flux-core

Oct 23, 2021

Contents:

CHAPTER 1

flux-core Manual Pages

1.1 man1

1.1.1 flux-broker(1)

SYNOPSIS

flux-broker [*OPTIONS*] [*initial-program* [*args...*]]

DESCRIPTION

flux-broker(1) is a distributed message broker daemon that provides communications services within a Flux instance. It may be launched as a parallel program under Flux or other resource managers that support PMI.

Resource manager services are implemented as dynamically loadable modules.

Brokers within a Flux instance are interconnected using ZeroMQ sockets, and each is assigned a rank from 0 to size - 1. The rank 0 node is the root of a tree-based overlay network. This network may be accessed by Flux commands and modules using Flux API services.

A logging service aggregates Flux log messages across the instance and emits them to a configured destination on rank 0.

After its overlay network has completed wire-up, flux-broker(1) starts the initial program on rank 0. If none is specified on the broker command line, an interactive shell is launched.

OPTIONS

-h, –help Summarize available options.

-v, –verbose Be annoyingly chatty.

-S, –setattr=*ATTR=VAL* Set initial value for broker attribute.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-broker-attributes(7)

1.1.2 flux-content(1)

SYNOPSIS

flux content load [*–bypass-cache*] *blobref*

flux content store [*–bypass-cache*]

flux content flush

flux content dropcache

DESCRIPTION

Each Flux instance implements an append-only, content addressable storage service, which stores blobs of arbitrary content under message digest keys termed "blobrefs".

flux content store accepts a blob on standard input, stores it, and prints the blobref on standard output.

flux content load accepts a blobref argument, retrieves the corresponding blob, and writes it to standard output.

After a store operation completes on any rank, the blob may be retrieved from any other rank.

The content service includes a cache on each broker which improves scalability. The **flux content flush** command initiates store requests for any dirty entries in the local cache and waits for them to complete. This is mainly used in testing. The **flux content dropcache** command drops all non-essential entries in the local cache; that is, entries which can be removed without data loss.

OPTIONS

-b, –bypass-cache Bypass the in-memory cache, and directly access the backing store, if available (see below).

BACKING STORE

The rank 0 cache retains all content until a module providing the "content.backing" service is loaded which can offload content to some other place. The **content-sqlite** module provides this service, and is loaded by default.

Content database files are stored persistently on rank 0 if the persist-directory broker attribute is set to a directory name for the session. Otherwise they are stored in the directory defined by the rundir attribute and are cleaned up when the instance terminates.

When one of these modules is loaded, it informs the rank 0 cache of its availability, which triggers the cache to begin offloading entries. Once entries are offloaded, they are eligible for expiration from the rank 0 cache.

To avoid data loss, once a content backing module is loaded, do not unload it unless the content cache on rank 0 has been flushed and the system is shutting down.

CACHE EXPIRATION

The parameters affecting local cache expiration may be tuned with flux-setattr(1):

content.purge-target-size The cache is purged to bring the sum of the size of cached blobs less than or equal to this value (default 16777216)

content.purge-old-entry Only entries that have not been accessed in **old-entry** seconds are eligible for purge (default 10).

Expiration becomes active on every heartbeat. Dirty or invalid entries are not eligible for purge.

CACHE ACCOUNTING

Some accounting info for the local cache can be viewed with flux-getattr(1):

content.acct-entries The total number of cache entries.

content.acct-size The sum of the size of cached blobs.

content.acct-dirty The number of dirty cache entries.

content.acct-valid The number of valid cache entries.

CACHE SEMANTICS

The cache is write-through with respect to the rank 0 cache; that is, a store operation does not receive a response until it is valid in the rank 0 cache.

The cache on rank 0 is write-back with respect to the backing store, if any; that is, a store operation may receive a response before it has been stored on the backing store.

The cache is hierarchical. Rank 0 (the root of the tree based overlay network) holds all blobs stored in the instance. Other ranks keep only what a they heuristically determine to be of benefit. On ranks > 0, a load operation that cannot be fulfilled from the local cache is "faulted" in from the level above it. A store operation that reaches a level that has already cached the same content is "squashed"; that is, it receives a response without traveling further up the tree.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

RFC 10: Content Store

1.1.3 flux-cron(1)

SYNOPSIS

flux cron *COMMAND* [*OPTIONS*]

DESCRIPTION

The Flux cron service offers an interface for executing commands on triggers such as a time interval or Flux events. The service is implemented as a Flux extension module which, when loaded, manages a set of cron entries and uses the built-in *broker.exec* service to run a command associated with the entry each time the defined trigger is reached. As with *flux-exec(1)*, these tasks run as direct children of the flux-broker and run outside of the control of any loaded job scheduling service.

The flux-cron(1) utility offers an interface to create, stop, start, query, and destroy these entries in the Flux cron service.

For a detailed description of the cron service operation and how it executes tasks, see the OPERATION and TASK EXECUTION sections below.

COMMANDS

help *cmd* Print help. If *cmd* is provided, print help for that sub-command.

sync [–epsilon=*delay*] [*topic*] Query and modify the current **sync-event** behavior for the cron module. If a sync-event is set, the cron module will defer all task execution until an event matching the sync-event *topic* is received. With *–epsilon* the cron module will **not** delay task execution if the task is normally scheduled to run within *delay* of the matching event. Without any *topic* supplied on command line, *flux cron sync* displays the current setting for sync. If a task is deferred due to sync-event, the *stats.deferred* statistic is incremented.

interval [OPTIONS] *interval command* Create a cron entry to execute *command* every *interval*, where *interval* is an arbitrary floating point duration with optional suffix *s* for seconds, *m* for minutes, *h* for hours and *d* for days. Options:

–name=*STRING*; -n *STRING* Set a name for this cron entry to *STRING*.

–after=*TIME*; -a *TIME* The first task will run after a delay of *TIME* instead of *interval*. After the first task the entry will continue to execute every *interval*.

–count=*N*; -c *N* The entry will be run a total of *N* times, then stopped.

–options=*LIST*; -o *LIST* The *–options* option allows a comma separated list of extra options to be passed to the flux-cron service. See EXTRA OPTIONS below.

–preserve-env; -E The *–preserve-env* option allows the current environment to be exported and used for the command being executed as part of the cron job. Normally, the broker environment is used.

–working-dir=*DIR*; -d *DIR* The *–working-dir* option allows the working directory to be set for the command being executed as part of the cron job. Normally, the working directory of the broker is used.

event [OPTIONS] *topic command* Create a cron entry to execute *command* after every event matching *topic*.

–name=*STRING*; -n *STRING* Set a name for this cron entry to *STRING*.

–nth=*N*; -n *N* If *–nth* is given then *command* will be run after each *N* events.

–count=*N*; -c *N* With *–count*, the entry is run *N* times then stopped.

–after=*N*; -a *N* Run the first task only after *N* matching events. Then run every event or *N* events with *–nth*.

–min-interval=*T*; -i *T* Set the minimum interval at which two cron jobs for this event will be run. For example, with –min-interval of 1s, the cron job will be at most run every 1s, even if events are generated more quickly.

–options=*LIST*; -o *LIST* Set comma separated EXTRA OPTIONS for this cron entry.

–preserve-env; -E The *–preserve-env* option allows the current environment to be exported and used for the command being executed as part of the cron job. Normally, the broker environment is used.

–working-dir=*DIR*; -d *DIR* The *–working-dir* option allows the working directory to be set for the command being executed as part of the cron job. Normally, the working directory of the broker is used.

tab [OPTIONS] [*file*] Process one or more lines containing crontab expressions from *file* (stdin by default) Each valid crontab line will result in a new cron entry registered with the flux-cron service. The cron expression format supported by `flux cron tab` has 5 fields: *minutes* (0-59), *hours* (0-23), *day of month* (1-31), *month* (0-11), and *day of week* (0-6). Everything after the day of week is considered a command to be run.

–options=*LIST*; -o *LIST* Set comma separated EXTRA OPTIONS for all cron entries.

at [OPTIONS] *string command* Run *command* at specific date and time described by *string*

–options=*LIST*; -o *LIST* Set comma separated EXTRA OPTIONS for all cron entries.

–preserve-env; -E The *–preserve-env* option allows the current environment to be exported and used for the command being executed as part of the cron job. Normally, the broker environment is used.

–working-dir=*DIR*; -d *DIR* The *–working-dir* option allows the working directory to be set for the command being executed as part of the cron job. Normally, the working directory of the broker is used.

list Display a list of current entries registered with the cron module and their current state, last run time, etc.

stop *id* Stop cron entry *id*. The entry will remain in the cron entry list until deleted.

start *id* Start a stopped cron entry *id*.

delete [–kill] *id* Purge cron entry *id* from the flux-cron entry list. If *–kill* is used, kill any running task associated with entry *id*.

dump [–key=KEY] *id* Dump all information for cron entry *id*. With *–key* print only the value for key *KEY*. For a list of keys run *flux cron dump ID*.

EXTRA OPTIONS

For `flux-cron` commands allowing `--options`, the following EXTRA OPTIONS are supported:

timeout=*N* Set a timeout for tasks invoked for this cron entry to *N* seconds, where N can be a floating point number. Default is no timeout.

rank=*R* Set the rank on which to execute the cron command to *R*. Default is rank 0.

task-history-count=*N* Keep history for the last *N* tasks invoked by this cron entry. Default is 1.

stop-on-failure=*N* Automatically stop a cron entry if the failure count exceeds *N*. If *N* is zero (the default) then the cron entry will not be stopped on failure.

OPERATION

The Flux cron module manages the set of currently configured cron jobs as a set of common entries, each with a unique ID supplied by a global sequence number and set of common attributes, options, and statistics. Basic attributes of a cron job include an optional *name*, the *command* to execute on the entry's trigger, the current *state* of the cron entry (stopped or not stopped), a *repeat* count indicating the total number of times to execute the cron job before stopping, and the *type* of entry.

All cron entries also support a less common list of options, which may be set at creation time via a comma-separated list of *option=value* parameters passed to the *-o*, *–option=OPTS*. These options are described in the EXTRA OPTIONS section at the end of this document.

Currently, flux-cron supports only two types of entries. The *interval* entry supports executing a command once every configured duration, optionally starting after a different time period. More detailed information about the interval type can be found in the documentation for the flux-cron *interval* command above. The *event* type entry supports running

a command once every N events matching the configured event topic. More information about this type can be found in the documentation for *flux cron event*.

The Flux cron module additionally keeps a common set of statistics for each entry, regardless of type . These include the creation time, last run time, and last time the cron entry was "started", as well a count of total number of times the command was executed and a count of successful and failed runs. Currently, the stats for a cron entry may be viewed via the *flux cron dump* subcommand *stats.** output.

When registered, cron entries are automatically *started*, meaning they are eligible to run the configured command when the trigger condition is met. Entries may be *stopped*, either by use of the *flux cron stop* command, or if a *stop-on-failure* value is set. Stopped entries are restarted using *flux cron start*, at which point counters used for repeat and stop-on-failure are reset.

Stopped entries are kept in the flux cron until deleted with *flux cron delete*. Active cron entries may also be deleted, with currently executing tasks optionally killed if the *–kill* option is provided.

TASK EXECUTION

As related above, cron entry commands are executed via the *broker.exec* service, which is a low level execution service offered outside of any scheduler control, described in more detail in the *flux-exec(1)* man page.

Standard output and error from tasks executed by the cron service are logged and may be viewed with *flux-dmesg(1)*. If a cron task exits with non-zero status, or fails to launch under the *broker.exec* service, a message is logged and the failure is added to the failure stats. On task failure, the cron job is stopped if *stop-on-failure* is set, and the current failure count exceeds the configured value. By default, *stop-on-failure* is not set.

By default, flux-cron module keeps information for the last task executed for each cron entry. This information can be viewed either via the *flux cron list* or *flux cron dump ID* subcommands. Data such as start and end time, exit status, rank, and PID for the task is available. The number of tasks kept for each cron entry may be individually tuned via the *task-history-count* option, described in the EXTRA OPTIONS section.

Commands are normally executed immediately on the interval or event trigger for which they are configured. However, if the *sync-event* option is active on the cron module, tasks execution will be deferred until the next synchronization event. See the documentation above for *flux cron sync* for more information.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-exec(1), flux-dmesg(1)

1.1.4 flux-dmesg(1)

SYNOPSIS

flux dmesg [*OPTIONS*]

DESCRIPTION

Each broker rank maintains a circular buffer of log entries which can be printed using flux-dmesg(1).

OPTIONS

-C, –clear Clear the ring buffer.

-c, –read-clear Clear the ring buffer after printing its contents.

-f, –follow After printing the contents of the ring buffer, wait for new entries and print them as they arrive.

EXAMPLES

To dump the ring buffer on all ranks

```
$ flux exec flux dmesg | sort
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-setattr(1), flux-broker-attributes(7)

1.1.5 flux-env(1)

SYNOPSIS

flux env [COMMAND]

DESCRIPTION

flux-env(1) dumps a list of all environment variables as set by flux if run without a command, when run with a command the environment is set and the command is run as it would be by the ENV(1) utility.

RESOURCES

Github: http://github.com/flux-framework

1.1.6 flux-event(1)

SYNOPSIS

flux event *COMMAND* [*OPTIONS*]

DESCRIPTION

Flux events are messages that are broadcast throughout the Flux instance with publish/subscribe semantics. Each event message has a *topic string* and an optional *payload*.

Subscriptions are by topic string. A subscription topic of length *N* matches an event if the first *N* characters of the event topic are identical to that of the subscription. For example the event topic *a.b.c* is matched by the subscription topic *a.b.c*, *a.b*, or *a*. A subscription to the empty string matches all events.

COMMANDS

pub [-r] [-l] [-s] [-p] *topic* [*payload*] Publish an event with optional payload. If payload is specified, it is interpreted as raw if the *-r* option is used, otherwise it is interpreted as JSON. If the payload spans multiple arguments, the arguments are concatenated with one space between them. If *-s* is specified, wait for the event's sequence number to be assigned before exiting. If *-l* is specified, subscribe to the published event and wait for it to be received before exiting. *-p* causes the privacy flag to be set on the published event.

sub *[-c N]* [*topic*] [*topic...*] Subscribe to events matching the topic string(s) provided on the command line. If none are specified, subscribe to all events. If *-c N* is specified, print the first *N* events on stdout and exit; otherwise continue printing events until a signal is received. Events are displayed one per line: the topic string, followed by a tab, followed by the payload, if any.

RESOURCES

Github: http://github.com/flux-framework

1.1.7 flux-exec(1)

SYNOPSIS

flux exec [–noinput] [*–label-io] [*—dir=DIR'] [*–rank=NODESET*] [*–verbose*] COMMANDS...

DESCRIPTION

flux-exec(1) runs commands across one or more flux-broker ranks using the *broker.exec* service. The commands are executed as direct children of the broker, and the broker handles buffering stdout and stderr and sends the output back to flux-exec(1) which copies output to its own stdout and stderr.

On receipt of SIGINT and SIGTERM signals, flux-exec(1) shall forward the received signal to all currently running remote processes.

In the event subprocesses are hanging or ignoring SIGINT, two SIGINT signals (typically sent via Ctrl+C) in short succession can force flux-exec(1) to exit.

flux-exec(1) is meant as an administrative and test utility, and cannot be used to launch Flux jobs.

EXIT STATUS

In the case that all processes are successfully launched, the exit status of flux-exec(1) is the largest of the remote process exit codes.

If a non-existent rank is targeted, flux-exec(1) will return with code 68 (EX_NOHOST from sysexits.h).

If one or more remote commands are terminated by a signal, then flux-exec(1) exits with exit code 128+signo.

OPTIONS

-l, –label-io Label lines of output with the source RANK.

-n, –noinput Do not attempt to forward stdin. Send EOF to remote process stdin.

-d, –dir=*DIR* Set the working directory of remote *COMMANDS* to *DIR*. The default is to propagate the current working directory of flux-exec(1).

-r, –rank=*NODESET* Target specific ranks in *NODESET*. Default is to target "all" ranks. See NODESET FORMAT below for more information.

-v, –verbose Run with more verbosity.

NODESET FORMAT

A NODESET is a comma separated list of integer ranks. Ranks may be listed individually or as a range in the form *l-k* where l < k.

Some examples of nodesets.

‘‘1” rank 1

‘‘0-3” ranks 0, 1, 2, and 3 listed in a range

‘‘0,1,2,3” ranks 0, 1, 2, and 3 listed individually

‘‘2,5” ranks 2 and 5

‘‘2,4-5” ranks 2, 4, and 5

As a special case, the string ‘‘*all*” can be specified to indicate every rank available in the flux instance.

RESOURCES

Github: http://github.com/flux-framework

1.1.8 flux-getattr(1)

SYNOPSIS

flux getattr *name*

flux setattr *name value*

flux setattr [*–expunge*] *name*

flux lsattr [*–values*]

DESCRIPTION

Flux broker attributes are both a simple, general-purpose key-value store with scope limited to the local broker rank, and a method for the broker to export information needed by Flux services and utilities.

flux-getattr(1) retrieves the value of an attribute.

flux-setattr(1) assigns a new value to an attribute, or optionally removes an attribute.

flux-lsattr(1) lists attribute names, optionally with their values.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_attr_get(3), flux-broker-attributes(7)

1.1.9 flux-hwloc(1)

SYNOPSIS

flux hwloc info [*OPTIONS*]

flux hwloc topology [*OPTIONS*]

DESCRIPTION

The **flux-hwloc** utility queries hwloc(7) topology information for an instance by gathering XML from the core resource module.

COMMANDS

flux hwloc requires a *COMMAND* argument. The supported commands are

info [*-l,–local*|*-r,–rank=NODESET*] Dump a short-form summary of the total number of Machines, Cores, and Processing Units (PUs) available across all flux-brokers in the current instance. With *–ranks*, dump information for only the specified ranks. With *–local* dump local system information.

topology [*-l,–local*|*-r,–rank=NODESET*] Dump current aggregate topology XML for the current session to stdout. With *–rank* only dump aggregate topology for specified ranks. With *–local* dump topology XML for the local system. With hwloc < 2.0, this command will dump a custom topology with multiple machines when the aggregate contains multiple ranks. This is not possible with hwloc 2.0 because multiple Machine objects in a topology is no longer supported, and therefore the XML for each rank will be printed separately.

NODESET FORMAT

A NODESET is a comma separated list of integer ranks. Ranks may be listed individually or as a range in the form *l-k* where l < k.

Some examples of nodesets.

"1" rank 1

"0-3" ranks 0, 1, 2, and 3 listed in a range

"0,1,2,3" ranks 0, 1, 2, and 3 listed individually

"2,5" ranks 2 and 5

"2,4-5" ranks 2, 4, and 5

As a special case, the string "*all*" can be specified to indicate every rank available in the flux instance.

EXAMPLES

When using HWLOC < 2.0 only, the output of `flux hwloc topology` may be piped to other hwloc(7) commands such as `lstopo(1)` or `hwloc-info(1)`, e.g.

