilities endpoints. This service exposes information about the libmtev version and build as well as any dispatch handlers that have been registered via `mtev_control_dispatch_delegate(...)` .</para>

GET /capa

Bot the CAPA HTTP verb and a GET of /capa will return the application capabilities in XML.

```
# curl -XCAPA http://127.0.0.1:8888/
# curl http://127.0.0.1:8888/capa

<?xml version="1.0" encoding="utf8"?>
<mtev_capabilities>
  <version>master.5553ef6a1c838ac5a639a634027db7b2c39be23d.1459876317</version>
  <unameBuild bitwidth="64">
    <sysname>Darwin</sysname>
    <nodename>cudgel</nodename>
    <release>15.3.0</release>
    <version>Darwin Kernel Version 15.3.0; root:xnu-3248.30.4~1/RELEASE_X86_64</version>
    <machine>x86_64</machine>
  </unameBuild>
  <unameRun bitwidth="64">
    <sysname>Darwin</sysname>
    <nodename>cudgel</nodename>
    <release>15.4.0</release>
    <version>Darwin Kernel Version 15.4.0; root:xnu-3248.40.184~3/RELEASE_X86_64</version>
    <machine>x86_64</machine>
  </unameRun>
  <features/>
  <current_time>1460120546.029</current_time>
  <services>
    <service name="control_dispatch" connected="true">
      <command name="mtev_wire_rest_api" version="1.0" code="0x504f5354"/>
      <command name="mtev_wire_rest_api" version="1.0" code="0x4d455247"/>
      <command name="mtev_wire_rest_api" version="1.0" code="0x48454144"/>
      <command name="mtev_wire_rest_api" version="1.0" code="0x47455420"/>
      <command name="capabilities_transit" version="1.0" code="0x43415041"/>
      <command name="mtev_wire_rest_api" version="1.0" code="0x50555420"/>
      <command name="mtev_wire_rest_api" version="1.0" code="0x44454c45"/>
    </service>
  </services>
  <modules/>
</mtev_capabilities>
```

GET /capa.json

Capabilities are also available in JSON format.

```
# curl http://127.0.0.1:8888/capa.json

{
  "version": "master.5553ef6a1c838ac5a639a634027db7b2c39be23d.1459876317",
  "unameBuild": {
    "bitwidth": 64,
    "sysname": "Darwin",
    "nodename": "cudgel",
    "release": "15.3.0",
    "version": "Darwin Kernel Version 15.3.0; root:xnu-3248.30.4~1/RELEASE_X86_64",
    "machine": "x86_64"
  },
  "unameRun": {
    "bitwidth": 64,
    "sysname": "Darwin",
    "nodename": "cudgel",
    "release": "15.4.0",
    "version": "Darwin Kernel Version 15.4.0; root:xnu-3248.40.184~3/RELEASE_X86_64",
    "machine": "x86_64"
  },
  "features": {},
  "current_time": "1460121207543",
  "services": {
    "0x10e19480": {
      "control_dispatch": "control_dispatch",
      "commands": {
        "0x504f5354": {
          "name": "mtev_wire_rest_api",
          "version": "1.0"
        },
        "0x4d455247": {
          "name": "mtev_wire_rest_api",
          "version": "1.0"
        },
        "0x48454144": {
          "name": "mtev_wire_rest_api",
          "version": "1.0"
        },
        "0x47455420": {
          "name": "mtev_wire_rest_api",
          "version": "1.0"
        },
        "0x43415041": {
          "name": "capabilities_transit",
          "version": "1.0"
        },
        "0x50555420": {
          "name": "mtev_wire_rest_api",
          "version": "1.0"
        },
        "0x44454c45": {
          "name": "mtev_wire_rest_api",
          "version": "1.0"
        }
      }
    }
  },
  "modules": {}
}
```

The Eventer System

Assuming the application has registered the eventer system reporting over rest via the `mtev_events_rest_init()` call, robust information about the current state of all events in the system is available in JSON.

GET /eventer/sockets.json

```
# curl http://localhost:8888/eventer/sockets.json

[
  {
    "callback": "listener(mtev_console)",
    "fd": 9,
    "local": {
      "address": "0.0.0.0",
      "port": 32322
    },
    "impl": "POSIX",
    "mask": 5,
    "eventer_pool": "default"
  },
  {
    "callback": "listener(control_dispatch)",
    "fd": 11,
    "local": {
      "address": "0.0.0.0",
      "port": 8888
    },
    "impl": "POSIX",
    "mask": 5,
    "eventer_pool": "default"
  },
  {
    "callback": "mtev_wire_rest_api/1.0",
    "fd": 12,
    "local": {
      "address": "127.0.0.1",
      "port": 8888
    },
    "remote": {
      "address": "127.0.0.1",
      "port": 60088
    },
    "impl": "POSIX",
    "mask": 5,
    "eventer_pool": "default"
  }
]
```

GET /eventer/timers.json

```
# curl http://localhost:8888/eventer/timers.json

[
  {
    "callback": "mtev_conf_watch_config_and_journal",
    "whence": 1480805474873,
    "eventer_pool": "default"
  }
]
```

GET /eventer/jobq.json

```
# curl http://localhost:8888/eventer/jobq.json

{
  "default_queue": {
    "concurrency": 10,
    "desired_concurrency": 10,
    "total_jobs": 1,
    "backlog": 0,
    "inflight": 0,
    "timeouts": 0,
    "avg_wait_ms": 0.034546,
    "avg_run_ms": 1142.833808
  },
  "default_back_queue/0": {
    "concurrency": 0,
    "desired_concurrency": 0,
    "total_jobs": 1,
    "backlog": 0,
    "inflight": 0,
    "timeouts": 0,
    "avg_wait_ms": 0,
    "avg_run_ms": 0
  },
  "default_back_queue/1": {
    "concurrency": 0,
    "desired_concurrency": 0,
    "total_jobs": 0,
    "backlog": 0,
    "inflight": 0,
    "timeouts": 0,
    "avg_wait_ms": 0,
    "avg_run_ms": 0
  }
}
```

GET /eventer/logs/<name>.json

Returns logs from the stream named `<name>` of type "memory".

Querystring parameters include:

- last=N

requests only the N most recent log lines. Log lines are returned with index numbers.

- since=I

  requests only log lines after index I.

```
# curl http://localhost:8888/eventer/logs/internal.json?last=2

[
  {
    "idx": 4,
    "whence": 1480805410721,
    "line": "Generating 1024 bit DH parameters.\n"
  },
  {
    "idx": 5,
    "whence": 1480805416435,
    "line": "Finished generating 1024 bit DH parameters.\n"
  }
]
```

Telnet Console Observability

A full telnet console is available for online operations of libmtev applications. To enable the telnet console, add something like the following to your configuration. Note that "bad things" can be done via the telnet console, so restricting access makes good sense.

listener.conf

```
    <consoles type="mtev_console">
       <listener address="127.0.0.1" port="32322">
         <config>
           <line_protocol>telnet</line_protocol>
         </config>
       </listener>
     </consoles>
```

The telnet console support tab completion, so navigating and exploring the possibilities can be an interactive experience.

Logs

```
app# show log internal
[1] Hello world.

app# show log internal 100
[1] Hello world.

app# log details debug/eventer
{ "name": "debug\/eventer",
  "enabled": false,
  "debugging": false,
  "timestamps": false,
  "facility": false,
  "outlets": [ { "name": "debug",
                 "outlets": [ { "name": "stderr" } ] } ] }

app# log notice Hello from the console
logged.
```

Commands

`show log <logname> [# lines]`

> Show the last requested number lines (23 if omitted) of `<logname>` if it is a "memory" type log.

`log details <logname>`

> Show the details of the the `<logname>` log stream, including outlets.

`log to <logname> <something to log>`

> Cause a `<something to log>` to be immediately logged to the specified log.

`log [dis]connect <logname> [tgtlogname]`

> Connect or disconnect `<tgtlogname>` as an outlets for `<logname>`. If `<tgtlogname>` is omitted, the attached console is used.

`[no] log <enable|facility|debug|timestamps> <logname>`

> Sets (or unsets) flags on the specified log

> Sets (or unsets) flags on the specified log.

DTrace-accessible Observability

See the DTrace Guide for general information on how to use DTrace.

libmtev includes a number of Statically Defined Trace points (SDTs) for key events in the system.

Probes will be visible using the provider `libmtev<pid>` where `<pid>` is the process ID of a libmtev application. To trace all PIDs of all libmtev applications currently running, one would use the provider definition `libmtev*:::` .

List all available libmtev probes:

```
dtrace -l -n 'libmtev*:::'
```

DTrace probe definitions

Logging

```
provider libmtev {
  probe log (char *facility, char *file, int line, char *msg);
};
```

Eventer

```
provider libmtev {
  probe eventer-accept-entry (int, void *, int, int, void *);
  probe eventer-accept-return (int, void *, int, int, void *, int);
  probe eventer-read-entry (int, char *, size_t, int, void *);
  probe eventer-read-return (int, char *, size_t, int, void *, int);
  probe eventer-write-entry (int, char *, size_t, int, void *);
  probe eventer-write-return (int, char *, size_t, int, void *, int);
  probe eventer-close-entry (int, int, void *);
  probe eventer-close-return (int, int, void *, int);
  probe eventer-callback-entry (void *, void *, char *, int, int, int);
  probe eventer-callback-return (void *, void *, char *, int);
};
```

Reverse Connections

```
provider libmtev {
  probe reverse-reschedule (int, char *, char *, int);
  probe reverse-shutdown-permanent (int, char *, char *);
  probe reverse-connect (int, char *, char *);
  probe reverse-connect-success (int, char *, char *);
  probe reverse-connect-close (int, char *, char *, int, int);
  probe reverse-connect-failed (int, char *, char *, int);
  probe reverse-connect-ssl (int, char *, char *);
  probe reverse-connect-ssl-success (int, char *, char *);
  probe reverse-connect-ssl-failed (int, char *, char *, char *, int);
};
```

HTTP Server

```
provider libmtev {
    probe http-accept (int, struct mtev_http_session_ctx *);
    probe http-request-start (int, struct mtev_http_session_ctx *);
    probe http-request-finish (int, struct mtev_http_session_ctx *);
    probe http-response-start (int, struct mtev_http_session_ctx *);
    probe http-response-finish (int, struct mtev_http_session_ctx *);
    probe http-log (int, struct mtev_http_session_ctx *, char *);
    probe http-close (int, struct mtev_http_session_ctx *);
  };
```

luamtev

libmtev takes Lua seriously. We're specific about supporting LuaJIT and while LuaJIT is a very powerful runtime, we felt we'd get more power by exposing the LuaJIT runtime via a standalone libmtev application; luamtev was born.

luamtev is a non-interactive interpreter for luacode. Unlike the normal LuaJIT interpreter, it only runs modules, but it first boots a comprehensive libmtev runtime and then runs the provided module within that runtime. This allows use of all of the advanced features of libmtev.

mtev-busted

mtev-busted is simply some subset of the excellent busted testing framework running under luamtev and leveraging the non-blocking libmtev runtime.

Programmer's Reference Manual

C

Lua

A

mtev_amqp_send

> Publish an AMQP message to one of the configured amqp brokers.

```
void
mtev_amqp_send(struct amqp_envelope_t_ *env, int mandatory, int immediate, int id)
```

- `env` An envelope with a valid message. The env pointer must be word aligned.
- `mandatory` Set to non-zero if the message should be sent with the mandatory flag.
- `immediate` Set to non-zero if the message should be sent with the immediate flag.
- `id` the ID of the connection: -1 to broadcast.

mtev_amqp_send_data

> Publish an AMQP message to one of the configured amqp brokers.

```
void
mtev_amqp_send_data(char *exchange, char *route, int mandatory, int immediate, void *payload,
                    int len, int id)
```

- `exchange` The AMQP exchange to publish to.
- `route` The route to set on the message.
- `mandatory` Set to non-zero if the message should be sent with the mandatory flag.
- `immediate` Set to non-zero if the message should be sent with the immediate flag.
- `payload` the contents of the message.
- `len` the number of bytes present in payload.
- `id` the ID of the connection: -1 to broadcast.

B

mtev_b32_decode

> Decode a base32 encoded input buffer into the provided output buffer.

```
int
mtev_b32_decode(const char *src, size_t src_len, unsigned char *dest, size_t dest_len)
```

- `src` The buffer containing the encoded content.
- `src_len` The size (in bytes) of the encoded data.
- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the decoded output. Returns zero is dest_len is too small.

mtev_b32_decode decodes input until an the entire input is consumed or until an invalid base32 character is encountered.

mtev_b32_encode

> Encode raw data as base32 encoded output into the provided buffer.

```
int
```

```
mtev_b32_encode(const unsigned char *src, size_t src_len, char *dest, size_t dest_len)
```

- `src` The buffer containing the raw data.
- `src_len` The size (in bytes) of the raw data.
- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the encoded output. Returns zero is out_sz is too small.

mtev_b32_encode_len

> Calculate how large a buffer must be to contain the base-32 encoding for a given number of bytes.

```
size_t
mtev_b32_encode_len(size_t src_len)
```

- `src_len` The size (in bytes) of the raw data buffer that might be encoded.
- **RETURN** The size of the buffer that would be needed to store an encoded version of an input string.

mtev_b32_max_decode_len

> Calculate how large a buffer must be to contain a decoded base-32-encoded string of a given length.

```
size_t
mtev_b32_max_decode_len(size_t src_len)
```

- `src_len` The size (in bytes) of the base-32-encoded string that might be decoded.
- **RETURN** The size of the buffer that would be needed to decode the input string.

mtev_b64_decode

> Decode a base64 encoded input buffer into the provided output buffer.

```
int
mtev_b64_decode(const char *src, size_t src_len, unsigned char *dest, size_t dest_len)
```

- `src` The buffer containing the encoded content.
- `src_len` The size (in bytes) of the encoded data.
- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the decoded output. Returns zero is dest_len is too small.

mtev_b64_decode decodes input until an the entire input is consumed or until an invalid base64 character is encountered.

mtev_b64_encode

> Encode raw data as base64 encoded output into the provided buffer.

```
int
mtev_b64_encode(const unsigned char *src, size_t src_len, char *dest, size_t dest_len)
```

- `src` The buffer containing the raw data.
- `src_len` The size (in bytes) of the raw data.

- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the encoded output. Returns zero is out_sz is too small.

mtev_b64_encode encodes an input string into a base64 representation with no linefeeds.

mtev_b64_encode_len

> Calculate how large a buffer must be to contain the base-64 encoding for a given number of bytes.

```
size_t
mtev_b64_encode_len(size_t src_len)
```

- `src_len` The size (in bytes) of the raw data buffer that might be encoded.
- **RETURN** The size of the buffer that would be needed to store an encoded version of an input string.

mtev_b64_encodev

> Encode raw data as base64 encoded output into the provided buffer.

```
int
mtev_b64_encodev(const struct iovec *iov, size_t iov_len, char *dest, size_t dest_len)
```

- `iov` The io-vectors containing the raw data.
- `iovcnt` The number of io-vectors.
- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the encoded output. Returns zero is out_sz is too small.

mtev_b64_encodev encodes an input string into a base64 representation with no linefeeds.

mtev_b64_max_decode_len

> Calculate how large a buffer must be to contain a decoded base-64-encoded string of a given length.

```
size_t
mtev_b64_max_decode_len(size_t src_len)
```

- `src_len` The size (in bytes) of the base-64-encoded string that might be decoded.
- **RETURN** The size of the buffer that would be needed to decode the input string.

C

callback

> Get the time of the last invoked callback in this thread.

```
eventer_gettimeofeventer_impl_data_t *t,
callback(struct timeval *now, void *tzp)
```

- `t` is the thread-local eventer data, if unknown pass NULL
- `now` a `struct timeval` to populate with the request time.
- `tzp` is ignored and for API compatibility with gettimeofday.

- **RETURN** 0 on success, non-zero on failure.

This function returns the time of the last callback execution. It is fast and cheap (cheaper than gettimeofday), so if a function wishes to know what time it is and the "time of invocation" is good enough, this is considerably cheaper than a call to `mtev_gettimeofday` or other system facilities.

mtev_cluster_alive_filter

> A `mtev_cluster_node_filter_func_t` for alive nodes.

```
mtev_cluster_node_filter_func_t
mtev_cluster_alive_filter
```

This function is available to be passed as the `filter` argument to `mtev_cluster_filter_owners`.

mtev_cluster_am_i_oldest_node

> Determines if the local node is the oldest node within the cluster.

```
mtev_boolean
mtev_cluster_am_i_oldest_node(const mtev_cluster_t *cluster)
```

- `cluster` The cluster in question.
- **RETURN** Returns mtev_false if there is a node in the cluster with a higher up-time than this one.

mtev_cluster_by_name

> Find the cluster with the registered name.

```
mtev_cluster_t *
mtev_cluster_by_name(const char *name)
```

- `name` The name of the cluster.
- **RETURN** Returns a pointer to the cluster or NULL is not found.

Takes a name and finds a globally registered cluster by that name.

mtev_cluster_do_i_own

> Determines if the local node should possess a given key based on internal CHTs.

```
mtev_boolean
mtev_cluster_do_i_own(mtev_cluster_t *cluster, void *key, size_t klen, int w)
```

- `cluster` The cluster in question.
- `key` A pointer to the key.
- `klen` The length, in bytes, of the key.
- `w` The number of nodes that are supposed to own this key.
- **RETURN** Returns mtev_true or mtev_false based on ownership status.

This function determines if the local node is among the w nodes in this cluster that should own the specified key.

mtev_cluster_enabled

> Report on the availability of the clusters feature.

```
mtev_boolean
mtev_cluster_enabled()
```

- **RETURN** mtev_true if clusters can be configured, otherwise mtev_false.

mtev_cluster_filter_owners

> Determines if the local node should possess a given key based on internal CHTs.

```
mtev_boolean
mtev_cluster_filter_owners(mtev_cluster_t *cluster, void *key, size_t klen, mtev_cluster_node_t **set,
                           int *w, mtev_cluster_node_filter_func_t filter, void *closure)
```

- `cluster` The cluster in question.
- `key` A pointer to the key.
- `klen` The length, in bytes, of the key.
- `set` A caller allocated array of at least *w length.
- `w` The number of nodes that are supposed to own this key, updated to set length that matches filter.
- `filter` The function used to qualify nodes.
- `closure` A user supplied value that is passed to the filter function.
- **RETURN** Returns mtev_true or mtev_false if set[0] is this node.

This function populates a set of owners for a key, but first filters them according to a user-specified function.

mtev_cluster_find_node

> Find a node by uuid within a cluster.

```
mtev_cluster_node_t *
mtev_cluster_find_node(mtev_cluster_t *cluster, uuid_t nodeid)
```

- `cluster` The " containing the node.
- `nodeid` The nodeid being searched for.
- **RETURN** Returns a pointer to the mtev_cluster_node_t or NULL if not found.

Takes a cluster and a node UUID and returns a pointer to the corresponding mtev_cluster_node_t.

mtev_cluster_get_config_seq

> Returns the current config sequence of the given cluster

```
int64_t
mtev_cluster_get_config_seq(mtev_cluster_t *cluster)
```

- `cluster` The cluster in question, may not be NULL.

This function returns the current config sequence of the given cluster

mtev_cluster_get_heartbeat_payload

> Gets the current value of a payload segment from a node.

```
int
mtev_cluster_get_heartbeat_payload(mtev_cluster_t *cluster, uint8_t app_id, uint8_t key, void **payload)
```

- `cluster` The cluster in question, may not be NULL.
- `app_id` Used to identify the application that attached the payload.
- `key` Used to identify the payload amongst other payloads from the application.
- `payload` Pointer to a payload pointer.
- **RETURN** The length of the payload, -1 if that payload segment does not exist.

mtev_cluster_get_my_boot_time

> Returns the boot time of the local node.

```
struct timeval
mtev_cluster_get_my_boot_time()
```

- **RETURN** The boot time of the local node.

mtev_cluster_get_name

> Returns the name of the cluster.

```
const char *
mtev_cluster_get_name(mtev_cluster_t *cluster)
```

- `cluster` a cluster
- **RETURN** A pointer to the cluster's name.

mtev_cluster_get_node

> Find a node in a cluster by id.

```
mtev_cluster_node_t *
mtev_cluster_get_node(mtev_cluster_t *cluster, uuid_t id)
```

- `cluster` The cluster in question.
- `id` The uuid of the node in question.
- **RETURN** An `mtev_cluster_node_t *` if one is found with the provided id, otherwise NULL,

mtev_cluster_get_nodes

> Reports all nodes in the cluster (possible excluding the local node)

```
int
mtev_cluster_get_nodes(mtev_cluster_t *cluster, mtev_cluster_node_t **nodes, int n
                       mtev_boolean includeme)
```

- `cluster` The cluster in question.
- `nodes` The destination array to which a node list will be written.
- `n` The number of positions available in the passed nodes array.
- `includeme` Whether the local node should included in the list.
- **RETURN** Returns the number of nodes populated in the supplied nodes array. If insufficient space is available, a negative

value is returned whose absolute value indicates the required size of the input array.

Enumerates the nodes in a cluster into a provided nodes array.

mtev_cluster_get_oldest_node

> Returns the oldest node within the given cluster.

```
mtev_cluster_get_oldest_node(const mtev_cluster_t *cluster)
```

- `cluster` The cluster in question.
- **RETURN** Returns the node in the given cluster with the highest up-time.

mtev_cluster_get_self

> Reports the UUID of the local node.

```
void
mtev_cluster_get_self(uuid_t id)
```

- `id` The UUID to be updated.
- **RETURN** Returns -1 on error

Pouplates the passed uuid_t with the local node's UUID.

mtev_cluster_init

> Initialize the mtev cluster configuration.

```
void
mtev_cluster_init()
```

Initializes the mtev cluster configuration.

mtev_cluster_node_get_addr

```
int8_t
mtev_cluster_node_get_addr(mtev_cluster_node_t *node, struct sockaddr **addr, socklen_t *addrlen)
```

mtev_cluster_node_get_boot_time

```
struct timeval
mtev_cluster_node_get_boot_time(mtev_cluster_node_t *node)
```

```
* **RETURN** boot time as timeval struct
```

mtev_cluster_node_get_cn

```
const char*
mtev_cluster_node_get_cn(mtev_cluster_node_t *node)
```

- **RETURN** cn (canonical name) of the cluster node

mtev_cluster_node_get_config_seq

```
int64_t
mtev_cluster_node_get_config_seq(mtev_cluster_node_t *node)
```

mtev_cluster_node_get_id

> Retrieve the ID of a cluster node.

```
void
mtev_cluster_node_get_id(mtev_cluster_node_t *node, uuid_t out)
```

- `node` The node in question.
- `out` A `uuid_t` to fill in.

mtev_cluster_node_get_idx

> Get the unique integer idx of the node within it's cluster.

```
int
mtev_cluster_node_get_idx(mtev_cluster_node_t *node)
```

- `node` The node in question
- **RETURN** A number between 0 and cluster_size - 1.

mtev_cluster_node_get_last_contact

```
struct timeval
mtev_cluster_node_get_last_contact(mtev_cluster_node_t *node)
```

- **RETURN** time of last contact to the given node

mtev_cluster_node_has_payload

> Determine a cluster node has a custom payload attached.

```
mtev_boolean
mtev_cluster_node_has_payload(mtev_cluster_node_t *node)
```

- `node` The node in question.
- **RETURN** True if there is a payload, false otherwise.

mtev_cluster_node_is_dead

> Detrmines if the node in question is dead.

```
mtev_boolean
mtev_cluster_node_is_dead(mtev_cluster_node_t *node)
```

- `node` The node in question.
- **RETURN** Returns true if the node is dead.

mtev_cluster_set_heartbeat_payload

> Triggers the attachment of an arbitrary payload to the cluster heartbeats (see mtev_cluster_handle_node_update)

```
void
mtev_cluster_set_heartbeat_payload(mtev_cluster_t *cluster, uint8_t app_id, uint8_t key, void* payload
                                   uint8_t payload_length)
```

- `cluster` The cluster in question, may not be NULL.
- `app_id` Used to identify the application that attached the payload.
- `key` Used to identify the payload amongst other payloads from the application.
- `payload` A pointer to the payload that should be attached to every heartbeat message.
- `payload_length` The number of bytes to be read from payload.
- **RETURN** Returns mtev_true if the payload was not enabled yet

This function triggers the attachment of an arbitrary payload to the cluster heartbeats (see mtev_cluster_get_payload)

mtev_cluster_set_node_update_callback

> Sets a callback which is called everytime a node in the cluster changes it's up-time.

```
int
mtev_cluster_set_node_update_callback(mtev_cluster_t *cluster, mtev_cluster_node_update_cb callback)
```

- `cluster` The cluster in question.
- `callback` Function pointer to the function that should be called.
- **RETURN** Returns mtev_true if the cluster is not NULL, mtev_false otherwise

mtev_cluster_set_self

> Sets the UUID of the local node.

```
void
mtev_cluster_set_self(uuid_t id)
```

- `id` The UUID.

Sets the local node's cluster identity, potentially updating the on-disk configuration.

mtev_cluster_size

> Report the number of nodes in the cluster.

```
int
mtev_cluster_size(mtev_cluster_t *cluster)
```

- `cluster` The cluster.
- **RETURN** The number of nodes in the cluster.

Determines the number of nodes in the given cluster.

mtev_cluster_unset_heartbeat_payload

> Detaches (clears) an arbitrary payload to the cluster heartbeats (see mtev_cluster_handle_node_update)

```
void
mtev_cluster_unset_heartbeat_payload(mtev_cluster_t *cluster, uint8_t app_id, uint8_t key)
```

- `cluster` The cluster in question, may not be NULL.
- `app_id` Used to identify the application that attached the payload.
- `key` Used to identify the payload amongst other payloads from the application.

mtev_cluster_update

> Add or update an mtev cluster.

```
int
mtev_cluster_update(mtev_conf_section_t cluster)
```

- `cluster` The " node configuration.
- **RETURN** Returns -1 on error, 0 on insert, or 1 on update.

Takes a configuration section representing a cluster and registers it in the global cluster configuration.

mtev_confstr_parse_duration

```
int
mtev_confstr_parse_duration(const char *input, uint64_t *output
                            const mtev_duration_definition_t *durations)
```

- `input` String representing a duration.
- `output` On successful parsing, filled in with the duration corresponding to `input` .
- `durations` Describes allowable duration suffixes when parsing.
- **RETURN** One of:
  -
    - MTEV_CONFSTR_PARSE_SUCCESS (`input` was parsed successfully, `output` filled in)
  -
    - MTEV_CONFSTR_PARSE_ERR_FORMAT (`input` was not well-formed.)

Parses a string representing a duration. The string should be formatted as a set of (optionally) white-space separated duration elements, where a duration element is a number with a resolution suffix. For example, `"1s"` is a duration element representing one second, while `"3min"` is a duration element representing three minutes. The total duration is calculated by adding together all the duration elements. For example, `"1min 30sec"` , with resolution in seconds, would result in `output` of `90` ; and `"1min5ms"` , at millisecond resolution, would result in `output` of `60005` .

mtev_confstr_parse_duration_ms

```
int
mtev_confstr_parse_duration_ms(const char *input, uint64_t *output)
```

Convenience function for parsing a duration with resolution in milliseconds. See mtev_confstr_parse_duration and mtev_get_durations_ms.

mtev_confstr_parse_duration_ns

```
int
mtev_confstr_parse_duration_ns(const char *input, uint64_t *output)
```

Convenience function for parsing a duration with resolution in nanoseconds. See mtev_confstr_parse_duration and mtev_get_durations_ns.

mtev_confstr_parse_duration_s

```
int
mtev_confstr_parse_duration_s(const char *input, uint64_t *output)
```

Convenience function for parsing a duration with resolution in seconds. See mtev_confstr_parse_duration and mtev_get_durations_s.

mtev_confstr_parse_duration_us

```
int
mtev_confstr_parse_duration_us(const char *input, uint64_t *output)
```

Convenience function for parsing a duration with resolution in microseconds. See mtev_confstr_parse_duration and mtev_get_durations_us.

mtev_curl_write_callback

```
size_t
mtev_curl_write_callback(char *ptr, size_t size, size_t nmemb, void *userdata)
```

*

> A function to pass as curls CURLOPT_WRITEFUNCTION

D

mtev_dyn_buffer_add

> add data to the dyn_buffer.

```
void
mtev_dyn_buffer_add(mtev_dyn_buffer_t *buf, uint8_t *data, size_t len)
```

- `buf` the buffer to add to.
- `data` the data to add.
- `len` the size of the data to add.

mtev_dyn_buffer_add_json_string

> add data to the dyn_buffer as an unquoted json-encoded string.

```
void
```

```
mtev_dyn_buffer_add_json_string(mtev_dyn_buffer_t *buf, uint8_t *data, size_t len)
```

- `buf` the buffer to add to.
- `data` the data to add.
- `len` the size of the data to add.
- `sol` 1 to escape the solipsis, 0 otherwise.

mtev_dyn_buffer_add_printf

> add data to the dyn_buffer using printf semantics.

```
void
mtev_dyn_buffer_add_printf(mtev_dyn_buffer_t *buf, const char *format, ...)
```

- `buf` the buffer to add to.
- `format` the printf style format string
- `args` printf arguments

This does NUL terminate the format string but does not advance the write_pointer past the NUL. Basically, the last mtev_dyn_buffer_add_printf will leave the resultant data NUL terminated.

mtev_dyn_buffer_advance

> move the write_pointer forward len bytes

```
void
mtev_dyn_buffer_advance(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to advance

mtev_dyn_buffer_data

> return the front of the dyn_buffer

```
void
mtev_dyn_buffer_data(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to get the pointer from.
- **RETURN** the pointer to the front (beginning) of the dyn_buffer

mtev_dyn_buffer_destroy

> destroy the dyn_buffer

```
void
mtev_dyn_buffer_destroy(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to destroy

  This must be called at the end of dyn_buffer interactions in case the buffer has overflowed into dynamic allocation space.

mtev_dyn_buffer_ensure

> possibly grow the dyn_buffer so it can fit len bytes

```
void
mtev_dyn_buffer_ensure(mtev_dyn_buffer_t *buf, size_t len)
```

- `buf` the buffer to ensure
- `len` the size of the data about to be added

mtev_dyn_buffer_init

> initialize a dyn_buffer

```
void
mtev_dyn_buffer_init(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to init

  Provided for completeness or non-stack allocations.

mtev_dyn_buffer_reset

> move the write position to the beginning of the buffer

```
void
mtev_dyn_buffer_reset(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to reset.

mtev_dyn_buffer_size

> return the total size of the buffer

```
void
mtev_dyn_buffer_size(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to get the size from.
- **RETURN** the total size of the buffer

mtev_dyn_buffer_used

> return the total used space of the buffer

```
void
mtev_dyn_buffer_used(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to get the used space from.
- **RETURN** the total used space of the buffer

mtev_dyn_buffer_write_pointer

> return the end of the dyn_buffer

```
void
```

```
mtev_dyn_buffer_write_pointer(mtev_dyn_buffer_t *buf)
```

- `buf` the buffer to get the pointer from.
- **RETURN** the pointer to the end of the dyn_buffer

E

eventer_accept

> Execute an opset-appropriate `accept` call.

```
int
eventer_accept(eventer_t e, struct sockaddr *addr, socklen_t *len, int *mask)
```

- `e` an event object
- `addr` a `struct sockaddr` to be populated.
- `len` a `socklen_t` pointer to the size of the `addr` argument; updated.
- `mask` a point the a mask. If the call does not complete, `*mask` it set.
- **RETURN** an opset-appropriate return value. (fd for POSIX, -1 for SSL).

If the function returns -1 and `errno` is `EAGAIN` , the `*mask` reflects the necessary activity to make progress.

eventer_aco_accept

> Execute an opset-appropriate `accept` call.

```
int
eventer_aco_accept(eventer_t e, struct sockaddr *addr, socklen_t *len, struct timeval *timeout)
```

- `e` an event object
- `addr` a `struct sockaddr` to be populated.
- `len` a `socklen_t` pointer to the size of the `addr` argument; updated.
- `timeout` if not NULL, the time after which we fail -1, ETIME
- **RETURN** an opset-appropriate return value. (fd for POSIX, -1 for SSL).

eventer_aco_arg

> Gets the argument used to start an aco coroutine.

```
void *
eventer_aco_arg(void)
```

- **RETURN** The closure parameter that was passed to `eventer_aco_start` .

eventer_aco_asynch

> Asynchronously execute a function.

```
void
eventer_aco_asynch(eventer_asynch_func_t func, void *closure)
```

- `func` the function to execute.

- `closure` the closure for the function.

eventer_aco_asynch_gated

> Asynchronously execute a function.

```
void
eventer_aco_asynch_gated(eventer_aco_gate_t gate, eventer_asynch_func_t func, void *closure)
```

- `gate` a gate to notify on completion.
- `func` the function to execute.
- `closure` the closure for the function.

eventer_aco_asynch_queue

> Asynchronously execute a function.

```
void
eventer_aco_asynch_queue(eventer_asynch_func_t func, void *closure, eventer_jobq_t *q)
```

- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.

eventer_aco_asynch_queue_gated

> Asynchronously execute a function.

```
void
eventer_aco_asynch_queue_gated(eventer_aco_gate_t gate, eventer_asynch_func_t func, void *closure
                               eventer_jobq_t *q)
```

- `gate` a gate to notify on completion.
- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.

eventer_aco_asynch_queue_subqueue

> Asynchronously execute a function.

```
void
eventer_aco_asynch_queue_subqueue(eventer_asynch_func_t func, void *closure, eventer_jobq_t *q, uint64_t id)
```

- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.
- `id` the subqueue within the jobq.

eventer_aco_asynch_queue_subqueue_deadline

> Asynchronously execute a function.

```
void
eventer_aco_asynch_queue_subqueue_deadline(eventer_asynch_func_t func, void *closure, eventer_jobq_t *q, uint64
_t id
                                           struct timeval *whence)
```

- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.
- `id` the subqueue within the jobq.
- `whence` the deadline

eventer_aco_asynch_queue_subqueue_deadline_gated

> Asynchronously execute a function.

```
void
eventer_aco_asynch_queue_subqueue_deadline_gated(eventer_aco_gate_t gate, eventer_asynch_func_t func, void *clo
sure,
                                                 eventer_jobq_t *q, uint64_t id, struct timeval *whence)
```

- `gate` a gate to notify on completion
- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.
- `id` the subqueue within the jobq.
- `whence` the deadline

eventer_aco_asynch_queue_subqueue_gated

> Asynchronously execute a function.

```
void
eventer_aco_asynch_queue_subqueue_gated(eventer_aco_gate_t gate, eventer_asynch_func_t func, void *closure,
                                        eventer_jobq_t *q, uint64_t id)
```

- `gate` a gate to notify on completion.
- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.
- `id` the subqueue within the jobq.

eventer_aco_close

> Execute an opset-appropriate `close` call.

```
int
eventer_aco_close(eventer_aco_t e)
```

- `e` an event object
- **RETURN** 0 on sucess or -1 with errno set.

eventer_aco_free

> Dereferences the event specified.

```
void
eventer_aco_free(eventer_aco_t e)
```

- `e` the event to dereference.

eventer_aco_gate

> Create a new asynchronous gate.

```
eventer_aco_gate_t
eventer_aco_gate(void)
```

eventer_aco_gate_wait

> Wait for any asynchronous work on this gate to finish.

```
void
eventer_aco_gate_wait(eventer_aco_gate_t gate)
```

- `gate` an asynchronous gate

eventer_aco_get_closure

> Retrieve an event's closure.

```
void *
eventer_aco_get_closure(eventer_aco_t e)
```

- `e` an event object
- **RETURN** The previous closure set.

eventer_aco_read

> Execute an opset-appropriate `read` call.

```
int
eventer_aco_read(eventer_aco_t e, void *buff, size_t len, struct timeval *timeout)
```

- `e` an event object
- `buff` a buffer in which to place read data.
- `len` the size of `buff` in bytes.
- `timeout` if not NULL, the time after which we fail -1, ETIME
- **RETURN** the number of bytes read or -1 with errno set.

eventer_aco_run_asynch

> Add an asynchronous event dependent on the current job and wait until completion.

```
mtev_boolean
eventer_aco_run_asynch(eventer_t e)
```

- `e` an event object
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the default job queue. `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_aco_run_asynch_gated

> Add an asynchronous event dependent on the current job and signal gate on completion.

```
mtev_boolean
eventer_aco_run_asynch_gated(eventer_aco_gate_t gate, eventer_t e)
```

- `gate` a gate
- `e` an event object
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the default job queue. `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_aco_run_asynch_queue

> Add an asynchronous event to a specific job queue dependent on the current job and wait until completion.

```
mtev_boolean
eventer_aco_run_asynch_queue(eventer_jobq_t *q, eventer_t e)
```

- `q` a job queue
- `e` an event object
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q` . `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_aco_run_asynch_queue_gated

> Add an asynchronous event to a specific job queue dependent on the current job and signal gate on completion.

```
mtev_boolean
eventer_aco_run_asynch_queue_gated(eventer_aco_gate_t gate, eventer_jobq_t *q, eventer_t e)
```

- `gate` a gate
- `q` a job queue
- `e` an event object
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q` . `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_aco_run_asynch_queue_subqueue

> Add an asynchronous event to a specific job queue dependent on the current job and wait until completion.

```
mtev_boolean
```

```
eventer_aco_run_asynch_queue_subqueue(eventer_jobq_t *q, eventer_t e, uint64_t id)
```

- `q` a job queue
- `e` an event object
- `id` is a fairly competing subqueue identifier
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q` . `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_aco_run_asynch_queue_subqueue_gated

> Add an asynchronous event to a specific job queue dependent on the current job and signal gate on completion.

```
mtev_boolean
eventer_aco_run_asynch_queue_subqueue_gated(eventer_aco_gate_t gate, eventer_jobq_t *q, eventer_t e, uint64_t i
d)
```

- `gate` a gate
- `q` a job queue
- `e` an event object
- `id` is a fairly competing subqueue identifier
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q` . `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_aco_set_accept_timeout

> Change the default timeout for ACO events.

```
void
eventer_aco_set_accept_timeout(eventer_aco_t e, struct timeval *duration)
```

- `e` the ACO event to update.
- `duration` a timeout duration, NULL will undo the default.

eventer_aco_set_closure

> Set an event's closure.

```
void
eventer_aco_set_closure(eventer_aco_t e, void *closure)
```

- `e` an event object
- `closure` a pointer to user-data to be supplied during callback.

eventer_aco_set_read_timeout

> Change the default timeout for ACO events.

```
void
eventer_aco_set_read_timeout(eventer_aco_t e, struct timeval *duration)
```

- `e` the ACO event to update.
- `duration` a timeout duration, NULL will undo the default.

eventer_aco_set_write_timeout

> Change the default timeout for ACO events.

```
void
eventer_aco_set_write_timeout(eventer_aco_t e, struct timeval *duration)
```

- `e` the ACO event to update.
- `duration` a timeout duration, NULL will undo the default.

eventer_aco_simple_asynch

> Asynchronously execute a function.

```
void
eventer_aco_simple_asynch(eventer_asynch_simple_func_t func, void *closure)
```

- `func` the function to execute.
- `closure` the closure for the function.

eventer_aco_simple_asynch_gated

> Asynchronously execute a function.

```
void
eventer_aco_simple_asynch_gated(eventer_aco_gate_t gate, eventer_asynch_simple_func_t func, void *closure)
```

- `gate` a gate to notify on completion.
- `func` the function to execute.
- `closure` the closure for the function.

eventer_aco_simple_asynch_queue

> Asynchronously execute a function.

```
void
eventer_aco_simple_asynch_queue(eventer_asynch_simple_func_t func, void *closure, eventer_jobq_t *q)
```

- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.

eventer_aco_simple_asynch_queue_gated

> Asynchronously execute a function.

```
void
eventer_aco_simple_asynch_queue_gated(eventer_aco_gate_t gate, eventer_asynch_simple_func_t func, void *closure
                                      eventer_jobq_t *q)
```

- `gate` a gate to notify on completion.
- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.

eventer_aco_simple_asynch_queue_subqueue

> Asynchronously execute a function.

```
void
eventer_aco_simple_asynch_queue_subqueue(eventer_asynch_simple_func_t func, void *closure, eventer_jobq_t *q
                                         uint64_t id)
```

- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.
- `id` the subqueue within the jobq.

eventer_aco_simple_asynch_queue_subqueue_gated

> Asynchronously execute a function.

```
void
eventer_aco_simple_asynch_queue_subqueue_gated(eventer_aco_gate_t gate, eventer_asynch_simple_func_t func, void
 *closure,
                                               eventer_jobq_t *q, uint64_t id)
```

- `gate` a gate to notify on completion.
- `func` the function to execute.
- `closure` the closure for the function.
- `q` the jobq on which to schedule the work.
- `id` the subqueue within the jobq.

eventer_aco_sleep

> Execute a sleep within an aco context.

```
void
eventer_aco_sleep(struct timeval *duration)
```

- `duration` the time to suspend.

eventer_aco_start

> Start a new aco coroutine to be eventer driven.

```
void
eventer_aco_start(void (*func)(void), void *closure)
```

- `func` The function to start.
- `closure` The closure to set (available within `func` via `eventer_aco_arg()`)

eventer_aco_start_stack

> Start a new aco coroutine to be eventer driven.

```
void
eventer_aco_start_stack(void (*func)(void), void *closure, size_t stksz)
```

- `func` The function to start.
- `closure` The closure to set (available within `func` via `eventer_aco_arg()`)
- `stksz` A specified maximum stack size other than the default 32k.

eventer_aco_try_run_asynch_queue_subqueue

> Add an asynchronous event to a specific job queue dependent on the current job and wait until completion.

```
mtev_boolean
eventer_aco_try_run_asynch_queue_subqueue(eventer_jobq_t *q, eventer_t e, uint64_t id)
```

- `q` a job queue
- `e` an event object
- `id` is a fairly competing subqueue identifier
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q` . `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_aco_write

> Execute an opset-appropriate `write` call.

```
int
eventer_aco_write(eventer_aco_t e, const void *buff, size_t len, struct timeval *timeout)
```

- `e` an event object
- `buff` a buffer containing data to write.
- `len` the size of `buff` in bytes.
- `timeout` if not NULL, the time after which we fail -1, ETIME
- **RETURN** the number of bytes written or -1 with errno set.

eventer_add

> Add an event object to the eventer system.

```
void
eventer_add(eventer_t e)
```

- `e` an event object to add.

eventer_add_asynch

> Add an asynchronous event to a specific job queue.

```
void
eventer_add_asynch(eventer_jobq_t *q, eventer_t e)
```

- `q` a job queue
- `e` an event object

This adds the `e` event to the job queue `q`. `e` must have a mask of `EVENTER_ASYNCH`.

eventer_add_asynch_dep

> Add an asynchronous event to a specific job queue dependent on the current job.

```
void
eventer_add_asynch_dep(eventer_jobq_t *q, eventer_t e)
```

- `q` a job queue
- `e` an event object

This adds the `e` event to the job queue `q`. `e` must have a mask of `EVENTER_ASYNCH`. This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_add_asynch_dep_subqueue

> Add an asynchronous event to a specific job queue dependent on the current job.

```
void
eventer_add_asynch_dep_subqueue(eventer_jobq_t *q, eventer_t e, uint64_t id)
```

- `q` a job queue
- `e` an event object
- `id` is a fairly competing subqueue identifier

This adds the `e` event to the job queue `q`. `e` must have a mask of `EVENTER_ASYNCH`. This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_add_asynch_subqueue

> Add an asynchronous event to a specific job queue.

```
void
eventer_add_asynch_subqueue(eventer_jobq_t *q, eventer_t e, uint64_t id)
```

- `q` a job queue
- `e` an event object
- `id` is a fairly competing subqueue identifier

This adds the `e` event to the job queue `q`. `e` must have a mask of `EVENTER_ASYNCH`.

eventer_add_at

> Convenience function to schedule a callback at a specific time.