```
$ flux hwloc topology | lstopo-no-graphics --if xml -i -
System (31GB total)
  Machine L#0 (7976MB) + Package L#0
    Core L#0 + PU L#0 (P#0)
    Core L#1 + PU L#1 (P#1)
    Core L#2 + PU L#2 (P#2)
    Core L#3 + PU L#3 (P#3)
  Machine L#1 (7976MB) + Package L#1
    Core L#4 + PU L#4 (P#0)
    Core L#5 + PU L#5 (P#1)
    Core L#6 + PU L#6 (P#2)
    Core L#7 + PU L#7 (P#3)
  Machine L#2 (7976MB) + Package L#2
    Core L#8 + PU L#8 (P#0)
    Core L#9 + PU L#9 (P#1)
    Core L#10 + PU L#10 (P#2)
    Core L#11 + PU L#11 (P#3)
  Machine L#3 (7976MB) + Package L#3
    Core L#12 + PU L#12 (P#0)
    Core L#13 + PU L#13 (P#1)
    Core L#14 + PU L#14 (P#2)
    Core L#15 + PU L#15 (P#3)
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

lstopo(1), hwloc: https://www.open-mpi.org/projects/hwloc/

1.1.10 flux-job(1)

SYNOPSIS

flux job cancel *id* [*message...*]

flux job cancelall [*OPTIONS*] [*message...*]

flux job kill [*–signal=SIG*] *id*

flux job killall [*OPTIONS*]

flux job raise [*OPTIONS*] *id* [*message...*]

flux job raiseall [*OPTIONS*] *type* [*message...*]

DESCRIPTION

flux-job(1) performs various job related housekeeping functions.

CANCEL

A single job may be canceled with `flux job cancel`.

Jobs may be canceled in bulk with `flux job cancelall`. Target jobs are selected with:

-u, –user=USER Set target user. The instance owner may specify *all* for all users.

-S, –states=STATES Set target job states (default: ACTIVE).

-f, –force Confirm the command

-q, –quiet Suppress output if no jobs match

SIGNAL

Running jobs may be signaled with `flux job kill`.

-s, –signal=SIG Send signal SIG (default: SIGTERM).

Running jobs may be signaled in bulk with `flux job killall`. In addition to the option above, target jobs are selected with:

-u, –user=USER Set target user. The instance owner may specify *all* for all users.

-f, –force Confirm the command.

EXCEPTION

An exception may raised on a single job with `flux job raise`.

-s, –severity=N Set exception severity. The severity may range from 0=fatal to 7=least severe (default: 0).

-t, –type=TYPE Set exception type (default: cancel).

Exceptions may be raised in bulk with `flux job raiseall`, which requires a type (positional argument) and accepts the following options:

-s, –severity=N Set exception severity. The severity may range from 0=fatal to 7=least severe (default: 7).

-u, –user=USER Set target user. The instance owner may specify *all* for all users.

-S, –states=STATES Set target job states (default: ACTIVE)

-f, –force Confirm the command.

RESOURCES

Github: http://github.com/flux-framework

1.1.11 flux-jobs(1)

SYNOPSIS

flux jobs [*OPTIONS*] [JOBID ...]

DESCRIPTION

flux-jobs(1) is used to list jobs run under Flux. By default only pending and running jobs for the current user are listed. Additional jobs and information can be listed using options listed below. Alternately, specific job ids can be listed on the command line to only list those job IDs.

OPTIONS

-a List all jobs of the current user, including inactive jobs. Equivalent to specifying *–filter=pending,running,inactive*.

-A List all jobs from all users, including inactive jobs. Equivalent to specifying *–filter=pending,running,inactive –user=all*.

-n, –suppress-header For default output, do not output column headers.

-u, –user=*[USERNAME|UID]* List jobs for a specific username or userid. Specify *all* for all users.

-c, –count=*N* Limit output to N jobs (default 1000)

-f, –filter=*STATE|RESULT* List jobs with specific job state or result. Multiple states or results can be listed separated by comma. See JOB STATUS below for additional information. Defaults to *pending,running*.

-o, –format=*FORMAT* Specify output format using Python's string format syntax. See OUTPUT FORMAT below for field names.

–color=*WHEN* Control output coloring. WHEN can be *never*, *always*, or *auto*. Defaults to *auto*.

–stats Output a summary of global job statistics before the header. May be useful in conjunction with utilities like `watch(1)`, e.g.:

```
$ watch -n 2 flux jobs --stats -f running -c 25
```

will display a summary of global statistics along with the top 25 running jobs, updated every 2 seconds.

–stats-only Output a summary of global job statistics and exit. `flux jobs` will exit with non-zero exit status with `--stats-only` if there are no active jobs. This allows the following loop to work:

```
$ while flux jobs --stats-only; do sleep 2; done
```

All other options are ignored when `--stats-only` is used.

JOB STATUS

Jobs may be observed to pass through five job states in Flux: DEPEND, SCHED, RUN, CLEANUP, and INACTIVE (see Flux RFC 21). Under the `state_single` field name, these are abbreviated as D, S, R, C, and I respectively. For convenience and clarity, the following virtual job states also exist: "pending", an alias for DEPEND,SCHED; "running", an alias for RUN,CLEANUP; "active", an alias for "pending,running".

After a job has finished and is in the INACTIVE state, it can be marked with one of three possible results: COMPLETED, FAILED, CANCELED. Under the *result_abbrev* field name, these are abbreviated as CD, F, and CA respectively.

The job status is a user friendly mix of both, a job is always in one of the following five statuses: PENDING, RUNNING, COMPLETED, FAILED, or CANCELED. Under the *status_abbrev* field name, these are abbreviated as P, R, CD, F, and CA respectively.

OUTPUT FORMAT

The *–format* option can be used to specify an output format to flux-jobs(1) using Python's string format syntax. For example, the following is the format used for the default format:

```
{id.f58:>12} {username:<8.8} {name:<10.10} {status_abbrev:>2.2} {ntasks:>6} {nnodes:>
↪6h} {runtime!F:>8h} {nodelist:h}
```

The special presentation type *h* can be used to convert an empty string, "0s", "0.0", or "0:00:00" to a hyphen. For example, normally "{nodelist}" would output an empty string if the job has not yet run. By specifying, "{nodelist:h}", a hyphen would be presented instead.

Additionally, the custom job formatter supports a set of special conversion flags. Conversion flags follow the format field and are used to transform the value before formatting takes place. Currently, the following conversion flags are supported by *flux-jobs*:

!D convert a timestamp field to ISO8601 date and time (e.g. 2020-01-07T13:31:00). Defaults to empty string if timestamp field does not exist.

!d convert a timestamp to a Python datetime object. This allows datetime specific format to be used, e.g. *{t_inactive!d:%H:%M:%S}*. However, note that width and alignment specifiers are not supported for datetime formatting. Defaults to datetime of epoch if timestamp field does not exist.

!F convert a duration in floating point seconds to Flux Standard Duration (FSD). string. Defaults to empty string if duration field does not exist.

!H convert a duration to hours:minutes:seconds form (e.g. *{runtime!H}*). Defaults to empty string if duration field does not exist.

Annotations can be retrieved via the *annotations* field name. Specific keys and sub-object keys can be retrieved separated by a period ("."). For example, if the scheduler has annotated the job with a reason pending status, it can be retrieved via "{annotations.sched.reason_pending}".

As a convenience, the field names *sched* and *user* can be used as substitutions for *annotations.sched* and *annotations.user*. For example, a reason pending status can be retrieved via "{sched.reason_pending}".

As a reminder to the reader, some shells may interpret special characters in Python's string format syntax. The format may need to be quoted or escaped to work under certain shells.

The field names that can be specified are:

id job ID

id.f58 job ID in RFC 19 F58 (base58) encoding

id.hex job ID in `0x` prefix hexadecimal representation

id.dothex job ID in dotted hexadecimal representation (`xx.xx.xx.xx`)

id.words job ID in mnemonic encoding

userid job submitter's userid

username job submitter's username

urgency job urgency

priority job priority

dependencies list of any currently outstanding job dependencies

status job status (PENDING, RUNNING, COMPLETED, FAILED, or CANCELED)

status_abbrev status but in a max 2 character abbreviation

name job name

ntasks job task count

nnodes job node count (if job ran / is running), empty string otherwise

ranks job ranks (if job ran / is running), empty string otherwise

nodelist job nodelist (if job ran / is running), empty string otherwise

state job state (DEPEND, SCHED, RUN, CLEANUP, INACTIVE)

state_single job state as a single character

result job result if job is inactive (COMPLETED, FAILED, CANCELED), empty string otherwise

result_abbrev result but in a max 2 character abbreviation

success True of False if job completed successfully, empty string otherwise

waitstatus The raw status of the job as returned by `waitpid(2)` if the job exited, otherwise an empty string. Note: *waitstatus* is the maximum wait status returned by all job shells in a job, which may not necessarily indicate the highest *task* wait status. (The job shell exits with the maximum task exit status, unless a task died due to a signal, in which case the shell exits with 128+signo)

returncode The job return code if the job has exited, or an empty string if the job is still active. The return code of a job is the highest job shell exit code, or negative signal number if the job shell was terminated by a signal. If the job was canceled before it started, then the returncode is set to the special value -128.

exception.occurred True of False if job had an exception, empty string otherwise

exception.severity If exception.occurred True, the highest severity, empty string otherwise

exception.type If exception.occurred True, the highest severity exception type, empty string otherwise

exception.note If exception.occurred True, the highest severity exception note, empty string otherwise

t_submit time job was submitted

t_depend time job entered depend state

t_run time job entered run state

t_cleanup time job entered cleanup state

t_inactive time job entered inactive state

runtime job runtime

expiration time at which job allocation was marked to expire

t_remaining If job is running, amount of time remaining before expiration

annotations annotations metadata, use "." to get specific keys

sched short hand for *annotations.sched*

user short hand for *annotations.user*

EXAMPLES

The default output of flux-jobs(1) will list the pending and running jobs of the current user. It is equivalent to:

```
$ flux jobs --filter=pending,running
```

To list all pending, running, and inactive jobs, of the current user, you can use *–filter* option or the *-a* option:

```
$ flux jobs -a

OR

$ flux jobs --filter=pending,running,inactive
```

To alter which user's jobs are listed, specify the user with *–user*:

```
$ flux jobs --user=flux
```

Jobs that have finished may be filtered further by specifying if they have completed, failed, or were canceled. For example, the following will list the jobs that have failed or were canceled:

```
$ flux jobs --filter=failed,canceled
```

The *–format* option can be used to alter the output format or output additional information. For example, the following would output all jobids for the user in decimal form, and output any annotations the scheduler attached to each job:

```
$ flux jobs -a --format="{id} {annotations.sched}"
```

The following would output the job id and exception information, so a user can learn why a job failed.

```
$ flux jobs --filter=failed --format="{id} {exception.type} {exception.note}"
```

RESOURCES

Github: http://github.com/flux-framework

1.1.12 flux-jobtap(1)

SYNOPSIS

flux jobtap *COMMAND* [*OPTIONS*] ARGS...

DESCRIPTION

The flux-jobtap(1) command is used to query, load, and remove *jobtap* plugins from the Flux job-manager module at runtime.

COMMANDS

list [*-a, –all*] Print the currently loaded list of plugins. Builtin plugins will only be displayed when the *–all* option is used. Plugins built in to the job manager have a leading `.` in the name, e.g. `.priority-default`.

load [*-r, –remove=NAME*] PLUGIN [KEY=VAL, KEY=VAL...] Load a new plugin into the job-manager, optionally removing plugin NAME first. With *–remove* NAME may be a glob(7) pattern match. Optional KEY=VAL occurring after PLUGIN will set config KEY to VAL for PLUGIN.

remove NAME Remove plugin NAME. NAME may be a glob(7) pattern in which case all matching, non-builtin plugins are removed. The special value *all* may be used to remove all loaded jobtap plugins. Builtin plugins (those starting with a leading `.`) must be removed explicitly or by preceding *NAME* with `.`, e.g. `.*`.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-jobtap-plugins(7)

1.1.13 flux-keygen(1)

SYNOPSIS

flux keygen *PATH*

DESCRIPTION

flux-keygen(1) generates a long-term CURVE certificate used to secure the overlay network of a Flux system instance.

The Flux overlay network implements cryptographic privacy and data integrity when data is sent over a network. Point to point ZeroMQ TCP connections are protected with the CURVE security mechanism built into ZeroMQ version 4, based on curve25519 and a CurveCP-like protocol.

All brokers participating in the system instance must use the same certificate. The certificate is part of the bootstrap configuration.

Flux instances that bootstrap with PMI do not require a configured certificate. In that case, each broker self-generates a unique certificate and the public keys are exchanged with PMI.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

ZAP: http://rfc.zeromq.org/spec:27

CurveZMQ: http://curvezmq.org/page:read-the-docs

ZMTP/3.0: http://rfc.zeromq.org/spec:23

Using ZeroMQ Security: http://hintjens.com/blog:48 and http://hintjens.com/blog:49

1.1.14 flux-kvs(1)

SYNOPSIS

flux kvs *COMMAND* [*OPTIONS*]

DESCRIPTION

The Flux key-value store (KVS) is a simple, distributed data storage service used a building block by other Flux components. flux-kvs(1) is a command line utility that operates on the KVS. It is a very thin layer on top of a C API.