```
eventer_t
eventer_add_at(eventer_func_t func, void *closure, struct timeval whence)
```

- `func` the callback function to run.
- `closure` the closure to be passed to the callback.

- `whence` the time at which to run the callback.
- **RETURN** N/A (C Macro).

eventer_add_in

> Convenience function to create an event to run a callback in the future

```
eventer_t
eventer_add_in(eventer_func_t func, void *closure, struct timeval diff)
```

- `func` the callback function to run.
- `closure` the closure to be passed to the callback.
- `diff` the amount of time to wait before running the callback.
- **RETURN** N/A (C Macro).

eventer_add_in_s_us

> Convenience function to create an event to run a callback in the future

```
eventer_t
eventer_add_in_s_us(eventer_func_t func, void *closure, unsigned long seconds
                    unsigned long microseconds)
```

- `func` the callback function to run.
- `closure` the closure to be passed to the callback.
- `seconds` the number of seconds to wait before running the callback.
- `microseconds` the number of microseconds (in addition to `seconds`) to wait before running the callback.
- **RETURN** N/A (C Macro).

eventer_add_recurrent

> Add an event to run during every loop cycle.

```
void
eventer_add_recurrent(eventer_t e)
```

- `e` an event object

`e` must have a mask of EVENER*RECURRENT. This event will be invoked on a single thread (dictated by `e`) once for each pass through the eventer loop. This happens _often*, so do light work.

eventer_add_timed

> Add a timed event to the eventer system.

```
void
eventer_add_timed(eventer_t e)
```

- `e` an event object

This adds the `e` event to the eventer. `e` must have a mask of `EVENTER_TIMED` .

eventer_add_timer_next_opportunity

> Convenience function to schedule a callback to run in a specific event-loop thread.

```
eventer_t
eventer_add_timer_next_opportunity(eventer_func_t func, void *closure, pthread_t owner)
```

- `func` the callback function to run.
- `closure` the closure to be passed to the callback.
- `owner` the event-loop thread in which to run the callback.
- **RETURN** N/A (C Macro).

eventer_alloc

> Allocate an event to be injected into the eventer system.

```
eventer_t
eventer_alloc()
```

- **RETURN** A newly allocated event.

The allocated event has a refernce count of 1 and is attached to the calling thread.

eventer_alloc_asynch

> Allocate an event to be injected into the eventer system.

```
eventer_t
eventer_alloc_asynch(eventer_func_t func, void *closure)
```

- `func` The callback function.
- `closure` The closure for the callback function.
- **RETURN** A newly allocated asynch event.

The allocated event has a refernce count of 1 and is attached to the calling thread.

eventer_alloc_asynch_timeout

> Allocate an event to be injected into the eventer system.

```
eventer_t
eventer_alloc_asynch_timeout(eventer_func_t func, void *closure, struct timeval *deadline)
```

- `func` The callback function.
- `closure` The closure for the callback function.
- `deadline` an absolute time by which the task must be completed.
- **RETURN** A newly allocated asynch event.

The allocated event has a refernce count of 1 and is attached to the calling thread. Depending on the timeout method, there are not hard guarantees on enforcing the deadline; this is more of a guideline for the schedule and the job could be aborted (where the `EVENTER_ASYNCH_WORK` phase is not finished or even started, but the `EVENTER_ASYNCH_CLEANUP` will be called).

eventer_alloc_copy

> Allocate an event copied from another to be injected into the eventer system.

```
eventer_t
eventer_alloc_copy(eventer_t src)
```

- `src` a source eventer_t to copy.
- **RETURN** A newly allocated event that is a copy of src.

The allocated event has a refernce count of 1.

eventer_alloc_fd

> Allocate an event to be injected into the eventer system.

```
eventer_t
eventer_alloc_fd(eventer_func_t func, void *closure, int fd, int mask)
```

- `func` The callback function.
- `closure` The closure for the callback function.
- `fd` The file descriptor.
- `mask` The mask of activity of interest.
- **RETURN** A newly allocated fd event.

The allocated event has a refernce count of 1 and is attached to the calling thread.

eventer_alloc_recurrent

> Allocate an event to be injected into the eventer system.

```
eventer_t
eventer_alloc_recurrent(eventer_func_t func, void *closure)
```

- `func` The callback function.
- `closure` The closure for the callback function.
- **RETURN** A newly allocated recurrent event.

The allocated event has a refernce count of 1 and is attached to the calling thread.

eventer_alloc_timer

> Allocate an event to be injected into the eventer system.

```
eventer_t
eventer_alloc_timer(eventer_func_t func, void *closure, struct timeval *whence)
```

- `func` The callback function.
- `closure` The closure for the callback function.
- `whence` The time at which the event should fire.
- **RETURN** A newly allocated timer event.

The allocated event has a refernce count of 1 and is attached to the calling thread.

eventer_alloc_timer_next_opportunity

> Convenience function to create an event to run a callback on a specific thread.

```
eventer_t
eventer_alloc_timer_next_opportunity(eventer_func_t func, void *closure, pthread_t owner)
```

- `func` the callback function to run.
- `closure` the closure to be passed to the callback.
- `owner` the event-loop thread on which to run the callback.
- **RETURN** an event that has not been added to the eventer.

> Note this does not actually schedule the event. See `eventer_add_timer_next_opportunity`.

eventer_allocations_current

```
int64_t
eventer_allocations_current()
```

- **RETURN** the number of currently allocated eventer objects.

eventer_allocations_total

```
int64_t
eventer_allocations_total()
```

- **RETURN** the number of allocated eventer objects over the life of the process.

eventer_at

> Convenience function to create an event to run a callback at a specific time.

```
eventer_t
eventer_at(eventer_func_t func, void *closure, struct timeval whence)
```

- `func` the callback function to run.
- `closure` the closure to be passed to the callback.
- `whence` the time at which to run the callback.
- **RETURN** an event that has not been added to the eventer.

> Note this does not actually schedule the event. See `eventer_add_at`.

eventer_callback

> Directly invoke an event's callback.

```
int
eventer_callback(eventer_t e, int mask, void *closure, struct timeval *now)
```

- `e` an event object
- `mask` the mask that callback should be acting upon (see `eventer_get_mask`)
- `closure` the closure on which the callback should act
- `now` the time the callback should see as "now".
- **RETURN** The return value of the callback function as invoked.

This does not call the callback in the contexts of the eventloop. This means that should the callback return a mask, the event-loop will not interpret it and change state appropriately. The caller must respond appropriately to any return values.

eventer_callback_for_name

> Find an event callback function that has been registered by name.

```
evneter_func_t
eventer_callback_for_name(const char *name)
```

- `name` the name of the callback.
- **RETURN** the function pointer or NULL if no such callback has been registered.

eventer_callback_ms

> Get the milliseconds since epoch of the current callback invocation.

```
uint64_t
eventer_callback_ms()
```

- **RETURN** milliseconds since epoch of callback invocation, or current time.

eventer_callback_us

> Get the microseconds since epoch of the current callback invocation.

```
uint64_t
eventer_callback_us()
```

- **RETURN** microseconds since epoch of callback invocation, or current time.

eventer_choose_owner

> Find a thread in the default eventer pool.

```
pthread_t
eventer_choose_owner(int n)
```

- `n` an integer.
- **RETURN** a pthread_t of an eventer loop thread in the default eventer pool.

This return the first thread when 0 is passed as an argument. All non-zero arguments are spread across the remaining threads (if existent) as `n` modulo one less than the concurrency of the default event pool.

This is done because many systems aren't thread safe and can only schedule their work on a single thread (thread 1). By spreading all thread-safe workloads across the remaining threads we reduce potential overloading of the "main" thread.

To assign an event to a thread, use the result of this function to assign: `e->thr_owner`.

eventer_choose_owner_pool

> Find a thread in a specific eventer pool.

```
pthread_t
```

```
eventer_choose_owner_pool(eventer_pool_t *pool, int n)
```

- `pool` an eventer pool.
- `n` an integer.
- **RETURN** a pthread_t of an eventer loop thread in the specified evneter pool.

This function chooses a thread within the specified pool by taking `n` modulo the concurrency of the pool. If the default pool is speicified, special assignment behavior applies. See `eventer_choose_owner`.

To assign an event to a thread, use the result of this function to assign: `e->thr_owner`.

eventer_close

> Execute an opset-appropriate `close` call.

```
int
eventer_close(eventer_t e, int *mask)
```

- `e` an event object
- `mask` a point the a mask. If the call does not complete, `*mask` it set.
- **RETURN** 0 on sucess or -1 with errno set.

If the function returns -1 and `errno` is `EAGAIN`, the `*mask` reflects the necessary activity to make progress.

eventer_deref

> See eventer_free.

```
void
eventer_deref(eventer_t e)
```

- `e` the event to dereference.

eventer_fd_opset_get_accept

> Retrieve the accept function from an fd opset.

```
eventer_fd_accept_t
eventer_fd_opset_get_accept(eventer_fd_opset_t opset)
```

- `opset` an opset (see `eventer_get_fd_opset`)
- **RETURN** An eventer_fd_accept_t function

eventer_fd_opset_get_close

> Retrieve the close function from an fd opset.

```
eventer_fd_close_t
eventer_fd_opset_get_close(eventer_fd_opset_t opset)
```

- `opset` an opset (see `eventer_get_fd_opset`)
- **RETURN** An eventer_fd_close_t function

eventer_fd_opset_get_read

> Retrieve the read function from an fd opset.

```
eventer_fd_read_t
eventer_fd_opset_get_read(eventer_fd_opset_t opset)
```

- `opset` an opset (see `eventer_get_fd_opset`)
- **RETURN** An eventer_fd_read_t function

eventer_fd_opset_get_write

> Retrieve the write function from an fd opset.

```
eventer_fd_write_t
eventer_fd_opset_get_write(eventer_fd_opset_t opset)
```

- `opset` an opset (see `eventer_get_fd_opset`)
- **RETURN** An eventer_fd_write_t function

eventer_find_fd

> Find an event object in the eventer system by file descriptor.

```
eventer_t
eventer_find_fd(int e)
```

- `fd` a file descriptor
- **RETURN** the event object if it exists; NULL if not found.

eventer_foreach_fdevent

> Run a user-provided function over all registered file descriptor events.

```
void
eventer_foreach_fdevent(void (*fn)(eventer_t, void *), void *closure)
```

- `fn` a function to be called with each event and `closure` as its arguments.
- `closure` the second argument to be passed to `fn` .

eventer_foreach_timedevent

> Run a user-provided function over all registered timed events.

```
void
eventer_foreach_timedevent(void (*fn)(eventer_t, void *), void *closure)
```

- `fn` a function to be called with each event and `closure` as its arguments.
- `closure` the second argument to be passed to `fn` .

eventer_free

> Dereferences the event specified.

```
void
eventer_free(eventer_t e)
```

- `e` the event to dereference.

eventer_get_callback

> Retrieve the callback function for an event.

```
eventer_func_t
eventer_get_callback(eventer_t e)
```

- `e` an event object
- **RETURN** An `eventer_func_t` callback function.

eventer_get_closure

> Retrieve an event's closure.

```
void *
eventer_get_closure(eventer_t e)
```

- `e` an event object
- **RETURN** The previous closure set.

eventer_get_context

> Get a context for an event.

```
int
eventer_get_context(eventer_t e, int ctx_idx)
```

- `e` an event object
- `ctx_idx` is an idx returned from `eventer_register_context`
- **RETURN** The attached context.

eventer_get_epoch

> Find the start time of the eventer loop.

```
int
eventer_get_epoch(struct timeval *epoch)
```

- `epoch` a point to a `struct timeval` to fill out.
- **RETURN** 0 on success; -1 on failure (eventer loop not started).

eventer_get_fd

> Retrieve the file descriptor for an fd-based event.

```
int
eventer_get_fd(eventer_t e)
```

- `e` an event object
- **RETURN** a file descriptor.

eventer_get_fd_opset

> Retrieve the fd opset from an event.

```
eventer_fd_opset_t
eventer_get_fd_opset(eventer_t e)
```

- `e` an event object
- **RETURN** The currently active opset for a fd-based eventer_t.

eventer_get_mask

> Retrieve the mask for an event.

```
int
eventer_get_mask(eventer_t e)
```

- `e` an event object
- **RETURN** a mask of bitwise-or'd valued.
    - `EVENTER_READ` -- trigger/set when a file descriptor is readable.
    - `EVENTER_WRITE` -- trigger/set when a file descriptor is writeable.
    - `EVENTER_EXCEPTION` -- trigger/set problems with a file descriptor.
    - `EVENTER_TIMER` -- trigger/set at a specific time.
    - `EVENTER_RECURRENT` -- trigger/set on each pass through the event-loop.
    - `EVENTER_ASYNCH_COMPLETE` -- trigger from a non-event-loop thread, set upon completion.
    - `EVENTER_ASYNCH_WORK` -- set during asynchronous work.
    - `EVENTER_ASYNCH_CLEANUP` -- set during asynchronous cleanup.

eventer_get_owner

> Retrieve the thread that owns an event.

```
pthread_t
eventer_get_owner(eventer_t e)
```

- `e` an event object
- **RETURN** a `pthread_t` thread.

eventer_get_pool_for_event

> Determin which eventer pool owns a given event.

```
eventer_pool_t *
eventer_get_pool_for_event(eventer_t e)
```

- `e` an event object.
- **RETURN** the `eventer_pool_t` to which the event is scheduled.

eventer_get_this_event

> Get the eventer_t (if any) of which we are currently in the callback.

```
eventer_t
eventer_get_this_event(void)
```

- **RETURN** An eventer_t or NULL.

eventer_get_thread_name

> Retrieve a human-friendly name for an eventer thread.

```
const char *
eventer_get_thread_name(void)
```

- **RETURN** A thread name.

eventer_get_whence

> Retrieve the time at which a timer event will fire.

```
struct timeval
eventer_get_whence(eventer_t e)
```

- `e` an event object
- **RETURN** A absolute time.

eventer_heartbeat_deadline

> Return the remaining time before the watchdog timeout on this thread.

```
mtev_boolean
eventer_heartbeat_deadline(struct timeval *now, struct timeval *delta)
```

- `now` the current time (NULL means now)
- `delta` the relative time remaining before a watchdog
- **RETURN** mtev_true if the timeout was successfully calculated.

eventer_impl_propset

> Set properties for the event loop.

```
int
eventer_impl_propset(const char *key, const char *value)
```

- `key` the property
- `value` the property's value.
- **RETURN** 0 on success, -1 otherwise.

Sets propoerties within the eventer. That can only be called prior to `eventer_init` . See [Eventer configuuration)(../config/eventer.md) for valid properties.

eventer_impl_setrlimit

> Attempt to set the rlimit on allowable open files.

```
int
eventer_impl_setrlimit()
```

* **RETURN** the limit of the number of open files.

The target is the `rlim_nofiles` eventer config option. If that configuration option is unspecified, 1048576 is used.

eventer_in

> Convenience function to create an event to run a callback in the future

```
eventer_t
eventer_in(eventer_func_t func, void *closure, struct timeval diff)
```

* `func` the callback function to run.
* `closure` the closure to be passed to the callback.
* `diff` the amount of time to wait before running the callback.
* **RETURN** an event that has not been added to the eventer.

> Note this does not actually schedule the event. See `eventer_add_in`.

eventer_in_loop

> Determine if the current thread is an event loop thread.

```
mtev_boolean
eventer_in_loop(void)
```

* **RETURN** mtev_true if currently in an event loop thread, mtev_false otherwise.

eventer_in_s_us

> Convenience function to create an event to run a callback in the future

```
eventer_t
eventer_in_s_us(eventer_func_t func, void *closure, unsigned long seconds
                unsigned long microseconds)
```

* `func` the callback function to run.
* `closure` the closure to be passed to the callback.
* `seconds` the number of seconds to wait before running the callback.
* `microseconds` the number of microseconds (in addition to `seconds`) to wait before running the callback.
* **RETURN** an event that has not been added to the eventer.

> Note this does not actually schedule the event. See `eventer_add_in_s_us`.

eventer_init_globals

> Initialize global structures required for eventer operation.

```
void
eventer_init_globals()
```

This function is called by `mtev_main`. Developers should not need to call this function directly.

eventer_is_aco

> Determine if an event is in ACO mode.

```
mtev_boolean
eventer_is_aco(eventer_t e)
```

- `e` The eventer_t in question (NULL represent "current context")
- **RETURN** True if in ACO mode, false otherwise.

eventer_is_loop

> Determine if a thread is participating in the eventer loop.

```
int
eventer_is_loop(pthread_t tid)
```

- `tid` a thread
- **RETURN** 0 if the specified thread lives outside the eventer loop; 1 otherwise.

eventer_jobq_create

> Create a new jobq.

```
eventer_jobq_t *
eventer_jobq_create(const char *queue_name)
```

- `queue_name` a name for the new jobq
- **RETURN** a pointer to a new (or existing) jobq with that name. NULL on error.

eventer_jobq_create_backq

> Create a new jobq for use as a return queue.

```
eventer_jobq_t *
eventer_jobq_create_backq(const char *queue_name)
```

- `queue_name` a name for the new jobq
- **RETURN** a pointer to a new (or existing) jobq with that name. NULL on error.

eventer_jobq_create_ms

> Create a new jobq with the specified memory safety.

```
eventer_jobq_t *
eventer_jobq_create_ms(const char *queue_name, eventer_jobq_memory_safety_t safety)
```

- `queue_name` a name for the new jobq
- `safety` a specific mtev_memory safey level for epoch-based memory reclamation schemes.
- **RETURN** a pointer to a new (or existing) jobq with that name. NULL on error.

eventer_jobq_destroy

> Destory a jobq.

```
void
eventer_jobq_destroy(eventer_jobq_t *jobq)
```

eventer_jobq_inflight

> Reveal the currently executing job (visiable to a callee).

```
eventer_job_t *
eventer_jobq_inflight(void)
```

- **RETURN** the job that is currentlt running in the calling thread.

eventer_jobq_post

> Wake up a jobq to see if there are pending events.

```
void
eventer_jobq_post(eventer_jobq_t *jobq)
```

- `jobq` the jobq to post to.

eventer_jobq_retrieve

> Find a jobq by name.

```
eventer_jobq_t *
eventer_jobq_retrieve(const char *name)
```

- `name` the name of a jobq
- **RETURN** a jobq or NULL if no such jobq exists.

eventer_jobq_set_concurrency

> Set a jobq's concurrency level.

```
void
eventer_jobq_set_concurrency(eventer_jobq_t *jobq, uint32_t new_concurrency)
```

- `jobq` the jobq to modify
- `new_concurrency` the new number of desired threads

eventer_jobq_set_floor

> Set a jobq's minimum active thread count.

```
void
eventer_jobq_set_floor(eventer_jobq_t *jobq, uint32_t new_floor)
```

- `jobq` the jobq to modify
- `new_floor` the new number of minimum threads

eventer_jobq_set_lifo

> Instruct the jobq system to process jobs in LIFO vs. FIFO ordering.

```
void
eventer_jobq_set_lifo(eventer_jobq_t *jobq, mtev_boolean nv)
```

- `jobq` the jobq to modify
- `nv` Use LIFO or FIFO ordering if true or false, respectively.

eventer_jobq_set_max_backlog

> Set and advisory limit on the backlog a jobq will handle.

```
void
eventer_jobq_set_max_backlog(eventer_jobq_t *jobq, uint32_t max)
```

- `jobq` the jobq to modify
- `max` a maximum pending jobs count before eventer_try_add_asynch calls will fail.

eventer_jobq_set_min_max

> Set the upper and lower bounds on desired concurrency for a jobq.

```
void
eventer_jobq_set_min_max(eventer_jobq_t *jobq, uint32_t min, uint32_t max)
```

- `jobq` the jobq to modify
- `min` a minimum number of threads to maintain
- `max` a maximum number of threads to not exceed

eventer_jobq_set_shortname

> Set a "shorter" name for a jobq to be used in terse displays.

```
void
eventer_jobq_set_shortname(eventer_jobq_t *jobq, const char *name)
```

- `jobq` the jobq to modify
- `name` a shorter name for a job (clipped to 13 characters)

eventer_loop

> Start the event loop.

```
void
eventer_loop()
```

- **RETURN** N/A (does not return)

This function should be called as that last thing in your `child_main` function. See `mtev_main` .

eventer_loop_concurrency

> Determine the concurrency of the default eventer loop.

```
int
eventer_loop_concurrency()
```

- **RETURN** number of threads used for the default eventer loop.

eventer_loop_return

> Start the event loop and return control to the caller.

```
void
eventer_loop_return()
```

This function should be called as that last thing in your `child_main` function. Be sure not to return or exit after calling this as it could terminate your program. See `mtev_main` .

eventer_name_callback

> Register a human/developer readable name for a eventer callback function.

```
int
eventer_name_callback(const char *name, eventer_func_t callback)
```

- `name` the human readable name. You should select clear/unique names for clarity in debugging.
- `callback` the function pointer of the eventer callback.
- **RETURN** 0 on success.

eventer_name_callback_ext

> Register a functional describer for a callback and it's event object.

```
int
eventer_name_callback_ext(const char *name, eventer_func_t callback, void (*fn)(char *buff, int bufflen,
                          eventer_t e, void *closure), void *closure)
```

- `name` the human readable name. You should select clear/unique names for clarity in debugging.
- `callback` the function pointer of the eventer callback.
- `fn` function to call when describing the event. It should write a null terminated string into buff (no more than bufflen). If this is defined, it will override the "name" paramter when eventer_name_for_callback is called.
- **RETURN** 0 on success.

This function allows more in-depth descriptions of events. When an event is displayed (over the console or REST endpoints), this function is called with the event in question and the closure specified at registration time.

eventer_name_for_callback

> Retrieve a human readable name for the provided callback with event context.

```
const char *
eventer_name_for_callback(evneter_func_t f, eventer_t e)
```

- `f` a callback function.
- `e` and event object
- **RETURN** name of callback

The returned value may be a pointer to reusable thread-local storage. The value should be used before a subsequent call to this function. Aside from that caveat, it is thread-safe.

eventer_pool

> Find an eventer pool by name.

```
eventer_pool_t *
eventer_pool(const char *name)
```

- `name` the name of an eventer pool.
- **RETURN** an `eventer_pool_t *` by the given name, or NULL.

eventer_pool_concurrency

> Retrieve the concurrency of an eventer pool.

```
uint32_t
eventer_pool_concurrency(eventer_pool_t *pool)
```

- `pool` an eventer pool.
- **RETURN** the number of threads powering the specified pool.

eventer_pool_name

> Retrieve the name of an eventer pool.

```
const char *
eventer_pool_name(eventer_pool_t *pool)
```

- `pool` an eventer pool.
- **RETURN** the name of the eventer pool.

eventer_pool_watchdog_timeout

> Set a custom watchdog timeout for threads in an eventer pool.