The Flux KVS stores values under string keys. The keys are hierarchical, using "." as a path separator, analogous to "/" separated UNIX file paths. A single "." represents the root directory of the KVS.

The KVS is distributed among the broker ranks of a Flux instance. Rank 0 is the leader, and other ranks are caching followers. All writes are flushed to the leader during a commit operation. Data is stored in a hash tree such that every commit results in a new root hash. Each new root hash is multicast across the session. When followers update their root hash, they atomically update their view to match the leader. There may be a delay after a commit while old data is served on a follower that has not yet updated its root hash, thus the Flux KVS consistency model is "eventually consistent". Followers cache data temporally and fault in new data through their parent in the overlay network.

Different KVS namespaces can be created in which kvs values can be read from/written to. By default, all KVS operations operate on the default KVS namespace "primary". An alternate namespace can be specified in most kvs commands via the *–namespace* option, or by setting the namespace in the environment variable FLUX_KVS_NAMESPACE.

flux-kvs(1) runs a KVS *COMMAND*. The possible commands and their arguments are described below.

COMMANDS

namespace create [-o owner] ***name*** **[*name* ...]** Create a new kvs namespace. User may specify an alternate userid of a user that owns the namespace via *-o*. Specifying an alternate owner would allow a non-instance owner to read/write to a namespace.

namespace remove ***name*** **[*name*...]** Remove a kvs namespace.

namespace list List all current namespaces and info on each namespace.

get [-N ns] [-r|-t] [-a treeobj] [-l] [-W] [-w] [-u] [-A] [-f] [-c count] ***key*** **[*key* ...]** Retrieve the value stored under *key*. If nothing has been stored under *key*, display an error message. Specify an alternate namespace to retrieve *key* from via *-N*. If no options, value is displayed with a newline appended (if value length is nonzero). If *-l*, a *key=* prefix is added. If *-r*, value is displayed without a newline. If *-t*, the RFC 11 object is displayed. *-a treeobj* causes the lookup to be relative to an RFC 11 snapshot reference. If *-W* is specified and a key does not exist, wait until the key has been created. If *-w*, after the initial value, display the new value each time the key is written to until interrupted, or if *-c count* is specified, until *count* values have been displayed. If *-u* is specified, only writes that change the key value will be displayed. If *-A* is specified, only display appends that occur on a key. By default, only a direct write to a key is monitored, which may miss several unique situations, such as the replacement of an entire parent directory. The *-f* option can be specified to monitor for many of these special situations.

put [-N ns] [-O|-s] [-r|-t] [-n] [-A] ***key=value*** **[*key=value* ...]** Store *value* under *key* and commit it. Specify an alternate namespace to commit value(s) via *-N*. If it already has a value, overwrite it. If no options, value is stored directly. If *-r* or *-t*, the value may optionally be read from standard input if specified as "-". If *-r*, the value may include embedded NULL bytes. If *-t*, value is stored as a RFC 11 object. *-n* prevents the commit from being merged with with other contemporaneous commits. *-A* appends the value to a key instead of overwriting the

value. Append is incompatible with the -j option. After a successful put, *-O* or *-s* can be specified to output the RFC11 treeobj or root sequence number of the root containing the put(s).

ls [-N ns] [-R] [-d] [-F] [-w COLS] [-1] [*key* ...] Display directory referred to by *key*, or "." (root) if unspecified. Specify an alternate namespace to display via *-N*. Remaining options are roughly equivalent to a subset of ls(1) options. *-R* lists directory recursively. *-d* displays directory not its contents. *-F* classifies files with one character suffix (. is directory, @ is symlink). *-w COLS* sets the terminal width in characters. *-1* causes output to be displayed in one column.

dir [-N ns] [-R] [-d] [-w COLS] [-a treeobj] [*key*] Display all keys and their values under the directory *key*. Specify an alternate namespace to display via *-N*. If *key* does not exist or is not a directory, display an error message. If *key* is not provided, "." (root of the namespace) is assumed. If *-R* is specified, recursively display keys under subdirectories. If *-d* is specified, do not output key values. Output is truncated to fit the terminal width. *-w COLS* sets the terminal width (0=unlimited). *-a treeobj* causes the lookup to be relative to an RFC 11 snapshot reference.

unlink [-N ns] [-O|-s] [-R] [-f] *key* [*key* ...] Remove *key* from the KVS and commit the change. Specify an alternate namespace to commit to via *-N*. If *key* represents a directory, specify *-R* to remove all keys underneath it. If *-f* is specified, ignore nonexistent files. After a successful unlink, *-O* or *-s* can be specified to output the RFC11 treeobj or root sequence number of the root containing the unlink(s).

link [-N ns] [-T ns] [-O|-s] *target linkname* Create a new name for *target*, similar to a symbolic link, and commit the change. *target* does not have to exist. If *linkname* exists, it is overwritten. Specify an alternate namespace to commit linkname to via *-N*. Specify the target's namespace via *-T*. After a successfully created link, *-O* or *-s* can be specified to output the RFC11 treeobj or root sequence number of the root containing the link.

readlink [-N ns] [-a treeobj] [-o | -k] *key* [*key* ...] Retrieve the key a link refers to rather than its value, as would be returned by **get**. Specify an alternate namespace to retrieve from via *-N*. *-a treeobj* causes the lookup to be relative to an RFC 11 snapshot reference. If the link points to a namespace, the namespace and key will be output in the format <*namespace*>*::*<*key*>. The *-o* can be used to only output namespaces and the *-k* can be used to only output keys.

mkdir [-N ns] [-O|-s] *key* [*key* ...] Create an empty directory and commit the change. If *key* exists, it is overwritten. Specify an alternate namespace to commit to via *-N*. After a successful mkdir, *-O* or *-s* can be specified to output the RFC11 treeobj or root sequence number of the root containing the new directory.

copy [-S src-ns] [-D dst-ns] *source destination* Copy *source* key to *destination* key. Optionally, specify a source and/or destination namespace for the *source* and/or *destination* respectively. If a directory is copied, a new reference is created; it is unnecessary for **copy** to recurse into *source*.

move [-S src-ns] [-D dst-ns] *source destination* Like **copy**, but *source* is unlinked after the copy.

dropcache [–all] Tell the local KVS to drop any cache it is holding. If *–all* is specified, send an event across the Flux instance instructing all KVS modules to drop their caches.

version [-N ns] Display the current KVS version, an integer value. The version starts at zero and is incremented on each KVS commit. Note that some commits may be aggregated for performance and the version will be incremented once for the aggregation, so it cannot be used as a direct count of commit requests. Specify an alternate namespace to retrieve the version from via *-N*.

wait [-N ns] *version* Block until the KVS version reaches *version* or greater. A simple form of synchronization between peers is: node A puts a value, commits it, reads version, sends version to node B. Node B waits for version, gets value.

getroot [-N ns] [-s | -o] Retrieve the current KVS root, displaying it as an RFC 11 dirref object. Specify an alternate namespace to retrieve from via *-N*. If *-o* is specified, display the namespace owner. If *-s* is specified, display the root sequence number.

eventlog get [-N ns] [-W] [-w] [-c count] [-u] *key* Display the contents of an RFC 18 KVS eventlog referred to by *key*. If *-u* is specified, display the log in raw form. If *-W* is specified and the eventlog does not exist, wait until

it has been created. If *-w* is specified, after the existing contents have been displayed, the eventlog is monitored and updates are displayed as they are committed. This runs until the program is interrupted or an error occurs, unless the number of events is limited with the *-c* option. Specify an alternate namespace to display from via *-N*.

eventlog append [-N ns] [-t SECONDS] *key name* [*context ...*] Append an event to an RFC 18 KVS eventlog referred to by *key*. The event *name* and optional *context* are specified on the command line. The timestamp may optionally be specified with *-t* as decimal seconds since the UNIX epoch (UTC), otherwise the current wall clock is used. Specify an alternate namespace to append to via *-N*.

eventlog wait-event [-N ns] [-t SECONDS] [-u] [-W] [-q] [-v] *key event* Wait for a specific *event* to occur in an RFC 18 KVS eventlog referred to by *key*. If *-t* is specified, timeout after *SECONDS* if the event has not occurred. If *-u* is specified, display the log in raw form. If *-W* is specified and the eventlog does not exist, wait until it has been created. If *-q* is specified, not output the matched event. If *-v* is specified, output all events prior to the matched event. This runs until the program is interrupted, the event occurs, or a timeout occurs if *-t* is specified. Specify an alternate namespace to display from via *-N*.

RESOURCES

Github: http://github.com/flux-framework

1.1.15 flux-logger(1)

SYNOPSIS

flux logger [*–severity SEVERITY*] [*–appname NAME*] *message ...*

DESCRIPTION

flux-logger(1) appends Flux log entries to the local Flux broker's circular buffer.

Log entries are associated with a syslog(3) style severity. Valid severity names are *emerg*, *alert*, *crit*, *err*, *warning*, *notice*, *info*, *debug*.

Log entries may also have a user-defined application name. This is different than the syslog *facility*, which is always set to LOG_USER in Flux log messages.

The wall clock time (UTC) and the broker rank are added to the log message when it is created.

OPTIONS

-s, –severity=*SEVERITY* Specify the log message severity. The default severity is *info*.

-n, –appname=*NAME* Specify a user-defined application name to associate with the log message. The default appname is *logger*.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-dmesg(1), flux_log(3), syslog(3)

1.1.16 flux-mini(1)

SYNOPSIS

flux mini submit [OPTIONS] [*–ntasks=N*] COMMAND...

flux mini bulksubmit [OPTIONS] [*–ntasks=N*] COMMAND...

flux mini run [OPTIONS] [*–ntasks=N*] COMMAND...

flux mini batch [OPTIONS] *–nslots=N* SCRIPT...

flux mini alloc [OPTIONS] *–nslots=N* [COMMAND...]

DESCRIPTION

flux-mini(1) submits jobs to run under Flux. In the case of **submit** or **run** the job consists of *N* copies of COMMAND launched together as a parallel job, while **batch** and **alloc** submit a script or launch a command as the initial program of a new Flux instance.

If *–ntasks* is unspecified, a value of *N=1* is assumed. Commands that take *–nslots* have no default and require that *–nslots* be explicitly specified.

The **submit** and **batch** commands enqueue the job and print its numerical Job ID on standard output.

The **run** and **alloc** commands do the same interactively, blocking until the job has completed.

The **bulksubmit** command enqueues one job each for a set of inputs read on either stdin, or given on the command line. The inputs are optionally substituted in `COMMAND` and/or many submission options. See more in the *BULKSUBMIT* section below.

For **flux-mini batch**, the SCRIPT given on the command line is assumed to be a file name, unless the *–wrap* option used, and the script file is read and submitted along with the job. If no SCRIPT is provided, then one will be read from *stdin*.

flux-mini alloc works similarly to **batch**, but instead blocks until the job has started and interactively attaches to the new Flux instance. By default, a new shell is spawned as the initial program of the instance, but this may be overridden by supplying COMMAND on the command line.

The intent is for the "mini" commands to remain simple with stable interfaces over time, making them suitable for use in scripts. For advanced usage, see flux-run(1) and flux-submit(1).

The available OPTIONS are detailed below.

JOB PARAMETERS

These commands accept only the simplest parameters for expressing the size of the parallel program and the geometry of its task slots:

The **run** and **submit** commands take the following options to specify the size of the job request:

-n, –ntasks=N Set the number of tasks to launch (default 1).

-c, –cores-per-task=N Set the number of cores to assign to each task (default 1).

-g, –gpus-per-task=N Set the number of GPU devices to assign to each task (default none).

The **batch** and **alloc** commands do not launch tasks directly, and therefore job parameters are specified in terms of resource slot size and number of slots. A resource slot can be thought of as the minimal resources required for a virtual task. The default slot size is 1 core.

-n, –nslots=N Set the number of slots requested. This parameter is required.

-c, –cores-per-slot=N Set the number of cores to assign to each slot (default 1).

-g, –gpus-per-slot=N Set the number of GPU devices to assign to each slot (default none).

The **run**, **submit**, **batch**, and **alloc** commands also take following additional job parameters:

-N, –nodes=N Set the number of nodes to assign to the job. Tasks will be distributed evenly across the allocated nodes. It is an error to request more nodes than there are tasks. If unspecified, the number of nodes will be chosen by the scheduler.

-t, –time-limit=FSD Set a time limit for the job in Flux standard duration (RFC 23). FSD is a floating point number with a single character units suffix ("s", "m", "h", or "d"). If unspecified, the job is subject to the system default time limit.

STANDARD I/O

By default, task stdout and stderr streams are redirected to the KVS, where they may be accessed with the `flux job attach` command.

In addition, `flux-mini run` processes standard I/O in real time, emitting the job's I/O to its stdout and stderr.

–output=TEMPLATE Specify the filename *TEMPLATE* for stdout redirection, bypassing the KVS. *TEMPLATE* may be a mustache template which supports the tags *{{id}}* and *{{jobid}}* which expand to the current jobid in the F58 encoding. If needed, an alternate encoding can be selected by using a subkey with the name of the desired encoding, e.g. *{{id.dec}}*. Supported encodings include *f58* (the default), *dec*, *hex*, *dothex*, and *words*. For **flux mini batch** the default *TEMPLATE* is *flux-{{id}}.out*. To force output to KVS so it is available with `flux job attach`, set *TEMPLATE* to *none* or *kvs*.

–error=TEMPLATE Redirect stderr to the specified filename *TEMPLATE*, bypassing the KVS. *TEMPLATE* is expanded as described above.

-l, –label-io Add task rank prefixes to each line of output.

DEPENDENCIES

Note: Flux supports a simple but powerful job dependency specification in jobspec. See Flux Framework RFC 26 for more detailed information about the generic dependency specification.

Dependencies may be specified on the `flux mini` command line using the following option

–dependency=URI Specify a dependency of the submitted job using RFC 26 dependency URI format. The URI format is **SCHEME:VALUE[?key=val[&key=val...]]**. The URI will be converted into RFC 26 JSON object form and appended to the jobspec `attributes.system.dependencies` array. If the current Flux instance does not support dependency scheme *SCHEME*, then the submitted job will be rejected with an error message indicating this fact.

The `--dependency` option may be specified multiple times. Each use appends a new dependency object to the `attributes.system.dependencies` array.

The following dependency schemes are built-in:

Note: The `after*` dependency schemes listed below all require that the target JOBID be currently active. If the target JOBID has become inactive by the time the dependent job is submitted, then the submission will be rejected with an error that the dependency job cannot be found.

---

after:JOBID This dependency is satisfied after JOBID starts.

afterany:JOBID This dependency is satisfied after JOBID enters the INACTIVE state, regardless of the result

afterok:JOBID This dependency is satisfied after JOBID enters the INACTIVE state with a successful result.

afternotok:JOBID This dependency is satisfied after JOBID enters the INACTIVE state with an unsuccessful result.

begin-time:TIMESTAMP This dependency is satisfied after TIMESTAMP, which is specified in floating point seconds since the UNIX epoch. See the `flux-mini --begin-time` option below for a more user-friendly interface to the `begin-time` dependency.

In any of the above `after*` cases, if it is determined that the dependency cannot be satisfied (e.g. a job fails due to an exception with afterok), then a fatal exception of type=dependency is raised on the current job.

ENVIRONMENT

By default, `flux-mini` duplicates the current environment when submitting jobs. However, a set of environment manipulation options are provided to give fine control over the requested environment submitted with the job.

–env=RULE Control how environment variables are exported with *RULE*. See *ENV RULE SYNTAX* section below for more information. Rules are applied in the order in which they are used on the command line. This option may be specified multiple times.

–env-remove=PATTERN Remove all environment variables matching *PATTERN* from the current generated environment. If *PATTERN* starts with a / character, then it is considered a regex(7), otherwise *PATTERN* is treated as a shell glob(7). This option is equivalent to `--env=-PATTERN` and may be used multiple times.

–env-file=FILE Read a set of environment *RULES* from a *FILE*. This option is equivalent to `--env=^FILE` and may be used multiple times.

ENV RULES

The `--env*` options of `flux-mini` allow control of the environment exported to jobs via a set of *RULE* expressions. The currently supported rules are

- If a rule begins with -, then the rest of the rule is a pattern which removes matching environment variables. If the pattern starts with /, it is a regex(7), optionally ending with /, otherwise the pattern is considered a shell glob(7) expression.

  Examples: `-*` or `-/.*/` filter all environment variables creating an empty environment.

- If a rule begins with ^ then the rest of the rule is a filename from which to read more rules, one per line. The ~ character is expanded to the user's home directory.

  Examples: `~/envfile` reads rules from file `$HOME/envfile`

- If a rule is of the form `VAR=VAL`, the variable `VAR` is set to `VAL`. Before being set, however, `VAL` will undergo simple variable substitution using the Python `string.Template` class. This simple substitution supports the following syntax:

  - `$$` is an escape; it is replaced with `$`
  - `$var` will substitute `var` from the current environment, falling back to the process environment. An error will be thrown if environment variable `var` is not set.
  - `${var}` is equivalent to `$var`
  - Advanced parameter substitution is not allowed, e.g. `${var:-foo}` will raise an error.

  Examples: `PATH=/bin`, `PATH=$PATH:/bin`, `FOO=${BAR}something`

- Otherwise, the rule is considered a pattern from which to match variables from the process environment if they do not exist in the generated environment. E.g. `PATH` will export `PATH` from the current environment (if it has not already been set in the generated environment), and `OMP*` would copy all environment variables that start with `OMP` and are not already set in the generated environment. It is important to note that if the pattern does not match any variables, then the rule is a no-op, i.e. an error is *not* generated.

  Examples: `PATH`, `FLUX_*_PATH`, `/^OMP.*/`

Since `flux-mini` always starts with a copy of the current environment, the default implicit rule is `*` (or `--env=*`). To start with an empty environment instead, the `-*` rule or `--env-remove=*` option should be used. For example, the following will only export the current `PATH` to a job:

```
flux mini run --env-remove=* --env=PATH ...
```

Since variables can be expanded from the currently built environment, and `--env` options are applied in the order they are used, variables can be composed on the command line by multiple invocations of `--env`, e.g.:

```
flux mini run --env-remove=* \
              --env=PATH=/bin --env='PATH=$PATH:/usr/bin' ...
```

Note that care must be taken to quote arguments so that `$PATH` is not expanded by the shell.

This works particularly well when specifying rules in a file:

```
-*
OMP*
FOO=bar
BAR=${FOO}/baz
```

The above file would first clear the environment, then copy all variables starting with `OMP` from the current environment, set `FOO=bar`, and then set `BAR=bar/baz`.

EXIT STATUS

The job exit status, normally the largest task exit status, is stored in the KVS. If one or more tasks are terminated with a signal, the job exit status is 128+signo.

The `flux-job attach` command exits with the job exit status.

In addition, `flux-mini run` runs until the job completes and exits with the job exit status.

OTHER OPTIONS

–urgency=N Specify job urgency, which affects queue order. Numerically higher urgency jobs are considered by the scheduler first. Guests may submit jobs with urgency in the range of 0 to 16, while instance owners may submit jobs with urgency in the range of 0 to 31 (default 16). In addition to numerical values, the special names `hold` (0), `default` (16), and `expedite` (31) are also accepted.

-v, –verbose *(run,alloc,submit,bulksubmit)* Increase verbosity on stderr. For example, currently `flux mini run -v` displays jobid, `-vv` displays job events, and `-vvv` displays exec events. `flux mini alloc -v` forces the command to print the submitted jobid on stderr. The specific output may change in the future.

-o, –setopt=KEY[=VAL] Set shell option. Keys may include periods to denote hierarchy. VAL is optional and may be valid JSON (bare values, objects, or arrays), otherwise VAL is interpreted as a string. If VAL is not set, then the default value is 1. See SHELL OPTIONS below.

–setattr=KEY=VAL Set jobspec attribute. Keys may include periods to denote hierarchy. VAL may be valid JSON (bare values, objects, or arrays), otherwise VAL is interpreted as a string. If KEY starts with a ^ character, then VAL is interpreted as a file, which must be valid JSON, to use as the attribute value.

–begin-time=DATETIME Convenience option for setting a `begin-time` dependency for a job. The job is guaranteed to start after the specified date and time. If *DATETIME* begins with a + character, then the remainder is considered to be an offset in Flux standard duration (RFC 23), otherwise, any datetime expression accepted by the Python parsedatetime module is accepted, e.g. `2021-06-21 8am`, `in an hour`, `tomorrow morning`, etc.

–dry-run Don't actually submit job. Just emit jobspec on stdout and exit for `run`, `submit`, `alloc`, and `batch`. For `bulksubmit`, emit a line of output including relevant options for each job which would have been submitted,

–debug Enable job debug events, primarily for debugging Flux itself. The specific effects of this option may change in the future.

-B, –broker-opts=OPT *(batch only)* For batch jobs, pass specified options to the Flux brokers of the new instance. This option may be specified multiple times.

–wrap *(batch only)* The `--wrap` option wraps the specified COMMAND and ARGS in a shell script, by prefixing with `#!/bin/sh`. If no COMMAND is present, then a SCRIPT is read on stdin and wrapped in a /bin/sh script.

–cc=IDSET *(submit,bulksubmit)* Replicate the job for each `id` in `IDSET`. `FLUX_JOB_CC=id` will be set in the environment of each submitted job to allow the job to alter its execution based on the submission index. (e.g. for reading from a different input file). When using `--cc`, the substitution string `{cc}` may be used in options and commands and will be replaced by the current `id`.

–bcc=IDSET *(submit,bulksubmit)* Identical to `--cc`, but do not set `FLUX_JOB_CC` in each job. All jobs will be identical copies. As with `--cc`, `{cc}` in option arguments and commands will be replaced with the current `id`.

–log=FILE *(submit,bulksubmit)* Log `flux-mini` output and stderr to `FILE` instead of the terminal. If a replacement (e.g. `{}` or `{cc}`) appears in `FILE`, then one or more output files may be opened. For example, to save all submitted jobids into separate files, use:

```
flux mini submit --cc=1-4 --log=job{cc}.id hostname
```

–log-stderr=FILE *(submit,bulksubmit)* Separate stderr into `FILE` instead of sending it to the terminal or a `FILE` specified by `--log`.

–wait *(submit,bulksubmit)* Wait on completion of all jobs before exiting.

–watch *(submit,bulksubmit)* Display output from all jobs. Implies `--wait`.

–progress *(submit,bulksubmit)* With `--wait`, display a progress bar showing the progress of job completion. Without `--wait`, the progress bar will show progress of job submission.

–jps *(submit,bulksubmit)* With `--progress`, display throughput statistics (jobs/s) in the progress bar.

–define=NAME=CODE *(bulksubmit)* Define a named method that will be made available as an attribute during command and option replacement. The string being processed is available as `x`. For example:

```
$ seq 1 8 | flux mini bulksubmit --define=pow="2**int(x)" -n {.pow} ...
```

–shuffle *(bulksubmit)* Shuffle the list of commands before submission.

–sep=STRING *(bulksubmit)* Change the separator for file input. The default is to separate files (including stdin) by newline. To separate by consecutive whitespace, specify `--sep=none`.

BULKSUBMIT

The `bulksubmit` utility allows rapid bulk submission of jobs using an interface similar to GNU parallel or `xargs`. The command takes inputs on stdin or the command line (separated by `:::`), and submits the supplied command template and options as one job per input combination.

The replacement is done using Python's `string.format()`, which is supplied a list of inputs on each iteration. Therefore, in the common case of a single input list, `{}` will work as the substitution string, e.g.:

```
$ seq 1 4 | flux mini bulksubmit echo {}
flux-mini: submit echo 1
flux-mini: submit echo 2
flux-mini: submit echo 3
flux-mini: submit echo 4
```

With `--dry-run` `bulksubmit` will print the args and command which would have been submitted, but will not perform any job submission.

The `bulksubmit` command can also take input lists on the command line. The inputs are separated from each other and the command with the special delimiter `:::`:

```
$ flux mini bulksubmit echo {} ::: 1 2 3 4
flux-mini: submit echo 1
flux-mini: submit echo 2
flux-mini: submit echo 3
flux-mini: submit echo 4
```

Multiple inputs are combined, in which case each input is passed as a positional parameter to the underlying `format()`, so should be accessed by index:

```
$ flux mini bulksubmit --dry-run echo {1} {0} ::: 1 2 ::: 3 4
flux-mini: submit echo 3 1
flux-mini: submit echo 4 1
flux-mini: submit echo 3 2
flux-mini: submit echo 4 2
```

If the generation of all combinations of an input list with other inputs is not desired, the special input delimited `:::+` may be used to "link" the input, so that only one argument from this source will be used per other input, e.g.:

```
$ flux mini bulksubmit --dry-run echo {0} {1} ::: 1 2 :::+ 3 4
flux-mini: submit 1 3
flux-mini: submit 2 4
```

The linked input will be cycled through if it is shorter than other inputs.

An input list can be read from a file with `::::`:

```
$ seq 0 3 >inputs
$ flux mini bulksubmit --dry-run :::: inputs
flux-mini: submit 0
flux-mini: submit 1
flux-mini: submit 2
flux-mini: submit 3
```

If the filename is `-` then `stdin` will be used. This is useful for including `stdin` when reading other inputs.

The delimiter `::::+` indicates that the next file is to be linked to the inputs instead of combined with them, as with `:::+`.

There are several predefined attributes for input substitution. These include:

- {.%} returns the input string with any extension removed.
- {./} returns the basename of the input string.
- {./%} returns the basename of the input string with any extension removed.
- {.//} returns the dirname of the input string
- {seq} returns the input sequence number (0 origin)
- {seq1} returns the input sequence number (1 origin)
- {cc} returns the current id from use of --cc or --bcc. Note that replacement of {cc} is done in a second pass, since the --cc option argument may itself be replaced in the first substitution pass. If --cc/bcc were not used, then {cc} is replaced with an empty string. This is the only substitution supported with flux-mini submit.

Note that besides {seq}, {seq1}, and {cc} these attributes can also take the input index, e.g. {0.%} or {1.//}, when multiple inputs are used.

Additional attributes may be defined with the --define option, e.g.:

```
$ flux mini bulksubmit --dry-run --define=p2='2**int(x)' -n {.p2} hostname \
   ::: $(seq 0 4)
flux-mini: submit -n1 hostname
flux-mini: submit -n2 hostname
flux-mini: submit -n4 hostname
flux-mini: submit -n8 hostname
flux-mini: submit -n16 hostname
```

The input string being indexed is passed to defined attributes via the local x as seen above.

SHELL OPTIONS

These options are provided by built-in shell plugins that may be overridden in some cases:

mpi=spectrum Load the MPI personality plugin for IBM Spectrum MPI. All other MPI plugins are loaded by default.

cpu-affinity=per-task Tasks are distributed across the assigned resources.

cpu-affinity=off Disable task affinity plugin.

gpu-affinity=per-task GPU devices are distributed evenly among local tasks. Otherwise, GPU device affinity is to the job.

gpu-affinity=off Disable GPU affinity for this job.

verbose Increase verbosity of the job shell log.

pmi.kvs=native Use the native Flux KVS instead of the PMI plugin's built-in key exchange algorithm.

pmi.exchange.k=N Configure the PMI plugin's built-in key exchange algorithm to use a virtual tree fanout of N for key gather/broadcast. The default is 2.

RESOURCES

Github: http://github.com/flux-framework

1.1.17 flux-module(1)

SYNOPSIS

flux module *COMMAND* [*OPTIONS*]

DESCRIPTION

flux-module(1) manages dynamically loadable Flux modules. It can load/remove/list modules for the flux-broker(1), and for other Flux services that support dynamic module extensions.

COMMANDS

info [*name*] Display information about module *name*. If *name* includes a slash / character, it is interpreted as a file path, and the module name is then determined by reading the **mod_name** symbol. Otherwise, FLUX_MODULE_PATH is searched for a module with **mod_name** equal to *name*.

load *name* [*module-arguments ...*] Load module *name*, interpreted as described above. The service that will load the module is inferred from the module name. When the load command completes successfully, the new module is ready to accept messages on all targeted ranks.

remove [–force] *name* Remove module *name*. The service that will unload the module is inferred from the name specified on the command line. If *-f, –force* is used, then do not error if module *name* is not loaded.

reload [–force] *name* [*module-arguments ...*] Reload module *name*. This is equivalent to running *flux module remove* followed by *flux module load*. It is a fatal error if module *name* is not loaded during removal unless the `-f, --force` option is specified.

list [*service*] List modules loaded by *service*, or by flux-broker(1) if *service* is unspecified.

stats [*OPTIONS*] [*name*] Request statistics from module *name*. A JSON object containing a set of counters for each type of Flux message is returned by default, however the object may be customized on a module basis.

debug [*OPTIONS*] [*name*] Manipulate debug flags in module *name*. The interpretation of debug flag bits is private to the module and its test drivers.

STATS OPTIONS

-p, –parse=*OBJNAME* OBJNAME is a period delimited list of field names that should be walked to obtain a specific value or object in the returned JSON.

-t, –type=*int|double* Force the returned value to be converted to int or double.

-s, –scale=*N* Multiply the returned (int or double) value by the specified floating point value.

-R, –rusage Return a JSON object representing an *rusage* structure returned by getrusage(2).

-c, –clear Send a request message to clear statistics in the target module.

-C, –clear-all Broadcast an event message to clear statistics in the target module on all ranks.

DEBUG OPTIONS

-c, –clear Set debug flags to zero.

-S, –set=*MASK* Set debug flags to MASK. The value may be prefixed with 0x to indicate hexadecimal or 0 to indicate octal, otherwise the value is interpreted as decimal.

-c, –clearbit=*MASK* Clear the debug bits specified in MASK without disturbing other bits. The value is interpreted as above.

-s, –setbit=*MASK* Set the debug bits specified in MASK without disturbing other bits. The value is interpreted as above.

LIST OUTPUT

The *list* command displays one line for each unique (as determined by SHA1 hash) module loaded by a service.

Module The value of the **mod_name** symbol for this module.

Size The size in bytes of the module .so file.

Digest The last 7 characters of the SHA1 digest of the contents of the module .so file.

Idle Idle times are defined for flux-broker(1) modules as the number of seconds since the module last sent a request or response message. The idle time may be defined differently for other services, or have no meaning.

MODULE SYMBOLS

All Flux modules define the following global symbols:

const char *mod_name; A null-terminated string defining the module name. Module names are words delimited by periods, with the service that will load the module indicated by the words that prefix the final one. If there is no prefix, the module is loaded by flux-broker(1).

int mod_main (void *context, int argc, char **argv); An entry function.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

syslog(3)

1.1.18 flux-ping(1)

SYNOPSIS

flux ping [*OPTIONS*] target

DESCRIPTION

flux-ping(1) measures round-trip latency to a Flux service implementing the "ping" method in a manner analogous to ping(8). The ping response is essentially an echo of the request, with the route taken to the service added by the service. This route is displayed in the output and can give insight into how various addresses are routed.

target may be the name of a Flux service, e.g. "kvs". flux-ping(1) will send a request to "kvs.ping". As a shorthand, *target* can include a rank prefix delimited by an exclamation point. "flux ping 4!kvs" is equivalent to "flux ping –rank 4 kvs" (see –rank option below). Don't forget to quote the exclamation point if it is interpreted by your shell.