```
void
eventer_pool_watchdog_timeout(eventer_pool_t *pool, double timeout)
```

- `pool` an eventer pool
- `timeout` the deadman timer in seconds.

eventer_read

> Execute an opset-appropriate `read` call.

```
int
eventer_read(eventer_t e, void *buff, size_t len, int *mask)
```

- `e` an event object
- `buff` a buffer in which to place read data.
- `len` the size of `buff` in bytes.
- `mask` a point the a mask. If the call does not complete, `*mask` it set.
- **RETURN** the number of bytes read or -1 with errno set.

If the function returns -1 and `errno` is `EAGAIN` , the `*mask` reflects the necessary activity to make progress.

eventer_ref

> Add a reference to an event.

```
void
eventer_ref(eventer_t e)
```

- `e` the event to reference.

Adding a reference to an event will prevent it from being deallocated prematurely. This is classic reference counting. It is are that one needs to maintain an actual event past the point where the eventer system would normally free it. Typically, one will allocate a new event and copy the contents of the old event into it allowing the original to be freed.

eventer_register_context

> Register an eventer carry context.

```
int
eventer_register_context(const char *name, eventer_context_opset_t *opset)
```

- `name` a string naming the context class.
- `opset` an opset for context maintenance
- **RETURN** A `ctx_idx` , -1 on failure.

eventer_remove

> Remove an event object from the eventer system.

```
eventer_t
eventer_remove(eventer_t e)
```

- `e` an event object to add.
- **RETURN** the event object removed if found; NULL if not found.

eventer_remove_fd

> Remove an event object from the eventer system by file descriptor.

```
eventer_t
eventer_remove_fd(int e)
```

- `fd` a file descriptor
- **RETURN** the event object removed if found; NULL if not found.

eventer_remove_fde

> Removes an fd event from the eventloop based on filedescriptor alone.

```
eventer_t
eventer_remove_fde(eventer_t e)
```

- `e` an event object
- **RETURN** The event removed, NULL if no event was present.

eventer_remove_recurrent

> Remove a recurrent event from the eventer.

```
eventer_t
eventer_remove_recurrent(eventer_t e)
```

- `e` an event object.
- **RETURN** The event removed (`== e`); NULL if not found.

eventer_remove_timed

> Remove a timed event from the eventer.

```
eventer_t
eventer_remove_timed(eventer_t e)
```

- `e` an event object (mask must be `EVENTER_TIMED`).
- **RETURN** the event removed, NULL if not found.

eventer_run_callback

> Run a callback as the eventer would

```
int
eventer_run_callback(eventer_funt_t f, eventer_t e, int mask, void *closure, struct timeval *now
                     uint64_t *dur)
```

- `f` The callback function (should always be eventer_get_callback(e))
- `e` The event to execute
- `mask` The mask observed.
- `closure` The closure (should always be eventer_get_closure(e))
- `now` The current time.
- `dur` An option nanosecond timing to populate.
- **RETURN** The mask that is desired from the callback.

eventer_run_in_thread

> Spawns a thread and runs the event until it returns 0.

```
void
eventer_run_in_thread(eventer_t e, int mask)
```

- `e` an event object
- `a` starting mask for triggering the event.

This function will remove the event from the eventer and set the socket into blocking mode.

eventer_set_callback

> Set an event's callback function.

```
void
eventer_set_callback(eventer_t e, eventer_func_t func)
```

- `e` an event object

eventer_set_closure

> Set an event's closure.

```
void
eventer_set_closure(eventer_t e, void *closure)
```

- `e` an event object
- `closure` a pointer to user-data to be supplied during callback.

eventer_set_context

> Set a context for an event.

```
void *
eventer_set_context(eventer_t e, int ctx_idx, void *data)
```

- `e` an event object
- `ctx_idx` is an idx returned from `eventer_register_context`
- `data` is new context data.
- **RETURN** The previously attached context.

eventer_set_eventer_aco

> Convert an eventer_t into an eventer_aco_t.

```
eventer_aco_t
eventer_set_eventer_aco(eventer_t e)
```

- `e` an event object
- **RETURN** The converted event.

This calls `eventer_set_eventer_aco_co` with the current aco as the `co` argument.

eventer_set_eventer_aco_co

> Convert an eventer_t into an eventer_aco_t.

```
eventer_aco_t
eventer_set_eventer_aco_co(eventer_t e, aco_t *co)
```

- `e` an event object
- `co` a coroutine to which the event should bound. NULL to revert.
- **RETURN** The converted event.

The input event is modified in-place. If the NULL is passed as co, then the event is reverted and NULL is returned. You almost always want to be calling this on a brand-new object or a `eventer_alloc_copy` of a pre-existing object.

eventer_set_fd_blocking

> Set a file descriptor into blocking mode.

```
int
eventer_set_fd_blocking(int fd)
```

- `fd` a file descriptor
- **RETURN** 0 on success, -1 on error (errno set).

eventer_set_fd_nonblocking

> Set a file descriptor into non-blocking mode.

```
int
eventer_set_fd_nonblocking(int fd)
```

- `fd` a file descriptor
- **RETURN** 0 on success, -1 on error (errno set).

eventer_set_mask

> Change an event's interests or intentions.

```
void
eventer_set_mask(eventer_t e, int mask)
```

- `e` an event object
- `mask` a new mask

Do not change change a mask from one event "type" to another. fd events must remain fd events. Timer must remain timer. Recurrent must remain recurrent. Do not alter asynch events at all. This simply changes the mask of the event without changing any eventer state and should be used with extremem care. Consider using the callback's return value or `eventer_update` to change the mask of an active event in the system.

eventer_set_owner

> Set the thread that owns an event.

```
void
eventer_set_owner(eventer_t e, pthread_t t)
```

- `e` an event object
- `t` a `pthread_t` thread; must be a valid event-loop.

eventer_thread_check

> Determine if the calling thread "owns" an event.

```
int
eventer_thread_check(eventer_t e)
```

- `e` an event object
- **RETURN** 0 if `e->thr_owner` is the `pthread_self()`, non-zero otherwise.

eventer_trigger

> Trigger an unregistered eventer and incorporate the outcome into the eventer.

```
void
eventer_trigger(eventer_t e, int mask)
```

- `e` an event object that is not registered with the eventer.
- `mask` the mask to be used when invoking the event's callback.

This is often used to "start back up" an event that has been removed from the eventer for any reason.

eventer_try_add_asynch

> Add an asynchronous event to a specific job queue.

```
mtev_boolean
eventer_try_add_asynch(eventer_jobq_t *q, eventer_t e)
```

- `q` a job queue
- `e` an event object
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q`. `e` must have a mask of `EVENTER_ASYNCH`.

eventer_try_add_asynch_dep

> Add an asynchronous event to a specific job queue dependent on the current job.

```
mtev_boolean
eventer_try_add_asynch_dep(eventer_jobq_t *q, eventer_t e)
```

- `q` a job queue
- `e` an event object
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q`. `e` must have a mask of `EVENTER_ASYNCH`. This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_try_add_asynch_dep_subqueue

> Add an asynchronous event to a specific job queue dependent on the current job.

```
mtev_boolean
eventer_try_add_asynch_dep_subqueue(eventer_jobq_t *q, eventer_t e, uint64_t id)
```

- `q` a job queue
- `e` an event object
- `id` is a fairly competing subqueue identifier
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q` . `e` must have a mask of `EVENTER_ASYNCH` . This should be called from within a asynch callback during a mask of `EVENTER_ASYNCH_WORK` and the new job will be a child of the currently executing job.

eventer_try_add_asynch_subqueue

> Add an asynchronous event to a specific job queue.

```
mtev_boolean
eventer_try_add_asynch_subqueue(eventer_jobq_t *q, eventer_t e, uint64_t id)
```

- `q` a job queue
- `e` an event object
- `id` is a fairly competing subqueue identifier
- **RETURN** `mtev_false` if over max backlog, caller must clean event.

This adds the `e` event to the job queue `q` . `e` must have a mask of `EVENTER_ASYNCH` .

eventer_update

> Change the activity mask for file descriptor events.

```
void
eventer_update(evneter_t e, int mask)
```

- `e` an event object
- `mask` a new mask that is some bitwise or of `EVENTER_READ` , `EVENTER_WRITE` , and `EVENTER_EXCEPTION`

eventer_update_whence

> Change the time at which a registered timer event should fire.

```
void void
eventer_update_whence(eventer_t e, struct timeval whence)
```

- `e` an event object
- `whence` an absolute time.

eventer_wakeup

> Signal up an event loop manually.

```
void
eventer_wakeup(eventer_t e)
```

- `e` an event

The event `e` is used to determine which thread of the eventer loop to wake up. If `e` is `NULL` the first thread in the default eventer loop is signalled. The eventer loop can wake up on timed events, asynchronous job completions and file descriptor activity. If, for an external reason, one needs to wake up a looping thread, this call is used.

eventer_watchdog_timeout

> Return the current watchdog timeout on this thread.

```
double
eventer_watchdog_timeout(void)
```

- **RETURN** A timeout in seconds, 0.0 if none configured.

eventer_watchdog_timeout_timeval

> Return the current watchdog timeout on this thread.

```
mtev_boolean
eventer_watchdog_timeout_timeval(struct timeval *dur)
```

- `dur` the timeval structure to populate with the timeout.
- **RETURN** mtev_true if a timeout is set, mtev_false otherwise.

eventer_write

> Execute an opset-appropriate `write` call.

```
int
eventer_write(eventer_t e, const void *buff, size_t len, int *mask)
```

- `e` an event object
- `buff` a buffer containing data to write.
- `len` the size of `buff` in bytes.
- `mask` a point the a mask. If the call does not complete, `*mask` it set.
- **RETURN** the number of bytes written or -1 with errno set.

If the function returns -1 and `errno` is `EAGAIN` , the `*mask` reflects the necessary activity to make progress.

F

mtev_flow_regulator_ack

> Acknowledge processing mtev_flow_regulator_toggle_t instruction.

```
mtev_flow_regulator_toggle_t
mtev_flow_regulator_ack(mtev_flow_regulator_t *fr, mtev_flow_regulator_toggle_t t)
```

- `t` Instruction returned from previous call to `mtev_flow_regulator_raise_one` , `mtev_flow_regulator_lower` , or `mtev_flow_regulator_ack` .
- **RETURN** New flow-toggle instruction.

The flow-regulator is designed to be usable in multi-producer (where multiple concurrent entities may produce work) / multi-consumer (where multiple concurrent entities may mark work completed) scenarios, which means that many entities may be adding and removing work from the flow-regulator at the same time. As such, when one entity observes that the flow-regulator has become disabled and takes some action to pause further work generation, it's possible that enough work will have drained from the flow-regulator that it needs to be re-enabled... Or, that after re-enabling the flow-regulator, enough work was already being scheduled that it needs to be disabled again. So `mtev_flow_regulator_ack` returns a *new* instruction, in case the flow-regulator needs further adjustment. Clients are expected to call this function in a loop, with the function's previous return value, until the flow-regulator settles on `MTEV_FLOW_REGULATOR_TOGGLE_KEEP` or `MTEV_FLOW_REGULATOR_TOGGLE_DISABLED`. (There is no harm in continuing to call `mtev_flow_regulator_ack` after it reaches one of these values: it will eventually settle on `MTEV_FLOW_REGULATOR_TOGGLE_KEEP`.)

The toggle-instruction should be interpreted as follows:

- `MTEV_FLOW_REGULATOR_TOGGLE_DISABLED`: Flow control is currently disabled. No client action necessary.
- `MTEV_FLOW_REGULATOR_TOGGLE_DISABLE`: Flow control *was* enabled, and we've started transitioning to DISABLED. (The transition to DISABLED is not complete until the client calls `mtev_flow_regulator_ack`, again.) Client MAY try to prevent generating new work before calling `mtev_flow_regulator_ack`, again.
- `MTEV_FLOW_REGULATOR_TOGGLE_KEEP`: No client action required.
- `MTEV_FLOW_REGULATOR_TOGGLE_ENABLE`: Flow control *was* disabled, and has just started transitioning to ENABLED. (The transition to ENABLED is not complete until the client calls `mtev_flow_regulator_ack`, again.) Client MAY re-enable work-generation before calling `mtev_flow_regulator_ack`, again.

To facilitate multi-producer / multi-consumer use, the flow-regulator enforces that only *one* client will see a flow-toggling result (*i.e.* `MTEV_FLOW_REGULATOR_TOGGLE_ENABLE` or `MTEV_FLOW_REGULATOR_TOGGLE_DISABLE`) until that client calls `mtev_flow_regulator_ack`, and that the same toggling result will not occur twice in a row across all concurrent clients.

mtev_flow_regulator_create

> Create a flow-regulator object.

```
mtev_flow_regulator_t *
mtev_flow_regulator_create(unsigned int low, unsigned int high)
```

- `low` Threshold that indicates when work flow should be re-enabled.
- `high` Threshold at which to stop work flow. Must be strictly greater than `low`.
- **RETURN** Flow-regulator object.

The returned flow-regulator object is "enabled" on creation. When `high` work items are added (by `mtev_flow_regulator_raise_one`) without being removed (by `mtev_flow_regulator_lower`), the flow-regulator will become disabled. When `high - low` work-items are subsequently marked done (by `mtev_flow_regulator_lower`), without new work being added (by `mtev_flow_regulator_raise_one`), the flow-regulator will transition back to "enabled".

mtev_flow_regulator_destroy

> Destroy a flow-regulator object.

```
void
mtev_flow_regulator_destroy(mtev_flow_regulator_t *fr)
```

mtev_flow_regulator_lower

> Release space for work-items in a flow-regulator.

```
mtev_flow_regulator_toggle_t
mtev_flow_regulator_lower(mtev_flow_regulator_t *fr, unsigned int by)
```

- `by` Number of work-items to mark completed.
- **RETURN** Action to take on releasing work.

See `mtev_flow_regulator_ack` for description of how to handle the return value. This function will return one of:

- `MTEV_FLOW_REGULATOR_TOGGLE_KEEP`
- `MTEV_FLOW_REGULATOR_TOGGLE_ENABLE`

mtev_flow_regulator_raise_one

> Reserve space for a work-item in a flow-regulator.

```
mtev_flow_regulator_toggle_t
mtev_flow_regulator_raise_one(mtev_flow_regulator_t *fr)
```

- **RETURN** Success / fail status on inserting work.

See `mtev_flow_regulator_ack` for description of how to handle the return value. This function will return one of:

- `MTEV_FLOW_REGULATOR_TOGGLE_DISABLED`
- `MTEV_FLOW_REGULATOR_TOGGLE_DISABLE`
- `MTEV_FLOW_REGULATOR_TOGGLE_KEEP`

Note that, unless the return value was `MTEV_FLOW_REGULATOR_TOGGLE_KEEP` , space was *not* reserved in the flow-regulator for the work-item.

mtev_flow_regulator_stable_lower

```
mtev_flow_regulator_toggle_t
mtev_flow_regulator_stable_lower(mtev_flow_regulator_t *fr, unsigned int by)
```

- **RETURN** `mtev_true` if the caller is responsible for re-scheduling work-creation for the flow-regulator, `mtev_false` otherwise.

This function is a simple wrapper around `mtev_flow_regulator_lower` and `mtev_flow_regulator_ack` , to simplify handling in cases where the client needs take no explicit action to enable or disable work-production before calling `mtev_flow_regulator_ack` .

mtev_flow_regulator_stable_try_raise_one

```
mtev_flow_regulator_toggle_t
mtev_flow_regulator_stable_try_raise_one(mtev_flow_regulator_t *fr)
```

- **RETURN** mtev_true if work-item was successfully added to the flow regulator, mtev_false otherwise.

This function is a simple wrapper around `mtev_flow_regulator_raise_one` and `mtev_flow_regulator_ack` , to simplify handling in cases where the client needs take no explicit action to enable or disable work-production before calling `mtev_flow_regulator_ack` .

mtev_frrh_adjust_prob

> Change the replacement probability on a `mtev_frrh_t`.

```
void
mtev_frrh_adjust_prob(mtev_frrh_t *cache, uint32_t prob)
```

- `cache` the `mtev_frrh_t` on which to change the probability.
- `prob` is the probability of replaement on collision (0 to UINT_MAX).

mtev_frrh_alloc

> Allocate a fast random replacement hash.

```
mtev_frrh_t *
mtev_frrh_alloc(uint64_t size, size_t datasize, uint32_t prob, mtev_frrh_hash hashf,
                mtev_frrh_alloc_entry allocf, mtev_frrh_free_entry freef)
```

- `size` is the total capacity of the hash.
- `datasize` is the fixed-size of the data which will be stored.
- `prob` is the probability of replaement on collision (0 to UINT_MAX).
- `hashf` is the hashing function, NULL uses XXH64.
- `allocf` is the allocation function to use, NULL uses malloc.
- `freef` is the free function to use, NULL uses free.
- **RETURN** a pointer to a `mtev_frrh_t` on success, NULL otherwise.

mtev_frrh_get

> Retrieves the data associated with the provided key from the cache.

```
const void *
mtev_frrh_get(mtev_frrh_t *cache, const void *key, uint32_t keylen)
```

- `cache` a `mtev_frrh_t`.
- `key` a pointer to the key.
- `keylen` the length of the key in bytes.
- **RETURN** a pointer to a copy of the data store with the key.

mtev_frrh_set

> Possibly set a key-value pair in a `mtev_frrh_t`

```
mtev_boolean
mtev_frrh_set(mtev_frrh_t *cache, const void *key, uint32_t keylen, const void *data)
```

- `cache` a `mtev_frrh_t`.
- `key` a pointer to the key.
- `keylen` the length of the key in bytes.
- `data` a pointer to the data (must be of the specified datasize for the `mtev_frrh_t`.
- **RETURN** `mtev_true` if added, `mtev_false` if not.

mtev_frrh_stats

> Retrieve access and hit statatistics.

```
void
mtev_frrh_stats(mtev_frrh_t *cache, uint64_t *accesses, uint64_t *hits)
```

- `cache` the `mtev_frrh_t` in question.
- `accesses` is an optional out pointer to store the number of accesses.
- `hits` is an optional out pointer to store the number of hits.

G

mtev_get_durations_ms

> Return suffixes for millisecond-resolution durations.

```
const mtev_duration_definition_t *
mtev_get_durations_ms(void)
```

Return value is suitable to pass as the second argument to mtev_confstr_parse_duration. Millisecond-scale duration suffixes are:

- `ms` (for milliseconds);
- `s` and `sec` (for seconds);
- `min` (for minutes);
- `hr` (for hours).
- `d` (for days).
- `w` (for weeks).

mtev_get_durations_ns

> Return suffixes for nanosecond-resolution durations.

```
const mtev_duration_definition_t *
mtev_get_durations_ns(void)
```

Return value is suitable to pass as the second argument to mtev_confstr_parse_duration. Nanosecond-scale duration suffixes are:

- `ns` (for nanoseconds);
- `us` (for microseconds);
- `ms` (for milliseconds);
- `s` and `sec` (for seconds);
- `min` (for minutes);
- `hr` (for hours).

mtev_get_durations_s

> Return suffixes for second-resolution durations.

```
const mtev_duration_definition_t *
mtev_get_durations_s(void)
```

Return value is suitable to pass as the second argument to mtev_confstr_parse_duration. Second-scale duration suffixes are:

- `s` and `sec` (for seconds);
- `min` (for minutes);
- `hr` (for hours).

- `d` (for days).
- `w` (for weeks).

mtev_get_durations_us

> Return suffixes for microsecond-resolution durations.