As a shorthand, *target* may also simply be a rank by itself indicating that the broker on that rank or ranks, rather than a Flux service, is to be pinged. "flux ping 1" is equivalent to "flux ping –rank 1 broker".

OPTIONS

-r, –rank=*N* Find target on a specific broker rank. Special case strings "*any*" and "*upstream*" available to ping FLUX_NODEID_ANY and FLUX_NODEID_UPSTREAM respectively. Default: send to "*any*".

-p, –pad=*N* Include in the payload a string of length *N* bytes. The payload will be echoed back in the response. This option can be used to explore the effect of message size on latency. Default: no padding.

-i, –interval=*N* Specify the delay, in seconds, between successive requests. A value of zero is valid and indicates that there should be no delay. Requests are sent without waiting for responses. Default: 1.0 seconds.

-c, –count=*N* Specify the number of requests to send, and terminate the command once responses have been received for all the requests. Default: unlimited.

-b, –batch Begin processing responses after all requests are sent. Requires –count.

-u, –userid Include userid and rolemask of original request, which are echoed back in ping response, in ping output.

EXAMPLES

One can ping a service by name, e.g.

```
$ flux ping kvs
kvs.ping pad=0 seq=0 time=0.774 ms (0EB02!A3368!0!382A6)
kvs.ping pad=0 seq=1 time=0.686 ms (0EB02!A3368!0!382A6)
...
```

This tells you that the local "kvs" service is alive and the round-trip latency is a bit over half a millisecond. The route hops are:

```
0EB02: UUID of the ping command
A3368: UUID of the API module
0:     rank of the local broker
382A6: UUID of the KVS module.
```

RESOURCES

Github: http://github.com/flux-framework

1.1.19 flux-proxy(1)

SYNOPSIS

flux proxy [*OPTIONS*] URI [command [args...]]

DESCRIPTION

flux proxy connects to the Flux instance identified by *URI*, then spawns a shell with FLUX_URI pointing to a local:// socket managed by the proxy program. As long as the shell is running, the proxy program routes messages between the instance and the local:// socket. Once the shell terminates, the proxy program terminates and removes the socket.

The purpose of **flux proxy** is to allow a connection to be reused, for example where connection establishment has high latency or requires authentication.

OPTIONS

-f, –force Allow the proxy command to connect to a broker running a different version of Flux with a warning message instead of a fatal error.

EXAMPLES

Connect to a job running on the localhost which has a FLUX_URI of `local:///tmp/flux-123456-abcdef/0/local` and spawn an interactive shell:

```
$ flux proxy local:///tmp/flux-123456-abcdef/0/local
```

Connect to the same job remotely on host foo.com:

```
$ flux proxy ssh://foo.com/tmp/flux-123456-abcdef/0/local
```

RESOURCES

Github: http://github.com/flux-framework

1.1.20 flux-start(1)

SYNOPSIS

flux start [*OPTIONS*] [initial-program [args...]]

DESCRIPTION

flux-start(1) launches a new Flux instance. By default, flux-start execs a single flux-broker(1) directly, which will attempt to use PMI to fetch job information and bootstrap a flux instance.

If a size is specified via *–test-size*, an instance of that size is to be started on the local host with flux-start as the parent.

A failure of the initial program (such as non-zero exit code) causes flux-start to exit with a non-zero exit code.

OPTIONS

-o, –broker-opts=*option_string* Add options to the message broker daemon, separated by commas.

-v, –verbose=*[LEVEL]* This option may be specified multiple times, or with a value, to set a verbosity level. See VERBOSITY LEVELS below.

-X, –noexec Don't execute anything. This option is most useful with -v.

–caliper-profile=*PROFILE* Run brokers with Caliper profiling enabled, using a Caliper configuration profile named *PROFILE*. Requires a version of Flux built with –enable-caliper. Unless CALI_LOG_VERBOSITY is already set in the environment, it will default to 0 for all brokers.

–rundir=*DIR* (only with *–test-size*) Set the directory that will be used as the rundir directory for the instance. If the directory does not exist then it will be created during instance startup. If a DIR is not set with this option, a unique temporary directory will be created. Unless DIR was pre-existing, it will be removed when the instance is destroyed.

–wrap=*ARGS,...* Wrap broker execution in a comma-separated list of arguments. This is useful for running flux-broker directly under debuggers or valgrind.

-s, –test-size=*N* Launch an instance of size *N* on the local host.

–test-hosts=*HOSTLIST* Set FLUX_FAKE_HOSTNAME in the environment of each broker so that the broker can bootstrap from a config file instead of PMI. HOSTLIST is assumed to be in rank order. The broker will use the fake hostname to find its entry in the configured bootstrap host array.

–test-exit-timeout=*FSD* After a broker exits, kill the other brokers after a timeout (default 20s).

–test-exit-mode=*MODE* Set the mode for the exit timeout. If set to `leader`, the exit timeout is only triggered upon exit of the leader broker, and the flux-start exit code is that of the leader broker. If set to `any`, the exit timeout is triggered upon exit of any broker, and the flux-start exit code is the highest exit code of all brokers. Default: `any`.

–test-start-mode=*MODE* Set the start mode. If set to `all`, all brokers are started immediately. If set to `leader`, only the leader is started. Hint: in `leader` mode, use `--setattr=broker.quorum=0` to let the initial program start before the other brokers are online. Default: `all`.

–test-rundir=*PATH* Set the directory to be used as the broker rundir instead of creating a temporary one. The directory must exist, and is not cleaned up unless `--test-rundir-cleanup` is also specified.

–test-rundir-cleanup Recursively remove the directory specified with `--test-rundir` upon completion of flux-start.

–test-pmi-clique=*MODE* Set the pmi clique mode, which determines how `PMI_process_mapping` is set in the PMI server used to bootstrap the brokers. If `none`, the mapping is not created. If `single`, all brokers are placed in one clique. Default: `single`.

VERBOSITY LEVELS

level 1 and above Display commands before executing them.

level 2 and above Trace PMI server requests (test mode only).

EXAMPLES

Launch an 8-way local Flux instance with an interactive shell as the initial program and all logs output to stderr:

```
flux start -s8 -o,--setattr=log-stderr-level=7
```

Launch an 8-way Flux instance within a slurm job, with an interactive shell as the initial program:

```
srun --pty -N8 flux start
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-broker(1)

1.1.21 flux-shell(1)

SYNOPSIS

flux-shell [*OPTIONS*] *JOBID*

DESCRIPTION

flux-shell(1), the Flux job shell, is the component of Flux which manages the startup and execution of user jobs. flux-shell(1) runs as the job user, reads the jobspec and assigned resource set R for the job from the KVS, and using this data determines what local job tasks to execute. While job tasks are running, the job shell acts as the interface between the Flux instance and the job by handling standard I/O, signals, and finally collecting the exit status of tasks as they complete.

The design of the Flux job shell allows customization through a set of builtin and runtime loadable shell plugins. These plugins are used to handle standard I/O redirection, PMI, CPU and GPU affinity, debugger support and more. Details of the flux-shell(1) plugin capabilities and design can be found in the PLUGINS section below.

flux-shell(1) also supports configuration via a Lua-based configuration file, called the shell `initrc`, from which shell plugins may be loaded or shell options and data examined or set. The flux-shell(1) initrc may even extend the shell itself via simple shell plugins developed directly in Lua. See the SHELL INITRC section below for details of the `initrc` format and features.

Most Flux users will interact with flux-shell(1) indirectly through the execution of Flux jobs, however flux-shell(1) accepts the following OPTIONS in standalone mode (`-s, --standalone`), for testing purposes.

OPTIONS

-h, –help Summarize available options.

-v, –verbose Log actions to stderr.

–initrc=*FILE* Load shell initrc from FILE instead of the system default.

-R, –resources=*FILE* Load resource set **R** from a file instead of job-info service. This is used for testing.

-j, –jobspec=*FILE* Load jobspec from FILE instead of job-info service. This is used for testing.

-s, –standalone Run as as a local program without Flux instance. Used for testing. In standalone mode an initrc file is not loaded unless specifically requested via the `--initrc` option or specified in jobspec.

OPERATION

When a job has been granted resources by a Flux instance, a flux-shell(1) process is invoked on each broker rank involved in the job. The job shell runs as the job user, and will always have `FLUX_KVS_NAMESPACE` set such that the root of the job shell's KVS accesses will be the guest namespace for the job.

Each flux-shell(1) connects to the local broker, fetches the jobspec and resource set **R** for the job from the job-info module, and uses this information to plan which tasks to locally execute.

Once the job shell has successfully gathered job information, the flux-shell(1) then goes through the following general steps to manage execution of the job:

- register service endpoint specific to the job and userid, typically `<userid>-shell-<jobid>`
- load the system default `initrc.lua` (`$sysconfdir/flux/shell/initrc.lua`), unless overridden by configuration (See JOBSPEC OPTIONS and INITRC sections below)
- call `shell.init` plugin callbacks
- change working directory to the cwd of the job
- enter a barrier to ensure shell initialization is complete on all shells
- emit `shell.init` event to exec.eventlog
- create all local tasks. For each task, the following procedure is used
  - call `task.init` plugin callback
  - launch task, call `task.exec` plugin callback just before `execve(2)`
  - call `task.fork` plugin callback
- once all tasks have started, call `shell.start` plugin callback
- enter shell "start" barrier
- emit `shell.start` event, after which all tasks are known running
- for each exiting task:
  - call `task.exit` plugin callback
  - collect exit status
- call `shell.exit` plugin callback when all tasks have exited.
- exit with max task exit code

PLUGINS

The job shell supports external and builtin plugins which implement most of the advanced job shell features. Job shell plugins are loaded into a plugin stack by name, where the last loaded name wins. Therefore, to override a builtin plugin, an alternate plugin which registers the same name may be loaded at runtime.

Note: Job shell plugins should be written with the assumption their access to Flux services may be restricted as a guest.

C plugins are defined using the Flux standard plugin format. A shell C plugin should therefore export a single symbol `flux_plugin_init()`, in which calls to `flux_plugin_add_handler(3)` should be used to register functions which will be invoked at defined points during shell execution. These callbacks are defined by "topic strings" to which plugins can "subscribe" by calling `flux_plugin_add_handler(3)` and/or `flux_plugin_register(3)` with topic `glob(7)` strings.

Note: `flux_plugin_init(3)` is not called for builtin shell plugins. If a dynamically loaded plugin wishes to set shell options to influence a shell builtin plugin (e.g. to disable its operation), it should therefore do so in `flux_plugin_init()` in order to guarantee that the shell option is set before the builtin attempts to read them.

Simple plugins may also be developed directly in the shell `initrc.lua` file itself (see INITRC section, `plugin.register()` below)

By default, flux-shell supports the following plugin callback topics:

shell.connect Called just after the shell connects to the local Flux broker. (Only available to builtin shell plugins.)

shell.init Called after the shell has finished fetching and parsing the **jobspec** and **R** from the KVS, but before any tasks are started.

task.init Called for each task after the task info has been constructed but before the task is executed.

task.exec Called for each task after the task has been forked just before `execve(2)` is called. This callback is made from within the task process.

task.fork Called for each task after the task if forked from the parent process (flux-shell process)

task.exit Called for each task after it exits and wait_status is available.

shell.start Called after all local tasks have been started. The shell "start" barrier is called just after this callback returns.

shell.log Called by the shell logging facility when a shell component posts a log message.

shell.log-setlevel Called by the shell logging facility when a request to set the shell loglevel is made.

Note however, that plugins may also call into the plugin stack to create new callbacks at runtime, so more topics than those listed above may be available in a given shell instance.

JOBSPEC OPTIONS

On startup, `flux-shell` will examine the jobspec for any shell specific options under the `attributes.system.shell.options` key. These options may be set by the `flux-mini -o, --setopt=OPT` option, or explicitly added to the jobspec by other means.

Job shell options may be switches to enable or disable a shell feature or plugin, or they may take an argument. Because jobspec is a JSON document, job shell options in jobspec may take arguments that are themselves JSON objects. This allows maximum flexibility in runtime configuration of optional job shell behavior. In the list below, if an option doesn't include a =, then it is a simple boolean option or switch and may be specified simply with `-o option` in commands like `flux mini run`.

Options supported by `flux-shell` proper include:

verbose=*INT* Set the shell verbosity to *INT*. A larger value indicates increased verbosity, though setting this value larger than 2 currently has no effect.

initrc=*FILE* Load flux-shell initrc.lua file from *FILE* instead of the default initrc path. For details of the job shell initrc.lua file format, see the INITRC section below.

Job shell plugins may also support configuration via shell options in the jobspec. For specific information about runtime-loaded plugins, see the documentation for the specific plugin in question. The following options are supported by the builtin plugins of `flux-shell`:

pty Allocate a pty for the first task rank.

cpu-affinity=*OPT* Adjust the operation of the builtin shell `affinity` plugin. *OPT* may be set to `off` to disable the affinity plugin, or `per-task` to have CPU affinity applied on a per task basis. The default is `on`, which binds all tasks to the assigned set of cores in the job.

gpu-affinity=*OPT* Adjust operation of the builtin shell `gpubind` plugin, which simply sets `CUDA_VISIBLE_DEVICES` to the GPU IDs allocated to the job. *OPT* may be set to `off` to disable the plugin, or `per-task` to divide allocated GPUs among tasks launched by the shell (sets a different GPU ID or IDs for each launched task)

stop-tasks-in-exec Stops tasks in `exec()` using `PTRACE_TRACEME`. Used for debugging parallel jobs. Users should not need to set this option directly.

output.{stdout,stderr}.type=*TYPE* Set job output to for **stderr** or **stdout** to *TYPE*. *TYPE* may be one of `term`, `kvs` or `file` (Default: `kvs`). If only `output.stdout.type` is set, then this option applies to both `stdout` and `stderr`. If set to `file`, then `output.<stream>.path` must also be set for the stream. Most users will not need to set this option directly, as it will be set automatically by options of higher level commands such as `flux-mini`.

output.{stdout,stderr}.path=*PATH* Set job stderr/out file output to PATH.

input.stdin.type=*TYPE* Set job input for **stdin** to *TYPE*. *TYPE* may be either `service` or `file`. Users should not need to set this option directly as it will be handled by options of higher level commands like `flux-mini`.

exit-timeout=*VALUE* A fatal exception is raised on the job 30s after the first task exits. The timeout period may be altered by providing a different value in Flux Standard Duration form. A value of `none` disables generation of the exception.

exit-on-error If the first task to exit was signaled or exited with a nonzero status, raise a fatal exception on the job immediately.

SHELL INITRC

At initialization, flux-shell(1) reads a Lua initrc file which can be used to customize the shell operation. The initrc is loaded by default from `$sysconfdir/flux/shell/initrc.lua` (or `/etc/flux/shell/initrc.lua` for a "standard" install), but a different path may be specified when launching a job via the `initrc` shell option.

A job shell initrc file may be used to adjust the shell plugin searchpath, load specific plugins, read and set shell options, and even extend the shell itself using Lua.

Since the job shell `initrc` is a Lua file, any Lua syntax is supported. Job shell specific functions and tables are described below:

plugin.searchpath The current plugin searchpath. This value can be queried, set, or appended. E.g. to add a new path to the plugin search path: `plugin.searchpath = plugin.searchpath + ':' + path`

plugin.load({file=glob, [conf=table]}) Explicitly load one more shell plugins. This function takes a table argument with `file` and `conf` arguments. The `file` argument is a glob of one or more plugins to load. If an absolute path is not specified, then the glob will be relative to `plugin.searchpath`. E.g. `plugin.load { file = "*.so" }` will load all `.so` plugins in the current search path. The `conf` option allows static configuration values to be passed to plugin initialization functions when supported.

For example a plugin `test.so` may be explicitly loaded with configuration via:

```
plugin.load { file = "test.so", conf = { value = "foo" } }
```

plugin.register({name=plugin_name, handlers=handlers_table) Register a Lua plugin. Requires a table argument with the plugin `name` and a set of `handlers`. `handlers_table` is an array of tables, each of which must define `topic`, a topic glob of shell plugin callbacks to which to subscribe, and `fn` a handler function to call for each match

For example, the following plugin would log the topic string for every possible plugin callback (except for callbacks which are made before the shell logging facility is initialized)

```
plugin.register {
  name = "test",
  handlers = {
     { topic = "*",
       fn = function (topic) shell.log ("topic="..topic) end
     },
  }
}
```

source(glob) Source another Lua file or files. Supports specification of a glob, e.g. `source ("*.lua")`. This function fails if a non-glob argument specifies a file that does not exist, or there is an error loading or compiling the Lua chunk.

source_if_exists(glob) Same as `source()`, but do not throw an error if the target file does not exist.

shell.rcpath The directory in which the current initrc file resides.

shell.getenv([name]) Return the job environment (not the job shell environment). This is the environment which will be inherited by the job tasks. If called with no arguments, then the entire environment is copied to a table and returned. Otherwise, acts as `flux_shell_getenv(3)` and returns the value for the environment variable name, or `nil` if not set.

shell.setenv(var, val, [overwrite]) Set environment variable `var` to value `val` in the job environment. If `overwrite` is set and is `0` or `false` then do not overwrite existing environment variable value.

shell.unsetenv(var) Unset environment variable `var` in job environment.

shell.options A virtual index into currently set shell options, including those set in jobspec. This table can be used to check jobspec options, and even to force certain options to a value by default e.g. `shell.options['cpu-affinity'] = "per-task"`, would force `cpu-affinity` shell option to `per-task`.

shell.options.verbose Current flux-shell verbosity. This value may be changed at runtime, e.g. `shell.options.verbose = 2` to set maximum verbosity.

shell.options.standalone True if the shell is running in "standalone" mode for testing.

shell.info Returns a Lua table of shell information obtained via `flux_shell_get_info(3)`. This table includes

jobid The current jobid.

rank The rank of the current shell within the job.

size The number of flux-shell processes participating in this job.

ntasks The total number of tasks in this job.

service The service string advertised by the shell.

options.verbose True if the shell is running in verbose mode.

options.standalone True if the shell was run in standalone mode.

jobspec The jobspec of the current job

R The resource set **R** of the current job

shell.rankinfo Returns a Lua table of rank-specific shell information for the current shell rank. See *shell.get_rankinfo()* for a description of the members of this table.

shell.get_rankinfo(shell_rank) Query rank-specific shell info as in the function call `flux_shell_get_rank_info(3)`. If `shell_rank` is not provided then the current rank is used. Returns a table of rank-specific information including:

broker_rank The broker rank on which `shell_rank` is running.

ntasks The number of local tasks assigned to `shell_rank`.

resources A table of resources by name (e.g. "core", "gpu") assigned to `shell_rank`, e.g. `{ core = "0-1", gpu = "0" }`.

shell.log(msg), shell.debug(msg), shell.log_error(msg) Log messages to the shell log facility at INFO, DEBUG, and ERROR levels respectively.

shell.die(msg) Log a FATAL message to the shell log facility. This generates a job exception and will terminate the job.

The following task-specific initrc data and functions are available only in one of the `task.*` plugin callbacks. An error will be generated if they are accessed from any other context.

task.info Returns a Lua table of task specific information for the "current" task (see `flux_shell_task_get_info(3)`). Included members of the `task.info` table include:

localid The local task rank (i.e. within this shell)

rank The global task rank (i.e. within this job)

state The current task state name

pid The process id of the current task (if task has been started)

wait_status (Only in `task.exit`) The status returned by `waitpid(2)` for this task.

exitcode (Only in `task.exit`) The exit code if `WIFEXTED()` is true.

signaled (Only in `task.exit`) If task was signaled, this member will be non-zero integer signal number that caused the task to exit.

task.getenv(var) Get the value of environment variable `var` if set in the current task's environment. This function reads the environment from the underlying `flux_cmd_t` for a shell task, and thus only makes sense before a task is executed, e.g. in `task.init` and `task.exec` callbacks.

task.unsetenv(var) Unset environment variable `var` for the current task. As with `task.getenv()` this function is only valid before a task has been started.

task.setenv(var, value, [overwrite]) Set environment variable `var` to `val` for the current task. If `overwrite` is set to `0` or `false`, then do not overwrite any current value. As with `task.getenv()` and `task.unsetenv()`, this function only has an effect before the task is started.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-mini(1)

1.1.22 flux-version(1)

SYNOPSIS

flux version

DESCRIPTION

flux-version(1) prints version information for flux components. At a minimum, the version of flux commands and the currently linked libflux-core.so library is displayed. If running within an instance, the version of the flux-broker found and FLUX_URI are also included. Finally, if flux is compiled against flux-security, then the version of the currently linked libflux-security is included.

RESOURCES

Github: http://github.com/flux-framework

1.1.23 flux(1)

SYNOPSIS

flux [*OPTIONS*] *CMD* [*CMD-OPTIONS*]

DESCRIPTION

Flux is a modular framework for resource management.

flux(1) is a front end for Flux sub-commands. "flux -h" summarizes the core Flux commands. "flux help *CMD*" displays the manual page for *CMD*.

If *CMD* contains a slash "/" character, it is executed directly, bypassing the sub-command search path.

OPTIONS

-h, –help Display help on options, and a list of the core Flux sub-commands.

-p, –parent If current instance is a child, connect to parent instead. Also sets *FLUX_KVS_NAMESPACE* if current instance is confined to a KVS namespace in the parent. This option may be specified multiple times.

-v, –verbose Display command environment, and the path search for *CMD*.

-V, –version Convenience option to run flux-version(1).

SUB-COMMAND ENVIRONMENT

flux(1) uses compiled-in install paths and its environment to construct the environment for sub-commands.

Sub-command search path Look for "flux-*CMD*" executable by searching a path constructed with the following prototype:

```
[getenv FLUX_EXEC_PATH_PREPEND]:install-path:\
  [getenv FLUX_EXEC_PATH]
```

setenv FLUX_MODULE_PATH Set up broker module search path according to:

```
[getenv FLUX_MODULE_PATH_PREPEND]:install-path:\
  [getenv FLUX_MODULE_PATH]
```

setenv FLUX_CONNECTOR_PATH Set up search path for connector modules used by libflux to open a connection to the broker

```
[getenv FLUX_CONNECTOR_PATH_PREPEND]:install-path:\
  [getenv FLUX_CONNECTOR_PATH]
```

setenv FLUX_SEC_DIRECTORY Set directory for Flux CURVE keys. This is not a search path. If unset, flux(1) sets it to $HOME/flux.

setenv LUA_PATH Set Lua module search path:

[getenv FLUX_LUA_PATH_PREPEND];[getenv LUA_PATH];install-path;

setenv LUA_CPATH Set Lua binary module search path:

[getenv FLUX_LUA_CPATH_PREPEND];[getenv LUA_CPATH];install-path;

setenv PYTHONPATH Set Python module search path:

```
[getenv FLUX_PYTHONPATH_PREPEND]:[getenv PYTHONPATH];install-path
```

RESOURCES

Github: http://github.com/flux-framework

1.2 man3

1.2.1 flux_attr_get(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
const char *flux_attr_get (flux_t *h, const char *name);
```

```
int flux_attr_set (flux_t *h, const char *name, const char *val);
```

DESCRIPTION

Flux broker attributes are both a simple, general-purpose key-value store with scope limited to the local broker rank, and a method for the broker to export information needed by Flux services and utilities.

`flux_attr_get()` retrieves the value of the attribute *name*.

Attributes that have the broker tags as *immutable* are cached automatically in the flux_t handle. For example, the "rank" attribute is a frequently accessed attribute whose value never changes. It will be cached on the first access and thereafter does not require an RPC to the broker to access.

`flux_attr_set()` updates the value of attribute *name* to *val*. If *name* is not currently a valid attribute, it is created. If *val* is NULL, the attribute is unset.

RETURN VALUE

`flux_attr_get()` returns the requested value on success. On error, NULL is returned and errno is set appropriately.

`flux_attr_set()` returns zero on success. On error, -1 is returned and errno is set appropriately.

ERRORS

ENOENT The requested attribute is invalid or has a NULL value on the broker.

EINVAL Some arguments were invalid.

EPERM Set was attempted on an attribute that is not writable with the user's credentials.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-lsattr(1), flux-getattr(1), flux-setattr(1), flux-broker-attributes(7), RFC 3: CMB1 - Flux Comms Message Broker Protocol

1.2.2 flux_aux_set(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_free_f)(void *arg);
```

```
void *flux_aux_get (flux_t *h, const char *name);
```

```
int flux_aux_set (flux_t *h, const char *name,
                  void *aux, flux_free_f destroy);
```

DESCRIPTION

`flux_aux_set()` attaches application-specific data to the parent object *h*. It stores data *aux* by key *name*, with optional destructor *destroy*. The destructor, if non-NULL, is called when the parent object is destroyed, or when *key* is overwritten by a new value. If *aux* is NULL, the destructor for a previous value, if any is called, but no new value is stored. If *name* is NULL, *aux* is stored anonymously.

`flux_aux_get()` retrieves application-specific data by *name*. If the data was stored anonymously, it cannot be retrieved. Note that `flux_aux_get()` does not scale to a large number of items, and flux module handles may persist for a long time.

Names beginning with "flux::" are reserved for internal use.

RETURN VALUE

`flux_aux_get()` returns data on success, or NULL on failure, with errno set.

`flux_aux_set()` returns 0 on success, or -1 on failure, with errno set.

ERRORS

EINVAL Some arguments were invalid.

ENOMEM Out of memory.

ENOENT `flux_aux_get()` could not find an entry for *key*.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_open(3)

1.2.3 flux_child_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
flux_watcher_t *flux_child_watcher_create (flux_reactor_t *r,
                                           int pid, bool trace,
                                           flux_watcher_f cb, void *arg);
```

```
int flux_child_watcher_get_rpid (flux_watcher_t *w);
```

```
int flux_child_watcher_get_rstatus (flux_watcher_t *w);
```

DESCRIPTION

`flux_child_watcher_create()` creates a reactor watcher that monitors state transitions of child processes. If *trace* is false, only child termination will trigger an event; otherwise, stop and start events may be generated.

The callback *revents* argument should be ignored.

The process id that had a transition may be obtained by calling `flux_child_watcher_get_rpid()`.

The status value returned by waitpid(2) may be obtained by calling `flux_child_watcher_get_rstatus()`.

Only a Flux reactor created with the FLUX_REACTOR_SIGCHLD flag can be used with child watchers, as the reactor must register a SIGCHLD signal watcher before any processes are spawned. Only one reactor instance per program may be created with this capability.

RETURN VALUE

flux_child_watcher_create() returns a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

EINVAL Reactor was not created with FLUX_REACTOR_SIGCHLD.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3)

libev home page

1.2.4 flux_content_load(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_content_load (flux_t *h,
                                  const char *blobref,
                                  int flags);
```

```
int flux_content_load_get (flux_future_t *f,
                           const void **buf,
                           size_t *len);
```

```
flux_future_t *flux_content_store (flux_t *h,
                                   const void *buf,
                                   size_t len,
                                   int flags);
```

```
int flux_content_store_get (flux_future_t *f,
                            const char **ref);
```

DESCRIPTION

The Flux content service is an append-only, immutable, content addressed data storage service unique to each Flux instance, described in RFC 10. All functions described below are idempotent.

`flux_content_load()` sends a request to the content service to load a data blob by *blobref*, a hash digest whose format is described in RFC 10. A `flux_future_t` object which encapsulates the remote procedure call state is returned. *flags* is a mask that may include the values described below.

`flux_request_load_get()` completes a load operation, blocking on response(s) if needed, parsing the result, and returning the requested blob in *buf* and its length in *len*. *buf* is valid until `flux_future_destroy()` is called.

`flux_content_store()` sends a request to the content service to store a data blob *buf* of length *len*. A `flux_future_t` object which encapsulates the remote procedure call state is returned. *flags* is a mask that may include the values described below.

`flux_content_store_get()` completes a store operation, blocking on response(s) if needed, and returning a blobref that can be used to retrieve the stored blob. The blobref string is valid until `flux_future_destroy()` is called.

These functions may be used asynchronously. See `flux_future_then(3)` for details.

FLAGS

The following are valid bits in a *flags* mask passed as an argument to `flux_content_load()` or `flux_content_store()`.

CONTENT_FLAG_CACHE_BYPASS Send the request directly to the backing store (default sqlite), bypassing the cache.

CONTENT_FLAG_UPSTREAM Direct the request to the next broker upstream on the TBON rather than to the local broker.

RETURN VALUE

`flux_content_load()` and `flux_content_store()` return a `flux_future_t` on success, or NULL on failure with errno set appropriately.

`flux_content_load_get()` and `flux_content_store_get()` return 0 on success, or -1 on failure with errno set appropriately.

ERRORS

EINVAL One of the arguments was invalid.

ENOMEM Out of memory.

ENOENT An unknown blob was requested.

EPROTO A request was malformed.

EFBIG A blob larger than the configured maximum blob size could not be stored. See flux-broker-attributes(7).

ENOSYS The CONTENT_FLAG_CACHE_BYPASS flag was set in a request, but no backing store module is loaded.

EHOSTUNREACH The CONTENT_FLAG_UPSTREAM flag was set in a request received by the rank 0 broker.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_rpc(3), flux_future_get(3)

RFC 10: Content Storage Service

1.2.5 flux_core_version(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
int flux_core_version (int *major, int *minor, int *patch);
```

```
const char *flux_core_version_string (void);
```

DESCRIPTION

flux-core defines several macros and functions to let API users determine the version they are working with. A version has three components (*major*, *minor*, *patch*), accessible with the following macros:

FLUX_CORE_VERSION_MAJOR (integer) incremented when there are incompatible API changes

FLUX_CORE_VERSION_MINOR (integer) incremented when functionality is added in a backwards-compatible manner

FLUX_CORE_VERSION_PATCH (integer) incremented when bug fixes are added in a backwards-compatible manner

These definitions conform to the *semantic versioning* standard (see below). In addition, the following convenience macros are available:

FLUX_CORE_VERSION_HEX (hex) the three versions combined into a three-byte integer value, useful for comparing versions with <, =, and > operators.

FLUX_CORE_VERSION_STRING (string) the three versions above separated by periods, with optional `git-describe(1)` suffix preceded by a hyphen, if the version is a development snapshot.

Note that major version zero (0.y.z) is for initial development. Under version zero, the public API should not be considered stable.

Functions are also available to access the same values. While the header macros tell what version of flux-core your program was compiled against, the functions tell what version your program is dynamically linked with.

`flux_core_version()` sets *major*, *minor*, and *patch* to the values of the macros above. If any parameters are NULL, no assignment is attempted.

`flux_core_version_string()` returns the string value.

RETURN VALUE

`flux_core_version ()` returns the hex version.