```
const mtev_duration_definition_t *
mtev_get_durations_us(void)
```

Return value is suitable to pass as the second argument to mtev_confstr_parse_duration. Microsecond-scale duration suffixes are:

- `us` (for microseconds);
- `ms` (for milliseconds);
- `s` and `sec` (for seconds);
- `min` (for minutes);
- `hr` (for hours).
- `d` (for days).

mtev_get_nanos

> Like mtev_gethrtime. It actually is the implementation of mtev_gethrtime()

```
uint64_t
mtev_get_nanos(void)
```

- **RETURN** number of nanos seconds from an arbitrary time in the past.

mtev_getip_ipv4

> find the local IPv4 address that would be used to talk to remote

```
int
mtev_getip_ipv4(struct in_addr remote, struct in_addr *local)
```

- `remote` the destination (no packets are sent)
- `local` the pointer to the local address to be set
- **RETURN** 0 on success, -1 on failure

mtev_gettimeofday

> Maybe fast-pathed version of gettimeofday

```
int
mtev_gettimeofday(struct timeval *t, void **ttp)
```

- **RETURN** same as system gettimeofday();

  If the fast path is taken, ttp is ignored.

H

mtev_hash__hash

```
uint32_t
mtev_hash__hash(const void *k, uint32_t length, uint32_t initval)
```

> the internal hash function that mtev_hash_table uses exposed for external usage

mtev_hash_adv

```
int
mtev_hash_adv(mtev_hash_table *h, mtev_hash_iter *iter)
```

> iterate through key/values in the hash_table

This is an iterator and requires the hash to not be written to during the iteration process. To use: mtev_hash_iter iter = MTEV_HASH_ITER_ZERO;

while(mtev_hash_adv(h, &iter)) { .... use iter.key.{str,ptr}, iter.klen and iter.value.{str,ptr} .... }

mtev_hash_adv_spmc

```
int
mtev_hash_adv_spmc(mtev_hash_table *h, mtev_hash_iter *iter)
```

> iterate through the key/values in the hash_table

This is an iterator and requires that if the hash it written to during the iteration process, you must employ SMR on the hash itself to prevent destruction of memory for hash resizes by using the special init function mtev_hash_init_mtev_memory.

To use: mtev_hash_iter iter = MTEV_HASH_ITER_ZERO;

while(mtev_hash_adv_spmc(h, &iter)) { .... use iter.key.{str,ptr}, iter.klen and iter.value.{str,ptr} .... }

mtev_hash_delete

```
int
mtev_hash_delete(mtev_hash_table *h, const void *k, int klen, NoitHashFreeFunc keyfree
                 NoitHashFreeFunc datafree)
```

> remove the key/value stored at "k" and call keyfree and datafree if they are provided

mtev_hash_delete_all

```
void
mtev_hash_delete_all(mtev_hash_table *h, NoitHashFreeFunc keyfree, NoitHashFreeFunc datafree)
```

> remove all keys and values and call keyfree and datafree if they are provided

mtev_hash_destroy

```
void
mtev_hash_destroy(mtev_hash_table *h, NoitHashFreeFunc keyfree, NoitHashFreeFunc datafree)
```

> remove all keys and values and call keyfree and datafree if they are provided but also wipe out the underlying map

This must be called on any hash_table that has been mtev_hash_inited or it will leak memory

mtev_hash_get

```
void *
mtev_hash_get(mtev_hash_table *h, const void *k, int klen)
```

> return the value at "k

mtev_hash_init

```
void
mtev_hash_init(mtev_hash_table *h)
```

> initialize a hash_table

will default to LOCK_MODE_NONE and MTEV_HASH_DEFAULT_SIZE (1<<7)

mtev_hash_init_locks

```
void
mtev_hash_init_locks(mtev_hash_table *h, int size, mtev_hash_lock_mode_t lock_mode)
```

> choose the lock mode when initing the hash.

It's worth noting that the lock only affects the write side of the hash, the read side remains completely lock free.

mtev_hash_init_mtev_memory

```
void
mtev_hash_init_mtev_memory(mtev_hash_table *h, int size, mtev_hash_lock_mode_t lock_mode)
```

> choose the lock mode when initing the hash.

It's worth noting that the lock only affects the write side of the hash, the read side remains completely lock free.

This variant will use mtev_memory ck allocator functions to allow this hash to participate in SMR via mtev_memory transactions. You need to wrap memory transactions in mtev_memory_begin()/mtev_memory_end()

mtev_hash_init_size

```
void
mtev_hash_init_size(mtev_hash_table *h, int size)
```

> initialize a hash_table with an initial size

will default to LOCK_MODE_NONE

mtev_hash_merge_as_dict

```
void
mtev_hash_merge_as_dict(mtev_hash_table *dst, mtev_hash_table *src)
```

> merge string values in "src" into "dst"

This is a convenience function only. It assumes that all keys and values in the destination hash are strings and allocated with malloc() and assumes that the source contains only keys and values that can be suitably duplicated by strdup().

mtev_hash_next

```
int
mtev_hash_next(mtev_hash_table *h, mtev_hash_iter *iter, const char **k, int *klen
               void **data)
```

> iterate through the key/values in the hash_table

These are older, more painful APIs... use mtev_hash_adv Note that neither of these sets the key, value, or klen in iter

mtev_hash_next_str

```
int
mtev_hash_next_str(mtev_hash_table *h, mtev_hash_iter *iter, const char **k, int *klen
                   const char **dstr)
```

> iterate through the key/values in the hash_table as strings

These are older, more painful APIs... use mtev_hash_adv

mtev_hash_replace

```
int
mtev_hash_replace(mtev_hash_table *h, const void *k, int klen, const void *data,
                  NoitHashFreeFunc keyfree, NoitHashFreeFunc datafree)
```

> replace and delete (call keyfree and datafree functions) anything that was already in this hash location

mtev_hash_retr_str

```
int
mtev_hash_retr_str(mtev_hash_table *h, const void *k, int klen, const char **dstr)
```

> fetch the value at "k" into "data" as a string

mtev_hash_retrieve

```
int
mtev_hash_retrieve(mtev_hash_table *h, const void *k, int klen, void **data)
```

> fetch the value at "k" into "data"

mtev_hash_set

```
int
mtev_hash_set(mtev_hash_table *h, const void *k, int klen, const void *data, char **oldkey
              void **olddata)
```

> replace and return the old value and old key that was in this hash location

will return MTEV_HASH_SUCCESS on successful set with no replacement will return MTEV_HASH_FAILURE on failure to set will return MTEV_HASH_SUCCESS_REPLACEMENT on successful set with replacement

mtev_hash_size

```
int
mtev_hash_size(mtev_hash_table *h)
```

> return the number of entries in the hash_table

mtev_hash_store

```
int
mtev_hash_store(mtev_hash_table *h, const void *k, int klen, const void *data)
```

> put something in the hash_table

This will fail if the key already exists in the hash_table

NOTE! "k" and "data" MUST NOT be transient buffers, as the hash table implementation does not duplicate them. You provide a pair of NoitHashFreeFunc functions to free up their storage when you call mtev_hash_delete(), mtev_hash_delete_all() or mtev_hash_destroy().

mtev_html_encode

> Encode raw data as html encoded output into the provided buffer.

```
int
mtev_html_encode(const char *src, size_t src_len, char *dest, size_t dest_len)
```

- `src` The buffer containing the raw data.
- `src_len` The size (in bytes) of the raw data.
- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the encoded output. Returns zero is out_sz is too small.

mtev_html_encode_len

> Calculate how large a buffer must be to contain the url encoding for a given number of bytes.

```
size_t
mtev_html_encode_len(size_t src_len)
```

- `src_len` The size (in bytes) of the raw data buffer that might be encoded.
- **RETURN** The size of the buffer that would be needed to store an encoded version of an input string.

mtev_huge_hash_adv

```
int
mtev_huge_hash_adv(mtev_huge_hash_iter_t *iter)
```

> iterate through key/values in the hash_table

To use: mtev_huge_hash_iter_t *iter = mtev_huge_hash_create_iter(hh);

while(mtev_huge_hash_adv(iter)) { size_t key_len, data_len; void *k = mtev_huge_hash_iter_key(iter, &key_len); void* d = mtev_huge_hash_iter_value(iter, &data_len); }

mtev_huge_hash_create

```
mtev_huge_hash_t *
mtev_huge_hash_create(const char *path)
```

> create or open a huge_hash

Failure to open or create will return NULL and errno will be set appropriately. See: mtev_huge_hash_strerror()

\fn mtev_huge_hash_iter_t *mtev_huge_hash_create_iter(mtev_huge_hash_t* hh);

> create an iterator for walking the huge_hash

Note that the existence of an interator can prevent calls to mtev_huge_hash_store from completing if the underlying data has to resize. Iterate with caution.

mtev_huge_hash_delete

```
mtev_boolean
mtev_huge_hash_delete(mtev_huge_hash_t *hh, const void *k, size_t klen)
```

> remove the key/value stored at "k"

mtev_huge_hash_replace

```
int
mtev_huge_hash_replace(mtev_huge_hash_t *hh, const void *k, size_t klen, const void *data
                       size_t dlen)
```

> replace anything that was already in this hash location

mtev_huge_hash_retrieve

```
const void *
mtev_huge_hash_retrieve(mtev_huge_hash_t *hh, const void *k, size_t klen, size_t *data_len)
```

> return the value at "k" and fill data_len with sizeof the data

The memory returned here is owned by the huge_hash. Do not modify

mtev_huge_hash_size

```
size_t
mtev_huge_hash_size(mtev_huge_hash_t *hh)
```

> return the number of entries in the huge_hash

mtev_huge_hash_store

```
int
mtev_huge_hash_store(mtev_huge_hash_t *hh, const void *k, size_t klen, const void *data
                     size_t dlen)
```

> put something in the huge_hash

This will fail if the key already exists in the hash_table Copies are made of `k` and `data`

Returns mtev_true on success

I

mtev_intern

> Like `mtev_intern_pool` invoked with `MTEV_INTERN_DEFAULT_POOL`.

```
mtev_intern_t
mtev_intern(const void *buff, size_t len)
```

- `buff` The data to be interned.
- `len` The length of data to be considered (0, 2^23)
- **RETURN** A new, or pre-existing intern from the default pool.

mtev_intern_copy

> Return a reference to an existing `mtev_intern_t`.

```
mtev_intern_t
mtev_intern_copy(const mtev_intern_t iv)
```

- `iv` An existing, valid `mtev_intern_t`
- **RETURN** A reference to the interned data.

The copy must be released just as if you created it via `mtev_intern_pool`.

mtev_intern_get_cstr

> Retrieve the string from an `mtev_intern_t` type.

```
const char *
mtev_intern_get_cstr(const mtev_intern_t iv, size_t *len)
```

- `iv` The interned data.
- `len` An out value for the length of the string. Unused if NULL.
- **RETURN** The string contained in the interned value.

The return value is only valid until `mtev_intern_release*` is called.

mtev_intern_get_ptr

> Retrieve the data from an `mtev_intern_t` type.

```
const void *
mtev_intern_get_ptr(const mtev_intern_t iv, size_t *len)
```

- `iv` The interned data.
- `len` An out value for the length of the string. Unused if NULL.
- **RETURN** The memory contained in the interned value.

The return value is only valid until `mtev_intern_release*` is called.

mtev_intern_get_refcnt

> Retrieve the current refcnt for an intern item.

```
uint32_t
mtev_intern_get_refcnt(mtev_intern_t iv)
```

- `iv` The interned value.
- **RETURN** The number of references currently outstanding.

mtev_intern_pool

> Request an interned data item with specific contents.

```
mtev_intern_t
mtev_intern_pool(mtev_intern_pool_t *pool, const void *buff, size_t len)
```

- `pool` The pool in which to intern the data.
- `buff` The data to be interned.
- `len` The length of data to be considered (0, 2^23)
- **RETURN** A new, or pre-existing intern from the pool.

This function will attempt to find the specified data in the pool, but create it on absence. The reference count of the interned object returned will be increased and it must be released using `mtev_intern_release_pool` .

mtev_intern_pool_by_id

> Return an existing pool by id.

```
mtev_intern_pool_t *
mtev_intern_pool_by_id(uint8_t id)
```

- `id` the pool id.
- **RETURN** A new intern pool.

mtev_intern_pool_compact

> Attempt a compaction of an intern pool.

```
int
mtev_intern_pool_compact(mtev_intern_pool_t *pool, mtev_boolean force)
```

- `pool` The pool to compact.
- `force` A boolean dictating if compaction should be forced.

- **RETURN** The number of free fragment merges that occurred, -1 if it will be performed asynch.

This function will walk all the free fragment lists within the pool joining adjacent ones and promoting them into the the right slabs. If force is false, compaction will be avoided if there are less than approximately 1.5x fragments as there were after the previous successful compaction. If this is called from an eventer thread (not a jobq), it will be scheduled to be done in the default jobq (work deferred) and will return -1 instead of a measurement of compaction. If force is specified, it will be done synchronously regardless of callsite location.

mtev_intern_pool_item_count

> Return the number of unique interned items in a pool.

```
uint32_t
mtev_intern_pool_item_count(mtev_intern_pool_t *pool)
```

- `pool` The pool to analyze.
- **RETURN** The number of unique interned items.

mtev_intern_pool_new

> Create a new intern pool.

```
mtev_intern_pool_t *
mtev_intern_pool_new(mtev_intern_pool_attr_t *attr)
```

- `attr` the attributes describing the pool.
- **RETURN** A new intern pool.

mtev_intern_pool_stats

> Return statistics for an intern pool.

```
void
mtev_intern_pool_stats(mtev_intern_pool_t *pool, mtev_intern_pool_stats_t *stats)
```

- `pool` The pool to inspect.
- `stats` The statistics structure to fill out.

mtev_intern_pool_str

> Request an interned string item with specific contents.

```
mtev_intern_t
mtev_intern_pool_str(mtev_intern_pool_t *pool, const char *buff, size_t len)
```

- `pool` The pool in which to intern the string.
- `buff` The string to be interned.
- `len` The length of `buff` . `len` must be less than 2^23-1. If 0, strlen will be invoked.
- **RETURN** A new, or pre-existing intern from the pool.

This function will attempt to find the specified string in the pool, but create it on absence. The reference count of the interned string returned will be increased and it must be released using `mtev_intern_release_pool` .

mtev_intern_release

> Release interned data back to the pool from which it was allocated.

```
void
mtev_intern_release(mtev_intern_t iv)
```

- `iv` The interned value to release.

mtev_intern_release_pool

> Release interned data back to a pool.

```
void
mtev_intern_release_pool(mtev_intern_pool_t *pool, mtev_intern_t iv)
```

- `pool` The pool to release `iv` to.
- `iv` The interned value to release.

Interned values must be released to the pool they were retrieved from. Attempting to release to a different pool will cause a crash.

mtev_intern_str

> Like `mtev_intern_pool` invoked with `MTEV_INTERN_DEFAULT_POOL` .

```
mtev_intern_t
mtev_intern_str(const char *buff, size_t len)
```

- `buff` The string to be interned.
- `len` The length of string. `len` must be less than 2^23-1. If 0, strlen will be invoked.
- **RETURN** A new, or pre-existing intern from the default pool.

L

mtev_lfu_create

```
mtev_lfu_create(int32_t max_entries, void (*free_fn)(void *))
```

> Create an LFU of max_entries size

Will call free_fn when an item is evicted. if free_fn is null, call free(). If max_entries == -1 then this devolves to a normal hashtable.

mtev_lfu_destroy

```
mtev_lfu_destroy(mtev_lfu_t *)
```

> Destroy the LFU

mtev_lfu_get

```
mtev_lfu_entry_token
```

```
mtev_lfu_get(mtev_lfu_t *lfu, const char *key, size_t key_len, void **value)
```

> Get an item from the LFU by key

This will fetch the item at "key" and put the value in "value". It will also return a token as the return value of the function. This token is used as the checkout of the item from the LFU. When you are finished using the value, you must call "mtev_lfu_release(mtev_lfu_t *lfu, mtev_lfu_entry_token token)" to let the LFU know that reclamation for that key/value is possible.

mtev_lfu_invalidate

```
mtev_lfu_invalidate(mtev_lfu_t *)
```

> Remove all entries from the LFU

mtev_lfu_iterate

```
void
mtev_lfu_iterate(mtev_lfu_t *lfu, void (*callback)(mtev_lfu_t *lfu, const char *key,
                 size_t key_len, void *value, void *closure), void *closure)
```

> Iterate through all entries in the LFU

```
* `lfu` The LFU to iterate
* `callback` This function is called for each item in the LFU
* `closure` The pointer to pass as the last param to the callback
```

This leaves the LFU locked during iteration which will starve out other operations. Keep this in mind if you are storing a lot of stuff in the LFU and have multithreaded access to it.

The "key" and "value" passed to the callback function is direct LFU memory and should not be freed.

mtev_lfu_put

```
mtev_lfu_put(mtev_lfu_t *lfu, const char *key, size_t key_len, void *value)
```

> Put a new item into the LFU

If some other thread has added a val at this key this will overwrite it and restart the frequency count at 1.

This will cause an eviction of the least frequently used item if the cache is full.

mtev_lfu_release

```
void
mtev_lfu_release(mtev_lfu_t *lfu, mtev_lfu_entry_token token)
```

> Surrender an item back to the LFU

To be memory safe LFU tokens must be released back to the LFU when the user is finished using them.

mtev_lfu_remove

```
mtev_lfu_remove(mtev_lfu_t *lfu, const char *key, size_t key_len)
```

> Remove key from the LFU

This does not call the free_fn, instead it returns the value

mtev_lfu_size

```
mtev_lfu_size(mtev_lfu_t *lfu)
```

> Return the total entry count in the LFU

mtev_lockfile_acquire

> lock the file immediately if possible, return -1 otherwise.

```
mtev_lockfile_t
mtev_lockfile_acquire(const char *fp)
```

- `fp` the path to the lock file
- **RETURN** >= 0 on success, -1 on failure

mtev_lockfile_acquire_owner

> lock the file immediately if possible, return -1 otherwise.

```
mtev_lockfile_t
mtev_lockfile_acquire_owner(const char *fp, pid_t *owner)
```

- `fp` the path to the lock file
- `owner` is a pointer to a pid. If the lock is owned by another process, this will be set to that pid, otherwise it will be set to -1.
- **RETURN** >= 0 on success, -1 on failure

mtev_lockfile_release

> release a held file lock

```
int
mtev_lockfile_release(mtev_lockfile_t fd)
```

- `fd` the file lock to release
- **RETURN** -1 on failure, 0 on success

mtev_lua_lmc_alloc

> Allocated and initialize a `lua_module_closure_t` for a new runtime.

```
lua_module_closure_t *
mtev_lua_lmc_alloc(mtev_dso_generic_t *self, mtev_lua_resume_info_t *resume)
```

- `self` the module implementing a custom lua runtime environment
- `resume` the custom resume function for this environment

* **RETURN** a new allocated and initialized `lua_module_closure`

> Note these are not thread safe because lua is not thread safe. If you are managing multiple C threads, you should have a `lua_module_closure_t` for each thread and maintain them in a thread-local fashion. Also ensure that any use of the eventer does not migrate cross thread.

mtev_lua_lmc_free

> Free a `lua_module_closure_t` structure that has been allocated.

```
void
mtev_lua_lmc_free(lua_module_closure_t *lmc)
```

* `lmc` The `lua_module_closure_t` to be freed.

mtev_lua_lmc_L

> Get the `lua_State *` for this module closure.

```
lua_State *
mtev_lua_lmc_L(lua_module_closure_t *lmc)
```

* `lmc` the `lua_module_closure_t` that was allocated for this runtime.
* **RETURN** a Lua state

mtev_lua_lmc_resume

> Invoke lua_resume with the correct context based on the `lua_module_closure_t`

```
int
mtev_lua_lmc_resume(lua_module_closure_t *lmc, mtev_lua_resume_info_t *ri, int nargs)
```

* `lmc` the `lua_module_closure_t` associated with the current lua runtime.
* `ri` resume meta information
* `nargs` the number of arguments on the lua stack to return
* **RETURN** the return value of the underlying `lua_resume` call.

mtev_lua_lmc_setL

> Set the `lua_State *` for this module closure, returning the previous value.

```
lua_State *
mtev_lua_lmc_setL(lua_module_closure_t *lmc)
```

* `lmc` the `lua_module_closure_t` that was allocated for this runtime.
* `lmc` the `lua_State *` that should be placed in this closure.
* **RETURN** the previous lua Lua state associated with this closure

M

mtev_main

> Run a comprehensive mtev setup followed by a "main" routine.

```
int
mtev_main(const char *appname, const char *config_filename, int debug, int foreground,
          mtev_log_op_t lock, const char *glider, const char *drop_to_user,
          const char *drop_to_group, int (*passed_child_main)(void))
```

- `appname` The application name (should be the config root node name).
- `config_filename` The path the the config file.
- `debug` Enable debugging (logging).
- `foreground` 0 to daemonize with watchdog, 1 to foreground, 2 to foreground with watchdog.
- `lock` Specifies where to not lock, try lock or exit, or lock or wait.
- `glider` A path to an executable to invoke against the process id on crash. May be NULL.
- `drop_to_user` A target user for dropping privileges when under watchdog. May be NULL.
- `drop_to_group` A target group for dropping privileges when under watchdog. May be NULL.
- `passed_child_main` A programmers supplied main function.
- **RETURN** -1 on failure, 0 on success if `foreground==1`, or the return value of `main` if run in the foreground.

mtev_main_eventer_config

> Set options for an app that need not be specified explicitly in a config.

```
void
mtev_main_eventer_config(const char *name, const char *value)
```

- `name` The config key name
- `value` The value of the config option

mtev_main_status

> Determine if that application is already running under this configuration.

```
int
mtev_main_status(const char *appname, const char *config_filename, int debug, pid_t *pid
                 pid_t *pgid)
```

- `appname` The application name (should be the config root node name).
- `config_filename` The path the the config file.
- `debug` Enable debugging (logging).
- `pid` If not null, it is populated with the process id of the running instance.
- `pgid` If not null, it is populated with the process group id of the running instance.
- **RETURN** 0 on success, -1 on failure.

mtev_main_terminate

> Terminate an already running application under the same configuration.

```
int
mtev_main_terminate(const char *appname, const char *config_filename, int debug)
```

- `appname` The application name (should be the config root node name).
- `config_filename` The path the the config file.
- `debug` Enable debugging (logging).
- **RETURN** 0 on success, -1 on failure. If the application is not running at the time of invocation, termination is considered

successful.

MTEV_MAYBE_DECL

> C Macro for declaring a "maybe" buffer.

```
MTEV_MAYBE_DECL(type, name, cnt)
```

- `type` A C type (e.g. char)
- `name` The name of the C variable to declare.
- `cnt` The number of type elements initially declared.

A "maybe" buffer is a buffer that is allocated on-stack, but if more space is required can be reallocated off stack (malloc). One should always call `MTEV_MAYBE_FREE` on any allocated maybe buffer.

MTEV_MAYBE_DECL_VARS

> C Macro for declaring a "maybe" buffer.

```
MTEV_MAYBE_DECL_VARS(type, name, cnt)
```

- `type` A C type (e.g. char)
- `name` The name of the C variable to declare.
- `cnt` The number of type elements initially declared.

MTEV_MAYBE_FREE

> C Macro to free any heap space associated with a "maybe" buffer.

```
MTEV_MAYBE_FREE(name)
```

- `name` The name of the "maybe" buffer.

MTEV_MAYBE_INIT_VARS

> C Macro for initializing a "maybe" buffer

```
MTEV_MAYBE_INIT_VARS(name)
```

- `name` The name of "maybe" buffer.

MTEV_MAYBE_REALLOC

> C Macro to ensure a maybe buffer has at least cnt elements allocated.

```
MTEV_MAYBE_REALLOC(name, cnt)
```

- `name` The name of the "maybe" buffer.
- `cnt` The total number of elements expected in the allocation.

This macro will never reduce the size and is a noop if a size smaller than or equal to the current allocation size is specified. It is safe to simply run this macro prior to each write to the buffer.

MTEV_MAYBE_SIZE

> C Macro for number of bytes available in this buffer.

```
MTEV_MAYBE_SIZE(name)
```

- `name` The name of the "maybe" buffer.

mtev_merge_sort

> Merge sort data starting at head_ptr_ptr, iteratively

```
void
mtev_merge_sort(void **head_ptr_ptr, mtev_sort_next_function next,
                mtev_sort_set_next_function set_next, mtev_sort_compare_function compare)
```

- `next` the function to call to get the next pointer from a node
- `set_next` the function to call to alter the item directly after current
- `compare` the function to call to compare 2 nodes

mkdir_for_file

> Create directories along a path.

```
int
mkdir_for_file(const char *file, mode_t m)
```

- `file` a filename for which a directory is desired.
- `m` the mode used for creating directories.
- **RETURN** Returns 0 on success, -1 on error.

Creates all directories from / (as needed) to hold a named file.

N

mtev_now_ms

```
uint64_t
mtev_now_ms()
```

> the current system time in milliseconds

```
* **RETURN** mtev_gettimeofday() in milliseconds since epoch
```

mtev_now_us

```
uint64_t
mtev_now_us()
```

> the current system time in microseconds

```
* **RETURN** mtev_gettimeofday() in microseconds since epoch
```

R

mtev_rand

> Generate a pseudo-random number between [0,2^64)

```
uint64_t
mtev_rand(void)
```

- **RETURN** A pseudo-random number in the range [0,2^64)

mtev_rand_buf

> Fill a buffer with pseudo-random bytes.