`flux_core_version_string ()` returns the version string

ERRORS

These functions cannot fail.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

semver.org[Semantic Versioning 2.0.0]

1.2.6 flux_event_decode(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
int flux_event_decode (const flux_msg_t *msg,
                       const char **topic,
                       const char **s);
```

```
int flux_event_decode_raw (const flux_msg_t *msg,
                           const char **topic,
                           const void **data, int *len);
```

```
int flux_event_unpack (const flux_msg_t *msg,
                       const char **topic,
                       const char *fmt, ...);
```

```
flux_msg_t *flux_event_encode (const char *topic,
                               const char *s);
```

```
flux_msg_t *flux_event_encode_raw (const char *topic,
                                   const void *data, int len);
```

```
flux_msg_t *flux_event_pack (const char *topic,
                             const char *fmt, ...);
```

DESCRIPTION

`flux_event_decode()` decodes a Flux event message *msg*.

topic, if non-NULL, will be set to the message's topic string. The storage for this string belongs to *msg* and should not be freed.

s, if non-NULL, will be set to the message's NULL-terminated string payload. If no payload exists, it is set to NULL. The storage for this string belongs to *msg* and should not be freed.

`flux_event_decode_raw()` decodes an event message with a raw payload, setting *data* and *len* to the payload data and length. The storage for the raw payload belongs to *msg* and should not be freed.

`flux_event_unpack()` decodes a Flux event message with a JSON payload as above, parsing the payload using variable arguments with a format string in the style of jansson's `json_unpack()` (used internally). Decoding fails if the message doesn't have a JSON payload.

`flux_event_encode()` encodes a Flux event message with topic string *topic* and optional NULL-terminated string payload *s*. The newly constructed message that is returned must be destroyed with `flux_msg_destroy()`.

`flux_event_encode_raw()` encodes a Flux event message with topic string *topic*. If *data* is non-NULL, its contents will be used as the message payload, and the payload type set to raw.

`flux_event_pack()` encodes a Flux event message with a JSON payload as above, encoding the payload using variable arguments with a format string in the style of jansson's `json_pack()` (used internally). Decoding fails if the message doesn't have a JSON payload.

Events propagated to all subscribers. Events will not be received without a matching subscription established using `flux_event_subscribe()`.

ENCODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_pack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a null terminated UTF-8 string to a JSON string.

s# (string)['const char *', 'int'] Convert a UTF-8 buffer of a given length to a JSON string.

s% (string)['const char *', 'size_t'] Like **s#** but the length argument is of type size_t.

+ ['const char *'] Like **s**, but concatenate to the previous string. Only valid after a string.

+# ['const char *', 'int'] Like **s#**, but concatenate to the previous string. Only valid after a string.

+% ['const char *', 'size_t'] Like **+#**, but the length argument is of type size_t.

n (null) Output a JSON null value. No argument is consumed.

b (boolean)['int'] Convert a C int to JSON boolean value. Zero is converted to *false* and non-zero to *true*.

i (integer)['int'] Convert a C int to JSON integer.

I (integer)['int64_t'] Convert a C int64_t to JSON integer. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert a C double to JSON real.

o (any value)['json_t *'] Output any given JSON value as-is. If the value is added to an array or object, the reference to the value passed to **o** is stolen by the container.

O (any value)['json_t *'] Like **o**, but the argument's reference count is incremented. This is useful if you pack into an array or object and want to keep the reference for the JSON value consumed by **O** to yourself.

[fmt] (array) Build an array with contents from the inner format string. **fmt** may contain objects and arrays, i.e. recursive value building is supported.

{fmt} (object) Build an object with contents from the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must be a string as object keys are always strings. The second, fourth, etc. format specifier represent a value. Any value may be an object or array, i.e. recursive value building is supported.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Building Values

DECODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_unpack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a JSON string to a pointer to a null terminated UTF-8 string. The resulting string is extracted by using 'json_string_value()' internally, so it exists as long as there are still references to the corresponding JSON string.

n (null) Expect a JSON null value. Nothing is extracted.

b (boolean)['int'] Convert a JSON boolean value to a C int, so that *true* is converted to 1 and *false* to 0.

i (integer)['int'] Convert a JSON integer to a C int.

I (integer)['int64_t'] Convert a JSON integer to a C int64_t. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert JSON real to a C double.

F (real)['double'] Convert JSON number (integer or real) to a C double.

o (any value)['json_t *'] Store a JSON value, with no conversion, to a json_t pointer.

O (any value)['json_t *'] Like **o**, but the JSON value's reference count is incremented.

[fmt] (array) Convert each item in the JSON array according to the inner format string. **fmt** may contain objects and arrays, i.e. recursive value extraction is supported.

{fmt} (object) Convert each item in the JSON object according to the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must by **s**. The corresponding argument to unpack functions is read as the object key. The second, fourth, etc. format specifier represent a value and is written to the address given as the corresponding argument. Note that every other argument is read from and every other is written to. **fmt** may contain objects and arrays as values, i.e. recursive value extraction is supported. Any **s** representing a key may be suffixed with **?** to make the key optional. If the key is not found, nothing is extracted.

! This special format specifier is used to enable the check that all object and array items are accessed, on a per-value basis. It must appear inside an array or object as the last format specifier before the closing bracket or brace.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Parsing and Validating Values

RETURN VALUE

Decoding functions return 0 on success. On error, -1 is returned, and errno is set appropriately.

Encoding functions return a message on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

EINVAL The *msg* argument was NULL or there was a problem encoding.

ENOMEM Memory was unavailable.

EPROTO Message decoding failed, such as due to incorrect message type, missing topic string, etc..

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_event_subscribe(3)

1.2.7 flux_event_publish(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_event_publish (flux_t *h,
                                   const char *topic, int flags,
                                   const char *s);
```

```
flux_future_t *flux_event_publish_pack (flux_t *h,
                                        const char *topic, int flags,
                                        const char *fmt, ...);
```

```
flux_future_t *flux_event_publish_raw (flux_t *h,
                                       const char *topic, int flags,
                                       const void *data, int len);
```

```
int flux_event_publish_get_seq (flux_future_t *f, int *seq);
```

DESCRIPTION

`flux_event_publish()` sends an event message with topic string *topic*, *flags* as described below, and optional payload *s*, a NULL-terminated string, or NULL indicating no payload. The returned future is fulfilled once the event is accepted by the broker and assigned a global sequence number.

`flux_event_publish_pack()` is similar, except the JSON payload is constructed using `json_pack()` style arguments (see below).

`flux_event_publish_raw()` is similar, except the payload is raw *data* of length *len*.

`flux_event_publish_get_seq()` may be used to retrieve the sequence number assigned to the message once the future is fulfilled.

CONFIRMATION SEMANTICS

All Flux events are "open loop" in the sense that publishers get no confirmation that subscribers have received a message. However, the above functions do confirm, upon fulfillment of the returned future, that the published event has been received by the broker and assigned a global sequence number.

Gaps in the sequence trigger the logging of errors currently, and in the future will trigger recovery of lost events, so these confirmations do indicate that Flux's best effort at event propagation is under way.

If this level of confirmation is not required, one may encode an event message directly using `flux_event_encode(3)` and related functions and send it directly with `flux_send(3)`.

FLAGS

The *flags* argument in the above functions must be zero, or the logical OR of the following values:

FLUX_MSGFLAG_PRIVATE Indicates that the event should only be visible to the instance owner and the sender.

ENCODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_pack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a null terminated UTF-8 string to a JSON string.

s# (string)['const char *', 'int'] Convert a UTF-8 buffer of a given length to a JSON string.

s% (string)['const char *', 'size_t'] Like **s#** but the length argument is of type size_t.

+ ['const char *'] Like **s**, but concatenate to the previous string. Only valid after a string.

+# ['const char *', 'int'] Like **s#**, but concatenate to the previous string. Only valid after a string.

+% ['const char *', 'size_t'] Like **+#**, but the length argument is of type size_t.

n (null) Output a JSON null value. No argument is consumed.

b (boolean)['int'] Convert a C int to JSON boolean value. Zero is converted to *false* and non-zero to *true*.

i (integer)['int'] Convert a C int to JSON integer.

I (integer)['int64_t'] Convert a C int64_t to JSON integer. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert a C double to JSON real.

o (any value)['json_t *'] Output any given JSON value as-is. If the value is added to an array or object, the reference to the value passed to **o** is stolen by the container.

O (any value)['json_t *'] Like **o**, but the argument's reference count is incremented. This is useful if you pack into an array or object and want to keep the reference for the JSON value consumed by **O** to yourself.

[fmt] (array) Build an array with contents from the inner format string. **fmt** may contain objects and arrays, i.e. recursive value building is supported.

{fmt} (object) Build an object with contents from the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must be a string as object keys are always strings. The second, fourth, etc. format specifier represent a value. Any value may be an object or array, i.e. recursive value building is supported.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Building Values

RETURN VALUE

These functions return a future on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

EINVAL Some arguments were invalid.

ENOMEM Out of memory.

EPROTO A protocol error was encountered.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_event_decode(3), flux_event_subscribe(3)

1.2.8 flux_event_subscribe(3)

SYNOPSIS

#include <flux/core.h>

int flux_event_subscribe (flux_t *h, const char *topic);

int flux_event_unsubscribe (flux_t *h, const char *topic);

DESCRIPTION

Flux events are broadcast across the session, but are only delivered to handles that subscribe to them by topic. Topic strings consist of one or more words separated by periods, interpreted as a hierarchical name space.

`flux_event_subscribe()` requests that event messages matching *topic* be delivered via `flux_recv(3)`. A match consists of a string comparison of the event topic and the subscription topic, up to the length of the subscription topic. Thus "foo." matches events with topics "foo.bar" and "foo.baz", and "" matches all events. This matching algorithm is inherited from ZeroMQ. Globs or regular expressions are not allowed in subscriptions, and the period delimiter is included in the comparison.

`flux_event_unsubscribe()` unsubscribes to a topic. The *topic* argument must exactly match that provided to `flux_event_subscribe()`.

Duplicate subscriptions are allowed in the subscription list but will not result in multiple deliveries of a given message. Each duplicate subscription requires a separate unsubscribe.

It is not necessary to remove subscriptions with `flux_event_unsubscribe()` prior to calling `flux_close(3)`.

RETURN VALUE

These functions return 0 on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL Some arguments were invalid.

ENOMEM Out of memory.

EXAMPLES

This example opens the Flux broker, subscribes to heartbeat messages, displays one, then quits.

```
#include <flux/core.h>
#include "src/common/libutil/log.h"

int main (int argc, char **argv)
{
    flux_t *h;
    flux_msg_t *msg;
    const char *topic;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("flux_open");
    if (flux_event_subscribe (h, "heartbeat.pulse") < 0)
        log_err_exit ("flux_event_subscribe");
    if (!(msg = flux_recv (h, FLUX_MATCH_EVENT, 0)))
        log_err_exit ("flux_recv");
    if (flux_msg_get_topic (msg, &topic) < 0)
        log_err_exit ("flux_msg_get_topic");
    printf ("Event: %s\n", topic);
    if (flux_event_unsubscribe (h, "heartbeat.pulse") < 0)
        log_err_exit ("flux_event_unsubscribe");
    flux_msg_destroy (msg);
    flux_close (h);
    return (0);
}
```

RESOURCES

Github: http://github.com/flux-framework

1.2.9 flux_fatal_set(3)

SYNOPSIS

#include <flux/core.h>

typedef void (*flux_fatal_f)(const char *msg, void *arg);

void flux_fatal_set (flux_t *h, flux_fatal_f fun, void *arg);

void flux_fatal_error (flux_t *h, const char *fun, const char *msg);

FLUX_FATAL (flux_t *h);

DESCRIPTION

`flux_fatal_set()` configures an optional fatal error function *fun* to be called internally by `libflux_core` if an error occurs that is fatal to the handle *h*. A fatal error is any error that renders the handle unusable. The function may log *msg*, terminate the program, or take other action appropriate to the application.

If a fatal error function is not registered, or if the fatal error function returns, error handling proceeds as normal.

The fatal error function will only be called once, for the first fatal error encountered.

arg is an optional argument passed through to the fatal error function.

`FLUX_FATAL()` is a macro that calls

:: flux_fatal_error (h, __FUNCTION__, strerror (errno))

which translates to a fatal error function called with *msg* set to "function name: error string".

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_open(3)

1.2.10 flux_fd_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
flux_watcher_t *flux_fd_watcher_create (flux_reactor_t *r,
                                        int fd, int events,
                                        flux_watcher_f callback,
                                        void *arg);
```

```
int flux_fd_watcher_get_fd (flux_watcher_t *w);
```

DESCRIPTION

`flux_fd_watcher_create()` creates a flux_watcher_t object which can be used to monitor for events on a file descriptor *fd*. When events occur, the user-supplied *callback* is invoked.

The *events* and *revents* arguments are a bitmask containing a logical OR of the following bits. If a bit is set in *events*, it indicates interest in this type of event. If a bit is set in *revents*, it indicates that this event has occurred.

FLUX_POLLIN The file descriptor is ready for reading.

FLUX_POLLOUT The file descriptor is ready for writing.

FLUX_POLLERR The file descriptor has encountered an error. This bit is ignored if it is set in the create *events* argument.

Events are processed in a level-triggered manner. That is, the callback will continue to be invoked as long as the event has not been fully consumed or cleared, and the watcher has not been stopped.

`flux_fd_watcher_get_fd()` is used to obtain the file descriptor from within the flux_watcher_f callback.

RETURN VALUE

`flux_fd_watcher_create()` returns a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

`flux_fd_watcher_get_fd()` returns the file descriptor associated with the watcher.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3).

1.2.11 flux_flags_set(3)

SYNOPSIS

#include <flux/core.h>

void flux_flags_set (flux_t *h, int flags);

void flux_flags_unset (flux_t *h, int flags);

int flux_flags_get (flux_t *h);

DESCRIPTION

`flux_flags_set()` sets new open *flags* in handle *h*. The resulting handle flags will be a logical or of the old flags and the new.

`flux_flags_unset()` clears open *flags* in handle *h*. The resulting handle flags will be a logical and of the old flags and the inverse of the new.

`flux_flags_get()` can be used to retrieve the current open flags from handle *h*.

The valid flags are described in `flux_open(3)`.

RETURN VALUE

`flux_flags_get()` returns the current flags.

ERRORS

These functions never fail.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_open(3)

1.2.12 flux_future_and_then(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_future_and_then (flux_future_t *f,
                                     flux_continuation_f cb, void *arg);
flux_future_t *flux_future_or_then (flux_future_t *f,
                                    flux_continuation_f cb, void *arg);
```

```
int flux_future_continue (flux_future_t *prev, flux_future_t *f);
void flux_future_continue_error (flux_future_t *prev, int errnum,
                                 const char *errstr);
```

```
int flux_future_fulfill_next (flux_future_t *f,
                              void *result,
                              flux_free_f free_fn);
```

DESCRIPTION

See `flux_future_get(3)` for general functions that operate on futures, and `flux_future_create(3)` for a description of the `flux_future_t` base type. This page covers functions for the sequential composition of futures, i.e. chains of dependent futures.

`flux_future_and_then(3)` is similar to `flux_future_then(3)`, but returns a future that may later be "continued" from the continuation callback `cb`. The provided continuation callback `cb` is only executed when the future argument `f` is fulfilled successfully. On error, the error from `f` is automatically propagated to the "next" future in the chain (returned by the function).

`flux_future_and_then()` is useful when a series of asynchronous operations, each returning a `flux_future_t`, depend on the result of a previous operation. That is, `flux_future_and_then()` returns a placeholder future for an eventual future that can't be created until the continuation `cb` is run. The returned future can then be used as a synchronization handle or even passed to another `flux_future_and_then()` in the

chain. By default, the next future in the chain will be fulfilled immediately using the result of the previous future after return from the callback `cb`. Most callbacks, however, should use either `flux_future_continue(3)` or `flux_future_continue_error(3)` to pass an intermediate future to use in fulfillment of the next future in the chain.

`flux_future_or_then(3)` is like `flux_future_and_then()`, except the continuation callback `cb` is run when the future `f` is fulfilled with an error. This function is useful for recovery or other error handling (other than the default behavior of propagating an error down the chain to the final result). The `flux_future_or_then()` callback offers a chance to successfully fulfill the "next" future in the chain, even when the "previous" future was fulfilled with an error.

As with `flux_future_and_then()` the continuation `cb` function for `flux_future_or_then()` should call `flux_future_continue()` or `flux_future_continue_error()`, or the result of the previous future will be propagated immediately to the next future in the chain.

`flux_future_continue(3)` continues the next future embedded in `prev` (created by `flux_future_and_then()` or `flux_future_or_then()`) with the eventual result of the provided future `f`. This allows a future that was not created until the context of the callback to continue a sequential chain of futures created earlier. After the call to `flux_future_continue(3)` completes, the future `prev` may safely be destroyed. `flux_future_continue(3)` may be called with `f` equal to `NULL` if the caller desires the next future in the chain to **not** be fulfilled, in order to disable the automatic fulfillment that normally occurs for non-continued futures after the callback completes.