```
size_t
mtev_rand_buf(void *buf, size_t len)
```

- `buf` A buffer to fill.
- `len` The number of bytes to populate.
- **RETURN** The number of bytes written to `buf` (always `len`).

mtev_rand_buf_secure

> Fill a buffer with securely random bytes.

```
size_t
mtev_rand_buf_secure(void *buf, size_t len)
```

- `buf` A buffer to fill.
- `len` The number of bytes to populate.
- **RETURN** The number of bytes written to `buf` (< len if insufficient entropy).

mtev_rand_buf_trysecure

> Fill a buffer with likely secure, but possibly pseudo-random bytes.

```
size_t
mtev_rand_buf_trysecure(void *buf, size_t len)
```

- `buf` A buffer to fill.
- `len` The number of bytes to populate.
- **RETURN** The number of bytes written to `buf` (always `len`).

mtev_rand_secure

> Generate a secure random number.

```
int
mtev_rand_secure(uint64_t *out)
```

- `out` A pointer to a `uint64_t` in which a securely generated random number will be stored.
- **RETURN** 0 on success, -1 on failure (not enough entropy available).

mtev_rand_trysecure

> Generate a likely secure, but possibly pseudo-random number between [0,2^64)

```
uint64_t
mtev_rand_trysecure(void)
```

- **RETURN** A random pseudo-random number in the range [0,2^64)

S

mtev_security_chroot

> chroot(2) to the specified directory.

```
int
mtev_security_chroot(const char *path)
```

- `path` The path to chroot to.
- **RETURN** Zero is returned on success.

mtev_security_chroot placing the calling application into a chroot environment.

mtev_security_setcaps

> change the capabilities of the process

```
int
mtev_security_setcaps(mtev_security_captype_t type, const char *capstring)
```

- `which` the effective, inherited or both
- `capstring` alteration to the capabilities
- **RETURN** Zero is returned on success.

mtev_security_setcaps will change the capability set of the current process.

mtev_security_usergroup

> change the effective or real, effective and saved user and group

```
int
mtev_security_usergroup(const char *user, const char *group, mtev_boolean effective)
```

- `user` The user name as either a login or a userid in string form.
- `group` The group name as either a login or a groupid in string form.
- `effective` If true then only effective user and group are changed.
- **RETURN** Zero is returned on success.

mtev_security_usergroup will change the real, effective, and saved user and group for the calling process. This is thread-safe.

mtev_sem_destroy

> releases all resources related to a semaphore

```
int
mtev_sem_destroy(mtev_sem_t *s)
```

- `s` the semaphore to destroy
- **RETURN** 0 on success or -1 on failure

mtev_sem_getvalue

> retrieves the current value of a semaphore, placing it in *value

```
int
mtev_sem_getvalue(mtev_sem_t *s, int *value)
```

- `s` the semaphore on which to operate
- `value` a pointer an integer that will be populated with the current value of the semaphore
- **RETURN** 0 on success or -1 on failure

mtev_sem_init

> initializes a counting semaphore for first time use.

```
int
mtev_sem_init(mtev_sem_t *s, int unused, int value)
```

- `s` the semaphore to be initialized
- `unused` is unused (keeps API combatibility with sem_init()
- `value` sets the initial value of the semaphore
- **RETURN** 0 on success or -1 on failure

mtev_sem_post

> increments the value of the semaphore releasing any waiters.

```
int
mtev_sem_post(mtev_sem_t *s)
```

- `s` the semaphore on which to wait
- **RETURN** 0 on success or -1 on failure

mtev_sem_trywait

> decrements the value of the semaphore if greater than 0 or fails

```
int
mtev_sem_trywait(mtev_sem_t *s)
```

- `s` the semaphore on which to wait
- **RETURN** 0 on success or -1 on failure

mtev_sem_wait

> decrements the value of the semaphore waiting if required.

```
int
mtev_sem_wait(mtev_sem_t *s)
```

- `s` the semaphore on which to wait
- **RETURN** 0 on success or -1 on failure

mtev_sem_wait_noeintr

> Convenience function that delegates to sem_wait, swallowing EINTR errors.

```
void
mtev_sem_wait_noeintr(mtev_sem_t *s)
```

- `s` the semaphore on which to wait
- **RETURN** 0 on success or -1 on failure

This function is built on sem_wait, and sem_wait can fail in other ways besides EINTR, such as due to EINVAL. These other failures will be due to mis-use of the semaphore API -- e.g., by trying to `sem_wait` on a structure that was never initialized with `sem_init`. Mis-using thread-synchronization primitives will often lead to subtle, serious, and hard-to-reproduce bugs, so this function will mtevFatal on other errors, rather than forcing clients to deal with (likely) un-handlable errors. If your client code can handle these other errors, use sem_wait directly, do not use this function.

mtev_sort_compare_function

> Function definition to compare sortable entries

```
int
mtev_sort_compare_function(void *left, void *right)
```

- `left` one object to compare
- `right` the other object to compare
- **RETURN** less than zero, zero, or greater than zero if left is less than, equal, or greater than right.

mtev_sort_next_function

> Function definition to get the next item from current

```
void *
mtev_sort_next_function(void *current)
```

- `current` the current node
- **RETURN** the item after current

mtev_sort_set_next_function

> Function definition to re-order objects

```
int
mtev_sort_set_next_function(void *current, void *value)
```

- `current` the current node
- `value` the value that should be directly after current

mtev_sys_gethrtime

> Exposes the system gethrtime() or equivalent impl

```
mtev_hrtime_t
mtev_sys_gethrtime(void)
```

- **RETURN** mtev_hrtime_t the system high-res time

T

mtev_time_fast_mode

> check to see if fast mode is enabled

```
mtev_boolean
mtev_time_fast_mode(const char **reason)
```

- **RETURN** true if fast mode is on, false otherwise, the reason param will contain a text description

mtev_time_maintain

> Usually this is managed for you, but this is safe to call at any time

```
mtev_boolean
mtev_time_maintain(void)
```

- **RETURN** mtev_true if it was successful in parameterizing the CPU for rdtsc, mtev_false otherwise

Safe to call at any time but if you start_tsc, you should never need to call this as the maintenance system can do it for you. However, if you find you need to call it you must be bound to a thread using the mtev_thread APIs and the function will return whether it was successful in parameterizing the CPU for rdtsc use.

mtev_time_start_tsc

> use TSC clock if possible for this CPU num

```
void
mtev_time_start_tsc()
```

This will remain active in the thread until you call stop

mtev_time_stop_tsc

> Turn off TSC usage for the current cpu of this thread (from when start_tsc was called)

```
void
mtev_time_stop_tsc(void)
```

mtev_time_toggle_require_invariant_tsc

> will switch on/off the requirement of an invariant tsc. This must be run before any call to mtev_time_toggle_tsc() or mtev_time_tsc_start() and is a one time call.

```
void
mtev_time_toggle_require_invariant_tsc(mtev_boolean enable)
```

Defaults to enabled.

mtev_time_toggle_tsc

> will switch on/off rdtsc usage across all cores regardless of detected state of rdtsc or start/stop usage.

```
void
mtev_time_toggle_tsc(mtev_boolean enable)
```

Defaults to enabled.

This is independent of start_tsc/stop_tsc. You can disable all and then reenable and the thread will keep going using the state from the last start/stop_tsc

U

mtev_url_decode

> Decode a url encoded input buffer into the provided output buffer.

```
int
mtev_url_decode(const char *src, size_t src_len, unsigned char *dest, size_t dest_len)
```

- `src` The buffer containing the encoded content.
- `src_len` The size (in bytes) of the encoded data.
- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the decoded output. Returns zero is dest_len is too small.

mtev_url_decode decodes input until an the entire input is consumed or until an invalid url-encoded character is encountered. If any error occurs, 0 is returned.

mtev_url_encode

> Encode raw data as url encoded output into the provided buffer.

```
int
mtev_url_encode(const unsigned char *src, size_t src_len, char *dest, size_t dest_len)
```

- `src` The buffer containing the raw data.
- `src_len` The size (in bytes) of the raw data.
- `dest` The destination buffer to which the function will produce.
- `dest_len` The size of the destination buffer.
- **RETURN** The size of the encoded output. Returns zero is out_sz is too small.

mtev_url_encode_len

> Calculate how large a buffer must be to contain the url encoding for a given number of bytes.

```
size_t
mtev_url_encode_len(size_t src_len)
```

- `src_len` The size (in bytes) of the raw data buffer that might be encoded.
- **RETURN** The size of the buffer that would be needed to store an encoded version of an input string.

mtev_url_max_decode_len

> Calculate how large a buffer must be to contain a decoded url-encoded string of a given length.

```
size_t
mtev_url_max_decode_len(size_t src_len)
```

- `src_len` The size (in bytes) of the url-encoded string that might be decoded.
- **RETURN** The size of the buffer that would be needed to decode the input string.

mtev_uuid_clear

```
void
mtev_uuid_clear(uuid_t uu)
```

> Set a uuid to the null uuid.

Follows the same semantics of uuid_clear from libuuid

mtev_uuid_compare

```
int
mtev_uuid_compare(const uuid_t uu1, const uuid_t uu2)
```

> Compare to uuids

- **RETURN** 0 if equal, -1 if uu1 is less than uu2, 1 otherwise.

  Follows the same semantics of uuid_compare from libuuid

mtev_uuid_copy

```
void
mtev_uuid_copy(uuid_t dst, const uuid_t src)
```

> Copy src to dst.

Follows the same semantics of uuid_copy from libuuid

mtev_uuid_generate

```
void
mtev_uuid_generate(uuid_t uu)
```

> Generate a V4 uuid.

Follows the same semantics of uuid_generate from libuuid

mtev_uuid_is_null

```
void
mtev_uuid_is_null(const uuid_t uu)
```

> Determine if the supplied uuid is the null uuid.

- **RETURN** 0 if not null, 1 if null.

  Follows the same semantics of uuid_is_null from libuuid

mtev_uuid_parse

```
int
mtev_uuid_parse(const char *in, uuid_t uu)
```

> Parse "in" in UUID format into "uu".

- **RETURN** 0 on success, non-zero on parse error

  Follows the same semantics of uuid_parse from libuuid

mtev_uuid_unparse

```
void
mtev_uuid_unparse(const uuid_t uu, char *out)
```

> Unparse "uu" into "out".

Follows the same semantics of uuid_unparse_lower from libuuid.

There is no bounds checking of "out", caller must ensure that "out" is at least UUID_STR_LEN in size. This also does not NULL terminate "out". That is also up to the caller.

mtev_uuid_unparse_lower

```
void
mtev_uuid_unparse_lower(const uuid_t uu, char *out)
```

> Unparse "uu" into "out".

Follows the same semantics of uuid_unparse_lower from libuuid.

There is no bounds checking of "out", caller must ensure that "out" is at least UUID_STR_LEN in size. This also does not NULL terminate "out". That is also up to the caller.

mtev_uuid_unparse_upper

```
void
mtev_uuid_unparse_upper(const uuid_t uu, char *out)
```

> Unparse "uu" into "out".

Follows the same semantics of uuid_unparse_upper from libuuid.

There is no bounds checking of "out", caller must ensure that "out" is at least UUID_STR_LEN in size. This also does not NULL terminate "out". That is also up to the caller.

W

mtev_watchdog_child_eventer_heartbeat

```
int
mtev_watchdog_child_eventer_heartbeat()
```

- **RETURN** Returns zero on success

mtev_watchdog_child_eventer_heartbeat registers a periodic heartbeat through the eventer subsystem. The eventer must be initialized before calling this function.

mtev_watchdog_child_heartbeat

```
int
mtev_watchdog_child_heartbeat()
```

- **RETURN** Returns zero on success

mtev_watchdog_child_heartbeat is called within the child function to alert the parent that the child is still alive and functioning correctly.

mtev_watchdog_create

```
mtev_watchdog_t *
mtev_watchdog_create()
```

- **RETURN** a new heartbeat identifier (or null, if none could be allocated)

mtev_watchdog_create creates a new heartbeat that must be assessed for liveliness by the parent.

mtev_watchdog_disable

```
void
mtev_watchdog_disable(mtev_watchdog_t *hb)
```

- `hb` the heart on which to act

mtev_watchdog_disable will make the parent ignore failed heartbeats.

mtev_watchdog_disable_asynch_core_dump

> Disable asynchronous core dumps.

```
void
mtev_watchdog_disable_asynch_core_dump(void)
```

Disable starting a new child while a faulting prior child is still dumping. This must be called before `mtev_main` and will be overridden by the MTEV_ASYNCH_CORE_DUMP environment variable.

mtev_watchdog_enable

```
void
mtev_watchdog_enable(mtev_watchdog_t *hb)
```

- `hb` the heart on which to act

mtev_watchdog_enable will make the parent respect and act on failed heartbeats.

mtev_watchdog_get_name

```
const char *
mtev_watchdog_get_name(mtev_watchdog_t *hb)
```

- `hb` the heart from which to retrieve a name
- **RETURN** the name of the heart (or NULL)

mtev_watchdog_get_timeout

> returns the timeout configured for this watchdog.

```
double
mtev_watchdog_get_timeout(mtev_watchdog_t *hb)
```

- `hb` the heart on which to act
- **RETURN** A timeout in seconds, 0 if hb is NULL.

mtev_watchdog_get_timeout_timeval

> returns the timeout configured for this watchdog.

```
struct timeval
mtev_watchdog_get_timeout_timeval(mtev_watchdog_t *hb)
```

- `hb` the heart on which to act
- `dur` a struct timeval to populate with the timeout
- **RETURN** mtev_true if there is a watchog, mtev_false if not.

mtev_watchdog_glider

> Sets a glider command.

```
int
mtev_watchdog_glider(const char *path)
```

- `path` the full path to the executable.
- **RETURN** 0 on success, non-zero on failure.

`path` is invoked with two parameters, the process id of the faulting child, and the reason for the fault (one of `crash`, `watchdog`, or `unknown`.

mtev_watchdog_glider_trace_dir

> Set the directory to store glider output.

```
int
mtev_watchdog_glider_trace_dir(const char *path)
```

- `path` a full path to a directory.
- **RETURN** 0 on success, non-zero on failure.

mtev_watchdog_heartbeat

```
int
mtev_watchdog_heartbeat(mtev_watchdog_t *hb)
```

- `hb` is the heart on which to pulse. If null, the default heart is used.
- **RETURN** Returns zero on success

mtev_watchdog_heartbeat will pulse on the specified heart.

mtev_watchdog_manage

> Ask the watchdog to manage a child process

```
void
mtev_watchdog_manage(const char *file, const char **argv, const char **envp, mtev_log_stream_t out
                     mtev_log_stream_t err)
```

- `file` The process executable
- `argv` The arguments to the process
- `envp` The environment of the process
- `user` The user (if not NULL) to setuid to.
- `group` The user (if not NULL) to setgid to.
- `dir` The directory (if not NULL) to chdir to.
- `out` A log stream for capturing stdout
- `err` A log stream for capturing stderr

An auto-restarting execve(...)

mtev_watchdog_number_of_starts

> Determine the number of times a child has been lauched.

```
uint32_t
mtev_watchdog_number_of_starts(void)
```

- **RETURN** The number of times fork has returned in the parent. In a running server, 0 means you're the first generation.

mtev_watchdog_override_timeout

```
void
mtev_watchdog_override_timeout(mtev_watchdog_t *hb, double timeout)
```

- `hb` the heart on which to act
- `timeout` the timeout in seconds for this heart (0 for default)

mtev_watchdog_override_timeout will allow the caller to override the timeout for a specific heart in the system.

mtev_watchdog_prefork_init

> Prepare the program to split into a child/parent-monitor relationship.

```
int
mtev_watchdog_prefork_init()
```

- **RETURN** Returns zero on success.

mtev_watchdog_prefork_init sets up the necessary plumbing to bridge across a child to instrument watchdogs.

mtev_watchdog_ratelimit

> Set rate limiting for child restarting.

```
void
mtev_watchdog_ratelimit(int retry_val, int span_val)
```

- `retry_val` the number of times to retry in the given `span_val`
- `span_val` the number of seconds over which to attempt retries.

mtev_watchdog_recurrent_heartbeat

```
eventer_t
mtev_watchdog_recurrent_heartbeat(mtev_watchdog_t *hb)
```

- `hb` is the heart on which to beat.
- **RETURN** Returns and event that the caller must schedule.

mtev_watchdog_recurrent_heartbeat creates a recurrent eventer_t to beat a heart.

mtev_watchdog_set_name

```
void
mtev_watchdog_set_name(mtev_watchdog_t *hb, const char *name)
```

- `hb` the heart to name
- `name` a new name for this heart

mtev_watchdog_start_child

> Starts a function as a separate child under close watch.

```
int
mtev_watchdog_start_child(const char *app, int (*func)(), int child_watchdog_timeout)
```

- `app` The name of the application (for error output).
- `func` The function that will be the child process.
- `child_watchdog_timeout` The number of seconds of lifelessness before the parent reaps and restarts the child.
- **RETURN** Returns on program termination.

mtev_watchdog_start_child will fork and run the specified function in the child process. The parent will watch. The child process must initialize the eventer system and then call mtev_watchdog_child_hearbeat to let the parent know it is alive. If the eventer system is being used to drive the child process, mtev_watchdog_child_eventer_heartbeat may be called once after the eventer is initalized. This will induce a regular heartbeat.

mtev_websocket_client_free

> Free a client

```
void
mtev_websocket_client_free(mtev_websocket_client_t *client)
```

- `client` client to be freed

This function will cleanup the client(and hence trigger any set cleanup_callback) first. This function does nothing if called with NULL.

mtev_websocket_client_get_closure

> Access the currently set closure, if any

```
void *
mtev_websocket_client_get_closure(mtev_websocket_client_t *client)
```

- `client` client to be accessed
- **RETURN** most recently set closure, or NULL if never set

mtev_websocket_client_init_logs

> Enable debug logging to "debug/websocket_client"

```
void
mtev_websocket_client_init_logs()
```

Error logging is always active to "error/websocket_client".

mtev_websocket_client_is_closed

> Check if a client has closed and can no longer send or receive

```
mtev_boolean
mtev_websocket_client_is_closed(mtev_websocket_client_t *client)
```

- `client` client to be checked
- **RETURN** boolean indicating whether the client is closed

Only a return value of mtev_true can be trusted(once closed, a client cannot re-open). Because the caller is unable to check this status inside of a locked section, it is possible that the client closes and invalidates the result of this function call before the caller can act on it.

mtev_websocket_client_is_ready

> Check if a client has completed its handshake and is ready to send messages

```
mtev_boolean
mtev_websocket_client_is_ready(mtev_websocket_client_t *client)
```

- `client` client to be checked
- **RETURN** boolean indicating whether the client is ready

This function will continue to return true after the client has closed.

mtev_websocket_client_new

> Construct a new websocket client

```
mtev_websocket_client_t *
mtev_websocket_client_new(const char *host, int port, const char *path, const char *service,
                          mtev_websocket_client_callbacks *callbacks, void *closure,
                          eventer_pool_t *pool, mtev_hash_table *sslconfig)
```

- `host` required, host to connect to(ipv4 or ipv6 address)
- `port` required, port to connect to on host
- `path` required, path portion of URI
- `service` required, protocol to connect with
- `callbacks` required, struct containing a msg_callback and optionally ready_callback and cleanup_callback
- `closure` optional, an opaque pointer that is passed through to the callbacks
- `pool` optional, specify an eventer pool; thread will be chosen at random from the pool
- `sslconfig` optional, enables SSL using the contained config
- **RETURN** a newly constructed mtev_websocket_client_t on success, NULL on failure

ready_callback will be called immediately upon successful completion of the websocket handshake. msg_callback is called with the complete contents of each non-control frame received. cleanup_callback is called as the last step of cleaning up the client, after the connection has been torn down. A client returned from this constructor must be freed with `mtev_websocket_client_free`.

mtev_websocket_client_new_noref

> Construct a new websocket client that will be freed automatically after cleanup

```
mtev_boolean
mtev_websocket_client_new_noref(const char *host, int port, const char *path, const char *service,
                                mtev_websocket_client_callbacks *callbacks, void *closure,
                                eventer_pool_t *pool, mtev_hash_table *sslconfig)
```

- `host` required, host to connect to(ipv4 or ipv6 address)
- `port` required, port to connect to on host
- `path` required, path portion of URI
- `service` required, protocol to connect with
- `callbacks` required, struct containing a msg_callback and optionally ready_callback and cleanup_callback
- `closure` optional, an opaque pointer that is passed through to the callbacks

- `pool` optional, specify an eventer pool; thread will be chosen at random from the pool
- `sslconfig` optional, enables SSL using the contained config
- **RETURN** boolean indicating success/failure

Clients allocated by this function are expected to be interacted with solely through the provided callbacks. There are two guarantees the caller must make:

1. The caller must not let a reference to the client escape from the provided callbacks.
2. The caller must not call `mtev_websocket_client_free()` with a reference to this client.

mtev_websocket_client_send

> Enqueue a message

```
mtev_boolean
mtev_websocket_client_send(mtev_websocket_client_t *client, int opcode, void *buf, size_t len)
```

- `client` client to send message over
- `opcode` opcode as defined in RFC 6455 and referenced in wslay.h
- `buf` pointer to buffer containing data to send
- `len` number of bytes of buf to send
- **RETURN** boolean indicating success/failure

This function makes a copy of buf of length len. This function may fail for the following reasons:

1. The client was not ready. See mtev_websocket_client_is_ready.
2. The client was already closed. See mtev_websocket_client_is_closed.
3. Out of memory.

mtev_websocket_client_set_cleanup_callback

> Set a new cleanup_callback on an existing client

```
void
mtev_websocket_client_set_cleanup_callback(mtev_websocket_client_t *client
                                           mtev_websocket_client_cleanup_callback cleanup_callback)
```

- `client` client to modify
- `cleanup_callback` new cleanup_callback to set

mtev_websocket_client_set_closure

> Set a new closure

```
void
mtev_websocket_client_set_closure(mtev_websocket_client_t *client, void *closure)
```

- `client` client to be modified
- `closure` closure to be set

If closure is NULL, this has the effect of removing a previously set closure.

mtev_websocket_client_set_msg_callback

> Set a new msg_callback on an existing client

```
void
mtev_websocket_client_set_msg_callback(mtev_websocket_client_t *client
                                       mtev_websocket_client_msg_callback msg_callback)
```

- `client` client to modify
- `msg_callback` new msg_callback to set

mtev_websocket_client_set_ready_callback

> Set a new ready_callback on an existing client

```
void
mtev_websocket_client_set_ready_callback(mtev_websocket_client_t *client
                                         mtev_websocket_client_ready_callback ready_callback)
```

- `client` client to modify
- `ready_callback` new ready_callback to set

Z

mtev_zipkin_active_span

> Find the currently active span of work.

```
Zipkin_Span *
mtev_zipkin_active_span(eventer_t e)
```

- `e` An event object (or NULL for the current event)
- **RETURN** A span or NULL if no span is currently active.

mtev_zipkin_annotation_set_endpoint

> Sets the endpoint for an annotation.

```
void
mtev_zipkin_annotation_set_endpoint(Zipkin_Annotation *annotation, const char *service_name,
                                    bool service_name_copy, struct in_addr host, unsigned short port)
```

- `annotation` The annotation to update.
- `service_name` The service name to use.
- `service_name_copy` Whether service_name should be allocated (copied) within the span.
- `host` The IPv4 host address of theservice.
- `port` The IP port of the service.

mtev_zipkin_annotation_set_endpoint sets an endpoint for the provided annotation.

mtev_zipkin_attach_to_aco

> Attach an active span (or new child span) to an aco thread.

```
void
mtev_zipkin_attach_to_aco(Zipkin_Span *span, bool new_child, mtev_zipkin_event_trace_level_t *track)
```

- `span` An existing zipkin span.
- `new_child` Whether or not a child should be created under the provided span.
- `track` Specifies how event activity should be tracked.

mtev_zipkin_attach_to_eventer

> Attach an active span (or new child span) to an event.

```
void
mtev_zipkin_attach_to_eventer(eventer_t e, Zipkin_Span *span, bool new_child
                              mtev_zipkin_event_trace_level_t *track)
```

- `e` An event object (or NULL for the current event)
- `span` An existing zipkin span.
- `new_child` Whether or not a child should be created under the provided span.
- `track` Specifies how event activity should be tracked.

mtev_zipkin_bannotation_set_endpoint

> Sets the endpoint for an annotation.

```
void
mtev_zipkin_bannotation_set_endpoint(Zipkin_BinaryAnnotation *annotation, const char *service_name,
                                     bool service_name_copy, struct in_addr host, unsigned short port)
```

- `annotation` The annotation to update.
- `service_name` The service name to use.
- `service_name_copy` Whether service_name should be allocated (copied) within the span.
- `host` The IPv4 host address of theservice.
- `port` The IP port of the service.

mtev_zipkin_bannotation_set_endpoint sets an endpoint for the provided annotation.

mtev_zipkin_client_drop

> Discard a client span if one exists.

```
void
mtev_zipkin_client_drop(eventer_t e)
```

- `e` An event object (or NULL for the current event)

mtev_zipkin_client_new

> Create a new span for client user (remote calling)

```
void
mtev_zipkin_client_new(eventer_t e, const char *name, bool name_copy)
```

- `e` An event object (or NULL for the current event)
- `name` A string to name the span
- `name_copy` Whether name should be allocated (copied) within the span.

mtev_zipkin_client_parent_hdr

> Format a parent span HTTP header for an HTTP request.

```
bool
mtev_zipkin_client_parent_hdr(eventer_t e, char *buf, size_t len)
```

- `e` An event object (or NULL for the current event)
- `buf` An output buffer for "Header: Value"
- `len` The available space in `buf`
- **RETURN** True if successful, false if no trace is available of len is too short.

mtev_zipkin_client_publish

> Publish a client span if one exists.

```
void
mtev_zipkin_client_publish(eventer_t e)
```

- `e` An event object (or NULL for the current event)

mtev_zipkin_client_sampled_hdr

> Format a sampled HTTP header for an HTTP request.

```
bool
mtev_zipkin_client_sampled_hdr(eventer_t e, char *buf, size_t len)
```

- `e` An event object (or NULL for the current event)
- `buf` An output buffer for "Header: Value"
- `len` The available space in `buf`
- **RETURN** True if successful, false if no trace is available of len is too short.

mtev_zipkin_client_span

> Retrieve the current client span should one exist.

```
Zipkin_Span *
mtev_zipkin_client_span(eventer_t e)
```

- `e` An event object (or NULL for the current event)
- **RETURN** A span for client actions or NULL is no span exists.

mtev_zipkin_client_span_hdr

> Format a span HTTP header for an HTTP request.

```
bool
mtev_zipkin_client_span_hdr(eventer_t e, char *buf, size_t len)
```

- `e` An event object (or NULL for the current event)
- `buf` An output buffer for "Header: Value"

- `len` The available space in `buf`
- **RETURN** True if successful, false if no trace is available of len is too short.

mtev_zipkin_client_trace_hdr

> Format a trace HTTP header for an HTTP request.

```
bool
mtev_zipkin_client_trace_hdr(eventer_t e, char *buf, size_t len)
```

- `e` An event object (or NULL for the current event)
- `buf` An output buffer for "Header: Value"
- `len` The available space in `buf`
- **RETURN** True if successful, false if no trace is available of len is too short.

mtev_zipkin_default_endpoint

> Sets the default endpoint used for new spans.

```
void
mtev_zipkin_default_endpoint(const char *service_name, bool service_name_copy, struct in_addr host
                             unsigned short port)
```

- `service_name` The service name to use.
- `service_name_copy` Whether service_name should be allocated (copied) within the span.
- `host` The IPv4 host address of theservice.
- `port` The IP port of the service.

mtev_zipkin_default_endpoint sets a default endpoint for any new spans created without their own default. Use this with care, it is application global. You should likely only call this once at startup.

mtev_zipkin_default_service_name

> Sets the default service name used for new spans.

```
void
mtev_zipkin_default_service_name(const char *service_name, bool service_name_copy)
```

- `service_name` The service name to use.
- `service_name_copy` Whether service_name should be allocated (copied) within the span.

mtev_zipkin_default_service_name sets a default service name for endpoints for any new spans created without their own default. Use this with care, it is application global. You should likely only call this once at startup.

mtev_zipkin_encode

> Encode a span into the specified buffer for Zipkin.

```
size_t
mtev_zipkin_encode(unsigned char *buffer, size_t len, Zipkin_Span *span)
```

- `buffer` The target buffer.
- `len` The target buffer's size.

- `span` The span to encode.
- **RETURN** The length of a successful encoding.

mtev_zipkin_encode will take a span and encode it for Zipkin using the Thift BinaryProtocol. The return value is always the length of a successful encoding, even if the buffer supplied is too small. The caller must check the the returned length is less than or equal to the provided length to determine whether the encoding was successful. The caller may provide a NULL buffer if and only if the provided len is 0.

mtev_zipkin_encode_list

> Encode a span into the specified buffer for Zipkin.

```
size_t
mtev_zipkin_encode_list(unsigned char *buffer, size_t len, Zipkin_Span **spans, int cnt)
```

- `buffer` The target buffer.
- `len` The target buffer's size.
- `spans` The array of spans to encode.
- `cnt` The number of spans in `spans` .
- **RETURN** The length of a successful encoding.

mtev_zipkin_encode_list will take a list of spans and encode it for Zipkin using the Thift BinaryProtocol. The return value is always the length of a successful encoding, even if the buffer supplied is too small. The caller must check the the returned length is less than or equal to the provided length to determine whether the encoding was successful. The caller may provide a NULL buffer if and only if the provided len is 0.

mtev_zipkin_event_trace_level

> Globally set the default event trace level.

```
void
mtev_zipkin_event_trace_level(mtev_zipkin_event_trace_level_t level)
```

- `level` The new global default level for event tracing.

mtev_zipkin_eventer_init

> Initialize zipkin contexts for the eventer.

```
void
mtev_zipkin_eventer_init(void)
```

mtev_zipkin_get_sampling

> Get sampling probabilities for different types of traces.

```
void
mtev_zipkin_get_sampling(double *new_traces, double *parented_traces, double *debug_traces)
```

- `new_traces` probability pointer to populate
- `parented_traces` probability pointer to populate
- `debug_traces` probability pointer to populate

mtev_zipkin_get_sampling gets sampling probabilities for creating new traces. See `mtev_zipkin_sampling` and the opentracing specification for more details on what each probability means.

mtev_zipkin_sampling

> Set sampling probabilities for different types of traces.

```
void
mtev_zipkin_sampling(double new_traces, double parented_traces, double debug_traces)
```

- `new_traces` probability of createing a new trace (trace_id == NULL)
- `parented_traces` probability of createing a parented trace (parent_span_id == NULL)
- `debug_traces` probability of createing a debug trace (debug != NULL && *debug)

mtev_zipkin_sampling sets sampling probabilities for creating new traces. Default values are 1.0

mtev_zipkin_span_annotate

> Annotate a span.

```
Zipkin_Annotation *
mtev_zipkin_span_annotate(Zipkin_Span *span, int64_t *timestamp, const char *value, bool value_copy)
```

- `span` The span to annotate.
- `timestamp` A pointer the number of microseconds since epoch. NULL means now.
- `value` The annotation value itself.
- `value_copy` Whether value should be allocated (copied) within the span.
- **RETURN** A new annotation.

mtev_zipkin_span_annotate make an annotation on the provided span. The returned resource is managed by the span and will be released with it.

mtev_zipkin_span_attach_logs

> Enable mtev_log appending if span is active.

```
void
mtev_zipkin_span_attach_logs(Zipkin_Span *span, bool on)
```

- `span` A zipkin span (NULL allowed)
- `on` Wether to enable or disable log appending.

mtev_zipkin_span_bannotate

> Annotate a span.

```
Zipkin_BinaryAnnotation *
mtev_zipkin_span_bannotate(Zipkin_Span *span, Zipkin_AnnotationType annotation_type, const char *key,
                           bool key_copy, const void *value, int32_t value_len, bool value_copy)
```

- `span` The span to annotate.
- `annotation_type` The type of the value being passed in.
- `key` The key for the annotation

- `key_copy` Whether key should be allocated (copied) within the span.
- `value` The pointer to a value for the annotation.
- `value_len` The length (in memory) of the binary value.
- `value_copy` Whether value should be allocated (copied) within the span.
- **RETURN** A new binary annotation.

mtev_zipkin_span_bannotate make a binary annotation on the provided span. The returned resource is managed by the span and will be released with it.

mtev_zipkin_span_bannotate_double

> Annotate a span.

```
Zipkin_BinaryAnnotation *
mtev_zipkin_span_bannotate_double(Zipkin_Span *span, const char *key, bool key_copy, double value)
```

- `span` The span to annotate.
- `annotation_type` The type of the value being passed in.
- `key` The key for the annotation
- `key_copy` Whether key should be allocated (copied) within the span.
- `value` The value for the annotation.
- **RETURN** A new binary annotation.

mtev_zipkin_span_bannotate_i32

> Annotate a span.

```
Zipkin_BinaryAnnotation *
mtev_zipkin_span_bannotate_i32(Zipkin_Span *span, const char *key, bool key_copy, int32_t value)
```

- `span` The span to annotate.
- `annotation_type` The type of the value being passed in.
- `key` The key for the annotation
- `key_copy` Whether key should be allocated (copied) within the span.
- `value` The value for the annotation.
- **RETURN** A new binary annotation.

mtev_zipkin_span_bannotate_i64

> Annotate a span.

```
Zipkin_BinaryAnnotation *
mtev_zipkin_span_bannotate_i64(Zipkin_Span *span, const char *key, bool key_copy, int64_t value)
```

- `span` The span to annotate.
- `annotation_type` The type of the value being passed in.
- `key` The key for the annotation
- `key_copy` Whether key should be allocated (copied) within the span.
- `value` The value for the annotation.
- **RETURN** A new binary annotation.

mtev_zipkin_span_bannotate_str

> Annotate a span.

```
Zipkin_BinaryAnnotation *
mtev_zipkin_span_bannotate_str(Zipkin_Span *span, const char *key, bool key_copy, const char *value
                               bool value_copy)
```

- `span` The span to annotate.
- `annotation_type` The type of the value being passed in.
- `key` The key for the annotation
- `key_copy` Whether key should be allocated (copied) within the span.
- `value` The value for the annotation.
- `value_copy` Whether value should be allocated (copied) within the span.
- **RETURN** A new binary annotation.

mtev_zipkin_span_default_endpoint

> Sets the default endpoint used for new annotations within the span.

```
void
mtev_zipkin_span_default_endpoint(Zipkin_Span *span, const char *service_name, bool service_name_copy,
                                  struct in_addr host, unsigned short port)
```

- `span` The span to update.
- `service_name` The service name to use.
- `service_name_copy` Whether service_name should be allocated (copied) within the span.
- `host` The IPv4 host address of theservice.
- `port` The IP port of the service.

mtev_zipkin_span_default_endpoint sets a default endpoint for any annotations or binary_annotations added to the span. All annotations added without an endpoint will use the last default set on the span.

mtev_zipkin_span_drop

> Release resources allociated with span without publishing.

```
void
mtev_zipkin_span_drop(Zipkin_Span *span)
```

- `span` The span to release.

mtev_zipkin_span_drop releases all resources associated with the span.

mtev_zipkin_span_get_ids

> Fetch the various IDs from a span.

```
bool
mtev_zipkin_span_get_ids(Zipkin_Span *span, int64_t *traceid, int64_t *parent_id, int64_t *id)
```

- `span` The span on which to operate.
- `traceid` A pointer to a trace id to populate.
- `parent_id` A pointer to a parent span id to populate.
- `span_id` A pointer to a span id to populate.

- **RETURN** True if the span has a parent, false otherwise.

mtev_zipkin_span_logs_attached

> Report whether a span should have logs attached.

```
bool
mtev_zipkin_span_logs_attached(Zipkin_Span *span)
```

- `span` A zipkin span to report on.

mtev_zipkin_span_new

> Allocate a new tracing span.

```
Zipkin_Span *
mtev_zipkin_span_new(int64_t *trace_id, int64_t *parent_span_id, int64_t *span_id,
                     const char *name, bool name_copy, bool debug, bool force)
```

- `trace_id` A pointer to the trace_id, if NULL, one will be assigned.
- `parent_span_id` A point to the span's parent_id (NULL is originating).
- `span_id` A pointer to the span's id (NULL will imply that trace_id should be used).
- `name` A name for this span.
- `name_copy` Wether the name should be allocated (copied) within the span.
- `debug` Pointer to whether this is a debug span (bypasses any sampling), NULL allowed.
- `force` force the span to be created as if all probabilities were 1.
- **RETURN** A new span.

mtev_zipkin_span_new allocates a new span in the system. The caller must eventually release the span via a call to either mtev_zipkin_span_drop or mtev_zipkin_span_publish.

mtev_zipkin_span_publish

> Pulish then release resources alloiciated with span without publishing.

```
void
mtev_zipkin_span_publish(Zipkin_Span *span)
```

- `span` The span to publish and release.

mtev_zipkin_span_publish first publishes, then releases all resources associated with the span.

mtev_zipkin_span_ref

> Increase the reference count to a span.

```
void
mtev_zipkin_span_ref(Zipkin_Span *span)
```

- `span` The span to reference.

mtev_zipkin_span_rename

> Rename a span after it has been created, but before publishing.

```
void
mtev_zipkin_span_rename(Zipkin_Span *span, const char *name, bool name_copy)
```

- `span` The span to rename.
- `name` The new name for the span.
- `name_copy` If the passed name will be freed or lost (copy required).

mtev_zipkin_str_to_id

> Convert a string Id to an int64_t Id.

```
int64_t *
mtev_zipkin_str_to_id(const char *in, int64_t *buf)
```

- `in` Id in string form
- `buf` working buffer (must not be NULL)
- **RETURN** pointer to translated id

mtev_zipkin_str_to_id will take string form id (trace_id, parent_span_id, or span_id) and convert it to an int64_t. If conversion fails, the function will return NULL.

mtev_zipkin_timeval_to_timestamp

> Convert a struct timeval to a timestamp.

```
int64_t
mtev_zipkin_timeval_to_timestamp(struct timeval *tv)
```

- `tv` A point to a struct timeval representing the time in question.
- **RETURN** a timestamp suitable for use in annotations.

mtev_zipkin_timeval_to_timestamp wil convert a struct timeval (e.g. from gettimeofday) to a the "microseconds since epoch" format expected by Zipkin.

A

mtev.Api:get

> Isse a GET request

```
api_response =
mtev.Api:get(path, payload, headers)
```

mtev.Api:http

> Wraps an HTTP Api

```
api =
mtev.Api:http(host, port, [headers])
```

Example:

```
local api = mtev.Api:http(host, port, [headers])
local result_text = api:get("/"):check():text()
local result_table = api:get("/"):check():json()
```

mtev.Api:https

> Wraps an HTTPS Api

```
api =
mtev.Api:https(host, port, [headers], [sslconfig])
```

mtev.Api:post

> Issue a POST request

```
api_response =
mtev.Api:post(path, payload, headers)
```

mtev.Api:put

> Issue a PUT request

```
api_response =
mtev.Api:put(path, payload, headers)
```

mtev.Api:request

> Issue a HTTP(S) request

```
api_response =
mtev.Api:request(method, path, payload, [headers])
```

- **RETURN** an mtev.ApiResponse object

mtev.ApiResponse:check

> Raise and error unless rc == 200

```
self =
mtev.ApiResponse:check()
```

- **RETURN** self

mtev.ApiResponse:json

```
t =
mtev.ApiResponse:json()
```

- **RETURN** parsed payload of response as table t

mtev.ApiResponse:rc

```
rc =
mtev.ApiResponse:rc()
```

mtev.ApiResponse:text

> return payload of response as string

```
text =
mtev.ApiResponse:text()
```

mtev.ApiResponse:xml

```
t =
mtev.ApiResponse:xml()
```

- **RETURN** parsed payload of response as table mtev.xmldoc

B

mtev.base64_decode

```
mtev.base64_decode()
```

mtev.base64_encode

```
mtev.base64_encode()
```

C

mtev.cancel_coro

```
mtev.cancel_coro()
```

mtev.chmod

> Change the mode of a file.

```
rv =
mtev.chmod(file, mode)
```

- `file` the path to a target file.
- `a` new file mode.
- **RETURN** rv is the return as documented by the `chmod` libc call.

mtev.close

> Close a file descripto.

```
mtev.close(fd)
```

- `fd` the integer file descriptor to close.

mtev.cluster

```
cluster =
mtev.cluster(name)
```

- `name` name of cluster
- **RETURN** a cluster object or nil if no cluster of that name is found.

mtev.conf_get_boolean

```
mtev.conf_get_boolean()
```

mtev.conf_get_float

```
mtev.conf_get_float()
```

mtev.conf_get_integer

```
mtev.conf_get_integer()
```

mtev.conf_get_string

```
mtev.conf_get_string()
```

mtev.conf_get_string_list

```
mtev.conf_get_string_list()
```

mtev.conf_replace_boolean

```
mtev.conf_replace_boolean()
```

mtev.conf_replace_value

```
mtev.conf_replace_value()
```

D

mtev.dns

> Create an `mtev.dns` object for DNS lookups.

```
mtev.dns =
mtev.dns(nameserver = nil)
```

- `nameserver` an optional argument specifying the nameserver to use.
- **RETURN** an `mtev.dns` object.

This function creates an `mtev.dns` object that can be used to perform lookups and IP address validation.

mtev.dns:is_valid_ip

> Determine address family of an IP address.

```
bool, family =
mtev.dns:is_valid_ip(ipstr)
```

- `ipstr` a string of an potential IP address.
- **RETURN** if the address is valid and, if it is, the family.

The first return is true if the suplied string is a valid IPv4 or IPv6 address, otherwise false. If the address is valid, the second argument will be the address family as an integer, otherwise nil.

mtev.dns:lookup

> Perform a DNS lookup.

```
record =
mtev.dns:lookup(query, rtype = "A", ctype = "IN")
```

- `query` a string representing the DNS query.
- `rtype` the DNS resource type (default "A").
- `ctype` the DNS class type (default "IN").
- **RETURN** a lua table, nil if the lookup fails.

DNS lookup works cooperatively with the eventer to schedule an lookup and yield the current coroutine to the event loop. If successful the table returned will contain field(s) for the requested resource. Possible fields are:

- `a` and `ttl`
- `aaaa` and `ttl`
- `mx` and `preference`
- `cname` and `ttl`
- `ptr` and `ttl`
- `ns` and `ttl`
- `mb` and `ttl`
- `md` and `ttl`
- `mf` and `ttl`
- `mg` and `ttl`
- `mr` and `ttl`

E

mtev.enable_log

> Enable or disable a log facility by name.

```
mtev.enable_log(facility, flags = true)
```

- `facility` the name of the mtev_log_stream (e.g. "debug")
- `flags` true enables, false disables

mtev.eventer:accept

> Accept a new connection.

```
mtev.eventer =
mtev.eventer:accept()
```

- **RETURN** a new eventer object representing the new connection.

mtev.eventer:bind

> Bind a socket to an address.

```
rv, err =
mtev.eventer:bind(address, port)
```

- `address` the IP address to which to bind.
- `port` the port to which to bind.
- **RETURN** rv is 0 on success, on error rv is non-zero and err contains an error message.

mtev.eventer:close

> Closes the socket.

```
rv =
mtev.eventer:close()
```

mtev.eventer:connect

> Request a connection on a socket.

```
rv, err =
mtev.eventer:connect(target[, port][, timeout])
```

- `target` the target address for a connection. Either an IP address (in which case a port is required), or a `reverse:` connection for reverse tunnelled connections.
- `timeout` for connect operation
- **RETURN** rv is 0 on success, non-zero on failure with err holding the error message.

mtev.eventer:listen

> Listen on a socket.

```
rv, errno, err =
mtev.eventer:listen(backlog)
```

- `backlog` the listen backlog.
- **RETURN** rv is 0 on success, on failure rv is non-zero and errno and err contain error information.

mtev.eventer:own

> Declare ownership of an event within a spawned co-routine.

```
ev =
mtev.eventer:own()
```

- **RETURN** New eventer object 'ev' that is owed by the calling co-routine

The old eventer object will be disowned and invalid for use!

mtev.eventer:peer_name

> Get details of the remote side of a socket.

```
address, port =
mtev.eventer:peer_name()
```

- **RETURN** local address, local port

mtev.eventer:read

> Read data from a socket.

```
payload =
mtev.eventer:read(stop)
```

- `stop` is either an integer describing a number of bytes to read or a string describing an inclusive read terminator.
- **RETURN** the payload read, or nothing on error.

mtev.eventer:recv

> Receive bytes from a socket.

```
rv, payload, address, port =
mtev.eventer:recv(nbytes)
```

- `nbytes` the number of bytes to receive.
- **RETURN** rv is the return of the `recvfrom` libc call, < 0 if error, otherwise it represents the number of bytes received. payload is a lua string representing the data received. address and port are those of the sender of the packet.

mtev.eventer:send

> Send data over a socket.

```
nbytes, err =
mtev.eventer:send(payload)
```

- `payload` the payload to send as a lua string.
- **RETURN** bytes is -1 on error, otherwise the number of bytes sent. err contains error messages.

mtev.eventer:sendto

> Send data over a disconnected socket.

```
nbytes, err =
mtev.eventer:sendto(payload, address, port)
```

- `payload` the payload to send as a lua string.
- `address` is the destination address for the payload.
- `port` is the destination port for the payload.
- **RETURN** bytes is -1 on error, otherwise the number of bytes sent. err contains error messages.

mtev.eventer:setsockopt

> Set a socket option.

```
rv, err =
mtev.eventer:setsockopt(feature, value)
```

- `feature` is on the the OS `SO_` parameters as a string.
- `value` is the value to which `feature` should be set.
- **RETURN** rv is 0 on success, -1 on failure. err contains error messages.

mtev.eventer:sock_name

> Get details of the local side of a socket.

```
address, port =
mtev.eventer:sock_name()
```

- **RETURN** local address, local port

mtev.eventer:ssl_ctx

> Gets the SSL context associated with an SSL-upgraded event.

```
mtev.eventer.ssl_ctx =
mtev.eventer:ssl_ctx()
```

- **RETURN** an mtev.eventer.ssl_ctx object.

mtev.eventer:ssl_upgrade_socket

> Upgrade a normal TCP socket to SSL.

```
rv, err =
mtev.eventer:ssl_upgrade_socket(cert, key[, ca[, ciphers[, snihost[, layer]]]])
```

- `cert` a path to a PEM-encoded certificate file.
- `key` a path to a PEM-encoded key file.
- `ca` a path to a PEM-encoded CA chain.
- `ciphers` an OpenSSL cipher preference list.
- `snihost` the host name to which we're connecting (SNI).
- `layer` a desired SSL layer.
- **RETURN** rv is 0 on success, -1 on failure. err contains error messages.

mtev.eventer:write

> Writes data to a socket.

```
nbytes =
mtev.eventer:write(data)
```

- `data` a lua string that contains the data to write.
- **RETURN** the number of bytes written.

mtev.eventer_loop_concurrency

```
mtev.eventer_loop_concurrency()
```

mtev.exec

> Spawn process return output on stdout, stderr as strings

```
status, stdout, stderr =
mtev.exec(path, argv, env, timeout)
```

- **RETURN** status is nil if a timeout was hit, stdout, stderr contain process output

G

mtev.getaddrinfo

> Resolves host name using the OS provided getaddrinfo function

```
ipstr, family =
mtev.getaddrinfo(hostname)
```

- **RETURN** ipstr - IP address represented as a string
- **RETURN** family - "inet" or "inet6" depending on whether the returned address is IPv4, IPv6

In particular this will respect the /etc/host entries.

In the case of error, we return `false, errormsg`

mtev.getcwd

```
path =
mtev.getcwd()
```

- **RETURN** path string or nil

mtev.getip_ipv4

> Returns the local address of a connection to tgt.

```
address =
mtev.getip_ipv4([tgt])
```

- `tgt` an IP address to target (default 8.8.8.8)
- **RETURN** address - IP address as a string

mtev.gettimeofday

```
sec, usec =
mtev.gettimeofday()
```

- **RETURN** the seconds and microseconds since epoch (1970 UTC)

mtev.gunzip

```
mtev.gunzip()
```

H

mtev.hmac_sha1_encode

```
mtev.hmac_sha1_encode()
```

mtev.hmac_sha256_encode

```
mtev.hmac_sha256_encode()
```

I

mtev.inet_pton

> Wrapper around inet_pton(3). Can be used to validate IP addresses and detect the address family (IPv4,IPv6)

```
rc, family, addr =
mtev.inet_pton(address)
```

- `address` to parse
- **RETURN** rc true if address is a valid IP address, false otherwise
- **RETURN** family address family of the address, either "inet" or "inet6".
- **RETURN** addr struct in_addr as udata

J

mtev.json:document

> return a lua prepresentation of an `mtev.json` object

```
obj =
mtev.json:document()
```

- **RETURN** a lua object (usually a table)

Returns a fair representation of the underlying JSON document as native lua objects.

mtev.json:tostring

> return a JSON-formatted string of an `mtev.json` object

```
obj =
mtev.json:tostring()
```

- **RETURN** a lua string

Returns a JSON document (as a string) representing the underlying `mtev.json` object.

L

mtev.log

> write message into the libmtev logging system

```
len =
mtev.log(facility, format, ...)
```

- `facility` the name of the mtev_log_stream (e.g. "error")
- `format` a format string see printf(3c)
- `...` arguments to be used within the specified format
- **RETURN** the number of bytes written

mtev.log_enabled

> Determine the enabled status of a log.

```
boolean =
mtev.log_enabled(facility)
```

- `facility` the name of the mtev_log_stream (e.g. "debug")
- **RETURN** a boolean indicating the enabled status of the log facility

mtev.LogWatch:stop

> stop watching, drain watch queue

```
mtev.LogWatch:stop()
```

mtev.LogWatch:wait

> wait for match

```
line =
mtev.LogWatch:wait(timeout)
```

- `timeout` maximial time to wait in seconds
- **RETURN** line matched or nil on timeout

M

mtev.md5

```
mtev.md5()
```

mtev.md5_hex

```
mtev.md5_hex()
```

mtev.mkdir

```
ok, errno, errstr =
mtev.mkdir(path)
```

- `path` string
- **RETURN** boolean success flag, error number, string representation of error

mtev.mkdir_for_file

```
ok, errno, errstr =
mtev.mkdir_for_file(path)
```

```
* `path` string
* **RETURN** boolean success flag, error number, string representation of error
```

N

mtev.notify

> Send notification message on given key, to be received by mtev.waitfor(key)

```
mtev.notify(key, ...)
```

- `key` key specifying notification channel
- `...` additional args to be included in the message

O

mtev.open

```
fh =
mtev.open(file, flags)
```

```
* `file` to open (string)
* `integer` flag
* **RETURN** file handle
```

The following flag constants are pre-defined: `O_RDONLY` , `O_WRONLY` , `O_RDWR` , `O_APPEND` , `O_SYNC` , `O_NOFOLLOW` , `O_CREAT` , `O_TRUNC` , `O_EXCL` see `man 2 open` for their semantics.

P

mtev.parsejson

> Convert a JSON strint to an `mtev.json` .

```
jsonobj, err, offset =
mtev.parsejson(string)
```

- `string` is a JSON formatted string.
- **RETURN** an mtev.json object plus errors on failure.

This converts a JSON string to a lua object. As lua does not support table keys with nil values, this implementation sets them to nil and thus elides the keys. If parsing fails nil is returned followed by the error and the byte offset into the string where the error occurred.

mtev.parsexml

> Parse xml string

```
mtev.parsexml(str)
```

- **RETURN** mtev.xmldoc representation

mtev.pcre

```
matcher =
mtev.pcre(pcre_expression)
```

- `pcre_expression` a perl compatible regular expression

- **RETURN** a matcher function `rv, m, ... = matcher(subject, options)`

A compiled pcre matcher function takes a string subject as the first argument and optional options as second argument.

The matcher will return first whether there was a match (true/false). If true, the next return value will be to entire scope of the match followed by any capture subexpressions. If the same subject variable is supplied, subsequent calls will act on the remainder of the subject past previous matches (allowing for global search emulation). If the subject changes, the match starting location is reset to the beginning. The caller can force a reset by calling `matcher(nil)` .

`options` is an option table with the optional fields `limit` (`PCRE_CONFIG_MATCH_LIMIT`) and `limit_recurse` (`PCRE_CONFIG_MATCH_LIMIT_RECURSION`). See the pcreapi man page for more details.

mtev.print

```
len =
mtev.print(format, ...)
```

- `format` a format string see printf(3c)
- `...` arguments to be used within the specified format
- **RETURN** the number of bytes written

This function is effectively the `mtev.log` function with the first argument set to "error". It is also aliased into the global `print` symbol such that one cannot accidentally call the print builtin.

Proc:kill

> Kill process by sending SIGTERM, then SIGKILL

```
ok, status, errno =
Proc:kill(timeout)
```

- `timeout` for the signals
- **RETURN** ok true if process was terminated, status, errno as returned by mtev.proc:wait()

mtev.Proc:loglisten

> Execute f on each line emitted to stderr

```
self =
mtev.Proc:loglisten(f)
```

mtev.Proc:loglog

> Forward process output on stderr to mtev log stream

```
self =
mtev.Proc:loglog(stream, [prefix])
```

mtev.Proc:logwatch

> Watch stderr for a line maching regexp

```
watch =
mtev.Proc:logwatch(regex, [limit])
```

- `regex` is either a regex string or a function that consumes lines
- `limit` is the maximal number of matches to find. Default infinite.
- **RETURN** watch an mtev.LogWatch object

mtev.Proc:logwrite

> Write process output on stderr to file

```
self =
mtev.Proc:logwrite(file)
```

mtev.Proc:new

> Create and control a subprocess

```
proc =
mtev.Proc:new(opts)
```

- `opts.path` path of the executable
- `opts.argv` list of command line arguments (including process name)
- `opts.dir` working directory of the process, defaults to CWD
- `opts.env` table with environment variables, defaults to ENV
- `opts.boot_match` message that signals readiness of process
- `opts.boot_timeout` time to wait until boot_match appars in stderr in seconds, defaults to 5s
- **RETURN** a Proc object

mtev.Proc:pause

> send SIGSTOP signal

```
status =
mtev.Proc:pause()
```

mtev.Proc:pid

```
pid =
mtev.Proc:pid()
```

Proc:ready

> wait for the process to become ready

```
status =
Proc:ready()
```

- **RETURN** status true/false depending on weather the process became ready Kills processes that did not become ready in time

mtev.Proc:resume

> send SIGCONT signal

```
status =
mtev.Proc:resume()
```

mtev.Proc:start

> start process

```
ok, msg =
mtev.Proc:start()
```

- **RETURN** self

mtev.Proc:wait

> wait for a process to terminate

```
term, status, errno =
mtev.Proc:wait(timeout)
```

- **RETURN** term is true if the process terminated normally; status, errno as in mtev.process:wait() In the case of normal termination, status is passed throught the WEXITSTATUS() before returning.

mtev.process:kill

> Kill a spawned process.

```
success, errno =
mtev.process:kill(signal)
```

- `signal` the integer signal to deliver, if omitted `SIGTERM` is used.
- **RETURN** true on success or false and an errno on failure.

mtev.process:pgkill

> Kill a spawned process group.

```
success, errno =
mtev.process:pgkill(signal)
```

- `signal` the integer signal to deliver, if omitted `SIGTERM` is used.
- **RETURN** true on success or false and an errno on failure.

mtev.process:pid

> Return the process id of a spawned process.

```
pid =
mtev.process:pid()
```

- **RETURN** The process id.

mtev.process:wait

> Attempt to wait for a spawned process to terminate.

```
status, errno =
mtev.process:wait(timeout)
```

- `timeout` an option time in second to wait for exit (0 in unspecified).
- **RETURN** The process status and an errno if applicable.

Wait for a process (using `waitpid` with the `WNOHANG` option) to terminate and return its exit status. If the process has not exited and the timeout has elapsed, the call will return with a nil value for status. The lua subsystem exists within a complex system that might handle process in different ways, so it does not rely on `SIGCHLD` signal delivery and instead polls the system using `waitpid` every 20ms.

R

mtev.realpath

> Return the real path of a relative path.

```
path =
mtev.realpath(inpath)
```

- `inpath` a relative path as a string
- **RETURN** The non-relative path inpath refers to (or nil on error).

mtev.rmdir

```
ok, errno, errstr =
mtev.rmdir(path)
```

- `path` string
- **RETURN** boolean success flag, error number, string representation of error

S

mtev.semaphore

> initializes semaphore with a given initial value, and returns a pointer to the semaphore object.

```
sem =
mtev.semaphore(name, value)
```

- `name` of the semaphore
- `value` initial semaphore value used if not already initialized

If a semaphore with the same name already exists, no initialization takes place, and the second argument is ignored.

Semaphores are a way to synchronize actions between different lua states.

Example:

```
sem = mtev.semaphore("my-first-semaphore", 10)
sem:acquire()
-- ... do something while holding the lock
```

```
sem:release()
```

semaphore:acquire

```
semaphore:acquire([timeout])
```

- `timeout` optional time to wait for the lock. Defaults to inifinite wait.

> returns true of the semaphore lock could be acquired within the given timeout, false if not.

At the time of this writing, the implementation of this functions uses polling. Expect delays between calls to :release() and subsequent :acquire() returning.

semaphore:release

> release the semaphore lock.

```
semaphore:release()
```

semaphore:try_acquire

> returns true of the semaphore lock could be acquired, false if not.

```
semaphore:try_acquire()
```

mtev.sh

> Run shell command, return output

```
status, stdout, stderr =
mtev.sh(command, [timeout], [shell])
```

- `command` to run
- `timeout` defaults to nil (infinite wait)
- `shell` which shell to use, defaults to $SHELL then to "/bin/sh"

mtev.sha1

```
mtev.sha1()
```

mtev.sha1_hex

```
mtev.sha1_hex()
```

mtev.sha256

```
digest =
mtev.sha256(s)
```

- `s` a string

- **RETURN** the SHA256 digest of the input string

mtev.sha256_hash

```
digest_hex =
mtev.sha256_hash(s)
```

- `s` a string
- **RETURN** the SHA256 digest of the input string, encoded in hexadecimal format

DEPRECATED

Use sha256_hex instead.

mtev.sha256_hex

```
digest_hex =
mtev.sha256_hex(s)
```

- `s` a string
- **RETURN** the SHA256 digest of the input string, encoded in hexadecimal format

mtev.shared_get

> Retrieve (via deserialization) a value at some globally named key.

```
mtev.shared_get(key)
```

- `key` must be string
- **RETURN** a lua value or nil

This function allows communication across lua states via mutex-protected storage of values.

mtev.shared_notify

> Enqueue (via serialization) a value in some globally named queue.

```
mtev.shared_notify(key, value)
```

- `key` must be string naming the queue
- `value` is a lua value simple enough to serialize (no functions, udata, etc.)
- **RETURN** none

This function allows communication across lua states.

mtev.shared_seq

> returns a sequence number that is increasing across all mtev-lua states and coroutines

```
seq =
mtev.shared_seq(keyname)
```

- `keyname` the globally unique name of the sequence to return and post-increment.

- **RETURN** seq the sequence number

mtev.shared_set

> Store (via serialization) a value at some globally named key.

```
mtev.shared_set(key, value)
```

- `key` must be string
- `value` is a lua value simple enough to serialize (no functions, udata, etc.)
- **RETURN** none

This function allows communication across lua states via mutex-protected storage of values.

mtev.shared_waitfor

> Retrieve (via deserialization) a value in some globally named queue.

```
mtev.shared_waitfor(key, timeout)
```

- `key` must be string naming the queue
- `timeout` number of sections to wait before returning nil
- **RETURN** a lua value or nil

This function allows communication across lua states.

mtev.sleep

```
slept =
mtev.sleep(duration_s)
```

- `duration_s` the number of sections to sleep
- **RETURN** the time slept as mtev.timeval object

mtev.socket

> Open a socket for eventer-friendly interaction.

```
mtev.eventer =
mtev.socket(address[, type])
```

- `address` a string 'inet', 'inet6' or an address to connect to
- `type` an optional string 'tcp' or 'udp' (default is 'tcp')
- **RETURN** an eventer object.

No connect() call is performed here, the address provided is only used to ascertain the address family for the socket.

mtev.spawn

> Spawn a subprocess.

```
mtev.process =
mtev.spawn(path, argv, env)
```

- `path` the path to the executable to spawn
- `argv` an array of arguments (first argument is the process name)
- `env` an optional array of "K=V" strings.
- **RETURN** an object with the mtev.process metatable set.

This function spawns a new subprocess running the binary specified as the first argument.

T

mtev.thread_self

```
thread, tid =
mtev.thread_self()
```

mtev.time

```
time =
mtev.time()
```

- **RETURN** the seconds since epoch (1970 UTC) as float

mtev.timezone

```
mtev.timezone =
mtev.timezone(zonename)
```

-
    - `zonename` is the name of the timezone (e.g. "UTC" or "US/Eastern")
-
    - **RETURN** an mtev.timezone object.

mtev.timezone:extract

```
a,... =
mtev.timezone:extract(time, field1, ...)
```

- `time` is the offset in seconds from UNIX epoch.
- `field1` is a field to extract in the time local to the timezone object.
- **RETURN** The value of each each requested field.

Valid fields are "second", "minute", "hour", "monthday", "month", "weekday", "yearday", "year", "dst", "offset", and "zonename."

mtev.tojson

> Convert a lua object into a json doucument.

```
jsonobj =
mtev.tojson(obj, maxdepth = -1)
```

- `obj` a lua object (usually a table).
- `maxdepth` if specified limits the recursion.

- **RETURN** an mtev.json object.

This converts a lua object, ignoring types that do not have JSON counterparts (like userdata, lightuserdata, functions, threads, etc.). The return is an `mtev.json` object not a string. You must invoke the `tostring` method to convert it to a simple string.

U

mtev.uname

> Returns info from the uname libc call

```
details =
mtev.uname()
```

- **RETURN** A table resembling the `struct utsname`

mtev.utf8tohtml

```
mtev.utf8tohtml()
```

mtev.uuid

```
mtev.uuid()
```

W

mtev.waitfor

> Suspend until for notification on key is received or the timeout is reached.

```
... =
mtev.waitfor(key, [timeout])
```

- **RETURN** arguments passed to mtev.notify() including the key.

mtev.watchdog_child_heartbeat

> Heartbeat from a child process.

```
rv =
mtev.watchdog_child_heartbeat()
```

- **RETURN** The return value of `mtev_watchdog_child_heartbeat()`

mtev.watchdog_timeout

> Return the watchdog timeout on the current thread.

```
timeout =
mtev.watchdog_timeout()
```

- **RETURN** A timeout in seconds, or nil if no watchdog configured.

mtev.WCOREDUMP

```
mtev.WCOREDUMP(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** true if the process produced a core dump

Only valid if `mtev.WIFSIGNALED(status)` is also true.

mtev.websocket_client:close

> Close a websocket client.

```
mtev.websocket_client:close()
```

mtev.websocket_client:send

```
mtev.websocket_client:send(opcode, payload)
```

- `opcode` The websocket opcode.
- `payload` The payload.

> Send a message over a websocket client.

The client object has fields exposing: `CONTINUATION` , `TEXT` , `BINARY` , `CONNECTION_CLOSE` , `PING` , and `PONG` .

mtev.websocket_client_connect

> Create a new web socket client.

```
success =
mtev.websocket_client_connect(host, port, uri, service, callbacks, sslconfig)
```

- `host` The host
- `port` The port
- `uri` The uri
- `service` The service
- `callbacks` A table of callbacks
- `sslconfig` An optional non-empty table of ssl configuration.
- **RETURN** True or false for success.

Callbacks may include:

- ready = function(mtev.websocket_client) return boolean
- message = function(mtev.websocket_client, opcode, payload) return boolean
- cleanup = function(mtev.websocket_client)

If callbacks returning boolean return false, the connection will shutdown. sslconfig can contain `ca_chain` `key` `cert` `layer` `ciphers` just as with other SSL functions.

mtev.WEXITSTATUS

```
mtev.WEXITSTATUS(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** the exit status of the process

Only valid if `mtev.WIFEXITED(status)` is true.

mtev.WIFCONTINUED

```
mtev.WIFCONTINUED(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** true if the process has continued after a job control stop, but not terminated

mtev.WIFEXITED

```
mtev.WIFEXITED(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** true if the process terminated normally

mtev.WIFSIGNALED

```
mtev.WIFSIGNALED(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** true if the process terminated due to receipt of a signal

mtev.WIFSTOPPED

```
mtev.WIFSTOPPED(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** true if the process was stopped, but not terminated

mtev.write

```
mtev.write(fd, str)
```

mtev.WSTOPSIG

```
mtev.WSTOPSIG(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** the number of the signal that caused the process to stop

Only valid if `mtev.WIFSTOPPED(status)` is true.

mtev.WTERMSIG

```
mtev.WTERMSIG(status)
```

- `status` a process status returned by `mtev.process:wait(timeout)`
- **RETURN** the number of the signal that caused the termination of the process

Only valid if `mtev.WIFSIGNALED(status)` is true.

X

mtev.xmldoc:root

```
node =
mtev.xmldoc:root()
```

- **RETURN** mtev.xmlnode containing root of document

mtev.xmldoc:tostring

```
str =
mtev.xmldoc:tostring()
```

- **RETURN** string representation of xmldoc

mtev.xmldoc:xpath

```
iter
mtev.xmldoc:xpath(xpath, [node])
```

- **RETURN** iterator over mtev.xmlnode objects

mtev.xmlnode:addchild

> Add child to the given xml node

```
child =
mtev.xmlnode:addchild(str)
```

- **RETURN** child mtev.xmlnode

mtev.xmlnode:attr

```
val =
mtev.xmlnode:attr(key)
```

mtev.xmlnode:children

```
iter =
mtev.xmlnode:children()
```

- **RETURN** iterator over child mtev.xmlnodes

mtev.xmlnode:contents

```
str =
mtev.xmlnode:contents()
```

- **RETURN** content of xml node as string

mtev.xmlnode:name

```
str =
mtev.xmlnode:name()
```

mtev.xmlnode:next

```
sibling =
mtev.xmlnode:next()
```

- **RETURN** next sibling xml node