`flux_future_continue_error(3)` is like `flux_future_continue()` but immediately fulfills the next future in the chain with an error and an optional error string. Once `flux_future_continue_error(3)` completes, the future `prev` may safely be destroyed.

`flux_future_fulfill_next(3)` is like `flux_future_fulfill(3)`, but fulfills the next future in the chain instead of the current future (which is presumably already fulfilled). This call is useful when a chained future is being used for post-processing a result from intermediate future-based calls, as it allows the next future to be fulfilled with a custom result, instead of with the value of another future as in `flux_future_continue(3)`.

RETURN VALUE

`flux_future_and_then()` and `flux_future_or_then()` return a `flux_future_t` on success, or NULL on error. If both functions are called on the same future, the returned `flux_future_t` from each will be the same object.

`flux_future_continue()` returns 0 on success, or -1 on error with errno set.

`flux_future_fulfill_next()` returns 0 on success, or -1 with errno set to `EINVAL` if the target future does not have a next future to fulfill.

ERRORS

ENOMEM Out of memory.

EINVAL Invalid argument.

ENOENT The requested object is not found.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_future_get(3), flux_future_create(3)

1.2.13 flux_future_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_future_init_f)(flux_future_t *f,
                                   flux_reactor_t *r, void *arg);
```

```
flux_future_t *flux_future_create (flux_future_init_f cb, void *arg);
```

```
void flux_future_fulfill (flux_future_t *f,
                          void *result, flux_free_f free_fn);
```

```
void flux_future_fulfill_error (flux_future_t *f, int errnum,
                                const char *errstr);
```

```
void flux_future_fulfill_with (flux_future_t *f, flux_future_t *p);
```

```
void flux_future_fatal_error (flux_future_t *f, int errnum,
                              const char *errstr);
```

```
void *flux_future_aux_get (flux_future_t *f, const char *name);
```

```
int flux_future_aux_set (flux_future_t *f, const char *name,
                         void *aux, flux_free_f destroy);
```

```
void flux_future_set_reactor (flux_future_t *f, flux_t *h);
```

```
flux_reactor_t *flux_future_get_reactor (flux_future_t *f);
```

```
void flux_future_set_flux (flux_future_t *f, flux_t *h);
```

```
flux_t *flux_future_get_flux (flux_future_t *f);
```

DESCRIPTION

See `flux_future_get(3)` for general functions that operate on futures. This page covers functions primarily used when building classes that return futures.

A Flux future represents some activity that may be completed with reactor watchers and/or message handlers. It is intended to be returned by other classes as a handle for synchronization and a container for results. This page describes the future interfaces used by such classes.

A class that returns a future usually provides a creation function that internally calls `flux_future_create()`, and may provide functions to access class-specific result(s), that internally call `flux_future_get()`. The create

function internally registers a *flux_future_init_f* function that is called lazily by the future implementation to perform class-specific reactor setup, such as installing watchers and message handlers.

`flux_future_create()` creates a future and registers the class-specific initialization callback *cb*, and an opaque argument *arg* that will be passed to *cb*. The purpose of the initialization callback is to set up class-specific watchers on a reactor obtained with `flux_future_get_reactor()`, or message handlers on a flux_t handle obtained with `flux_future_get_flux()`, or both. `flux_future_get_reactor()` and `flux_future_get_flux()` return different results depending on whether the initialization callback is triggered by a user calling `flux_future_then()` or `flux_future_wait_for()`. The function may be triggered in one or both contexts, at most once for each. The watchers or message handlers must eventually call `flux_future_fulfill()`, `flux_future_fulfill_error()`, or `flux_future_fatal_error()` to fulfill the future. See REACTOR CONTEXTS below for more information.

`flux_future_fulfill()` fulfills the future, assigning an opaque *result* value with optional destructor *free_fn* to the future. A NULL *result* is valid and also fulfills the future. The *result* is contained within the future and can be accessed with `flux_future_get()` as needed until the future is destroyed.

`flux_future_fulfill_error()` fulfills the future, assigning an *errnum* value and an optional error string. After the future is fulfilled with an error, `flux_future_get()` will return -1 with errno set to *errnum*.

`flux_future_fulfill_with()` fulfills the target future *f* using a fulfilled future *p*. This function copies the pending result or error from *p* into *f*, and adds read-only access to the *aux* items for *p* from *f*. This ensures that any `get` method which requires *aux* items for *p* will work with *f*. This function takes a reference to the source future *p*, so it safe to call `flux_future_destroy (p)` after this call. `flux_future_fulfill_with()` returns -1 on error with *errno* set on failure.

`flux_future_fulfill()`, `flux_future_fulfill_with()`, and `flux_future_fulfill_error()` can be called multiple times to queue multiple results or errors. When callers access future results via `flux_future_get()`, results or errors will be returned in FIFO order. It is an error to call `flux_future_fulfill_with()` multiple times on the same target future *f* with a different source future *p*.

`flux_future_fatal_error()` fulfills the future, assigning an *errnum* value and an optional error string. Unlike `flux_future_fulfill_error()` this fulfillment can only be called once and takes precedence over all other fulfillments. It is used for catastrophic error paths in future fulfillment.

`flux_future_aux_set()` attaches application-specific data to the parent object *f*. It stores data *aux* by key *name*, with optional destructor *destroy*. The destructor, if non-NULL, is called when the parent object is destroyed, or when *key* is overwritten by a new value. If *aux* is NULL, the destructor for a previous value, if any is called, but no new value is stored. If *name* is NULL, *aux* is stored anonymously.

`flux_future_aux_get()` retrieves application-specific data by *name*. If the data was stored anonymously, it cannot be retrieved.

Names beginning with "flux::" are reserved for internal use.

`flux_future_set_reactor()` may be used to associate a Flux reactor with a future. The reactor (or a temporary one, depending on the context) may be retrieved using `flux_future_get_reactor()`.

`flux_future_set_flux()` may be used to associate a Flux broker handle with a future. The handle (or a clone associated with a temporary reactor, depending on the context) may be retrieved using `flux_future_get_flux()`.

Futures may "contain" other futures, to arbitrary depth. That is, an init callback may create futures and use their continuations to fulfill the containing future in the same manner as reactor watchers and message handlers.

REACTOR CONTEXTS

Internally, a future can operate in two reactor contexts. The initialization callback may be called in either or both contexts, depending on which synchronization functions are called by the user. `flux_future_get_reactor()`

and `flux_future_get_flux()` return a result that depends on which context they are called from.

When the user calls `flux_future_then()`, this triggers a call to the initialization callback. The callback would typically call `flux_future_get_reactor()` and/or `flux_future_get_flux()` to obtain the reactor or flux_t handle to be used to set up watchers or message handlers. In this context, the reactor or flux_t handle are exactly the ones passed to `flux_future_set_reactor()` and `flux_future_set_flux()`.

When the user calls `flux_future_wait_for()`, this triggers the creation of a temporary reactor, then a call to the initialization callback. The temporary reactor allows these functions to wait *only* for the future's events, without allowing unrelated watchers registered in the main reactor to run, which might complicate the application's control flow. In this context, `flux_future_get_reactor()` returns the temporary reactor, not the one passed in with `flux_future_set_reactor()`. `flux_future_get_flux()` returns a temporary flux_t handle cloned from the one passed to `flux_future_set_flux()`, and associated with the temporary reactor. After the internal reactor returns, any messages unmatched by the dispatcher on the cloned handle are requeued in the main flux_t handle with `flux_dispatch_requeue()`.

Since the init callback may be made in either reactor context (at most once each), and is unaware of which context that is, it should take care when managing any context-specific state not to overwrite the state from a prior call. The ability to attach objects with destructors anonymously to the future with `flux_future_aux_set()` may be useful for managing the life cycle of reactor watchers and message handlers created by init callbacks.

RETURN VALUE

`flux_future_create()` returns a future on success. On error, NULL is returned and errno is set appropriately.

`flux_future_aux_set()` returns zero on success. On error, -1 is returned and errno is set appropriately.

`flux_future_aux_get()` returns the requested object on success. On error, NULL is returned and errno is set appropriately.

`flux_future_get_flux()` returns a flux_t handle on success. On error, NULL is returned and errno is set appropriately.

`flux_future_get_reactor()` returns a flux_reactor_t on success. On error, NULL is returned and errno is set appropriately.

`flux_future_fulfill_with()` returns zero on success. On error, -1 is returned with errno set to EINVAL if either *f* or *p* is NULL, or *f* and *p* are the same, EAGAIN if the future *p* is not ready, or EEXIST if the function is called multiple times with different *p*.

ERRORS

ENOMEM Out of memory.

EINVAL Invalid argument.

ENOENT The requested object is not found.

EAGAIN The requested operation is not ready. For `flux_future_fulfill_with()`, the target future *p* is not fulfilled.

EEXIST `flux_future_fulfill_with()` was called multiple times with a different target future *p*.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_future_get(3), flux_clone(3)

1.2.14 flux_future_get(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_continuation_f)(flux_future_t *f, void *arg);
```

```
int flux_future_get (flux_future_t *f, const void **result);
```

```
int flux_future_then (flux_future_t *f, double timeout,
                      flux_continuation_f cb, void *arg);
```

```
int flux_future_wait_for (flux_future_t *f, double timeout);
```

```
void flux_future_reset (flux_future_t *f);
```

```
void flux_future_destroy (flux_future_t *f);
```

```
bool flux_future_has_error (flux_future_t *f);
```

```
const char *flux_future_error_string (flux_future_t *f);
```

OVERVIEW

A Flux future represents some activity that may be completed with reactor watchers and/or message handlers. It is both a handle for synchronization and a container for the result. A Flux future is said to be "fulfilled" when a result is available in the future container, or a fatal error has occurred. Flux futures were inspired by similar constructs in other programming environments mentioned in RESOURCES, but are not a faithful implementation of any particular one.

Generally other Flux classes return futures, and may provide class-specific access function for results. The functions described in this page can be used to access, synchronize, and destroy futures returned from any such class. Authors of classes that return futures are referred to `flux_future_create(3)`.

DESCRIPTION

`flux_future_get()` accesses the result of a fulfilled future. If the future is not yet fulfilled, it calls `flux_future_wait_for()` internally with a negative *timeout*, causing it to block until the future is fulfilled. A pointer to the result is assigned to *result* (caller must NOT free), or -1 is returned if the future was fulfilled with an error.

`flux_future_then()` sets up a continuation callback *cb* that is called with opaque argument *arg* once the future is fulfilled. The continuation will normally use `flux_future_get()` or a class-specific access function to obtain the result from the future container without blocking. The continuation may call `flux_future_destroy()` or `flux_future_reset()`. If *timeout* is non-negative, the future must be fulfilled within the specified amount of time or the timeout fulfills it with an error (errno set to ETIMEDOUT).

`flux_future_wait_for()` blocks until the future is fulfilled, or *timeout* (if non-negative) expires. This function may be called multiple times, with different values for *timeout*. If the timeout expires before the future is fulfilled, an error is returned (errno set to ETIMEDOUT) but the future remains unfulfilled. If *timeout* is zero, function times out immediately if the future has not already been fulfilled.

`flux_future_reset()` unfulfills a future, invalidating any result stored in the container, and preparing it to be fulfilled once again. If a continuation was registered, it remains in effect for the next fulfillment. If a timeout was specified when the continuation was registered, it is restarted.

`flux_future_destroy()` destroys a future, including any result contained within.

`flux_future_has_error()` tests if an error exists in the future or not. It can be useful for determining if an error exists in a future or in other parts of code that may wrap around a future. It is commonly called before calling `flux_future_error_string()`.

`flux_future_error_string()` returns the error string stored in a future. If the future was fulfilled with an optional error string, `flux_future_error_string()` will return that string. Otherwise, it will return the string associated with the error number set in a future. If the future is a NULL pointer, not fulfilled, or fulfilled with a non-error, NULL is returned.

RETURN VALUE

`flux_future_then()`, `flux_future_get()`, and `flux_future_wait_for()` return zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

EINVAL Invalid argument.

ETIMEDOUT A timeout passed to `flux_future_wait_for()` expired before the future was fulfilled.

RESOURCES

Github: http://github.com/flux-framework

C++ std::future: http://en.cppreference.com/w/cpp/thread/future

Java `util.concurrent.Future`: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

Python3 concurrent.futures: https://docs.python.org/3/library/concurrent.futures.html

SEE ALSO

flux_future_create (3)

1.2.15 flux_future_wait_all_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_future_wait_all_create (void);
flux_future_t *flux_future_wait_any_create (void);
```

```
int flux_future_push (flux_future_t *cf, const char *name, flux_future_t *f);
```

```
const char *flux_future_first_child (flux_future_t *cf);
const char *flux_future_next_child (flux_future_t *cf);
flux_future_t *flux_future_get_child (flux_future_t *cf, const char *name);
```

DESCRIPTION

See `flux_future_get(3)` for general functions that operate on futures, and `flux_future_create(3)` for a description of the `flux_future_t` base type. This page covers functions used for composing futures into composite types using containers that allow waiting on all or any of a set of child futures.

`flux_future_wait_all_create(3)` creates a future that is an empty container for other futures, which can subsequently be pushed into the container using `flux_future_push(3)`. The returned future will be automatically fulfilled when **all** children futures have been fulfilled. The caller may then use `flux_future_first_child(3)`, `flux_future_next_child(3)`, and/or `flux_future_get_child(3)` and expect that `flux_future_get(3)` will not block for any of these child futures. This function is useful to synchronize on a series of futures that may be run in parallel.

`flux_future_wait_any_create(3)` creates a composite future that will be fulfilled once **any** one of its children are fulfilled. Once the composite future is fulfilled, the caller will need to traverse the child futures to determine which was fulfilled. This function is useful to synchronize on work where any one of several results is sufficient to continue.

`flux_future_push(3)` places a new child future `f` into a future composite created by either `flux_future_wait_all_create(3)` or `flux_future_wait_any_create(3)`. A `name` is provided for the child so that the child future can be easily differentiated from other futures inside the container once the composite future is fulfilled.

Once a `flux_future_t` is pushed onto a composite future with `flux_future_push(3)`, the memory for the child future is "adopted" by the new parent. Thus, calling `flux_future_destroy(3)` on the parent composite will destroy all children. Therefore, child futures that have been the target of `flux_future_push(3)` should **not** have flux_future_destroy(3)‘ called upon them to avoid double-free.

`flux_future_first_child(3)` and `flux_future_next_child(3)` are used to iterate over child future names in a composite future created with either `flux_future_wait_all_create(3)` or `flux_future_wait_any_create(3)`. The `flux_future_t` corresponding to the returned *name* can be then fetched with `flux_future_get_child(3)`. `flux_future_next_child` will return a `NULL` once all children have been iterated.

`flux_future_get_child(3)` retrieves a child future from a composite by name.

RETURN VALUE

`flux_future_wait_any_create()` and `flux_future_wait_all_create()` return a future on success. On error, NULL is returned and errno is set appropriately.

`flux_future_push()` returns zero on success. On error, -1 is returned and errno is set appropriately.

`flux_future_first_child()` returns the name of the first child future in the targeted composite in no given order. If the composite is empty, a NULL is returned.

`flux_future_next_child()` returns the name of the next child future in the targeted composite in no given order. If the last child has already been returned then this function returns NULL.

`flux_future_get_child()` returns a `flux_future_t` corresponding to the child future with the supplied string `name` parameter. If no future with that name is a child of the composite, then the function returns NULL.

ERRORS

ENOMEM Out of memory.

EINVAL Invalid argument.

ENOENT The requested object is not found.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_future_get(3), flux_future_create(3)

1.2.16 flux_get_rank(3)

SYNOPSIS

#include <flux/core.h>

int flux_get_rank (flux_t *h, uint32_t *rank);

int flux_get_size (flux_t *h, uint32_t *size);

DESCRIPTION

`flux_get_rank()` and `flux_get_size()` ask the Flux broker for its rank in the Flux instance, and the size of the Flux instance.

Ranks are numbered 0 through size - 1.

RETURN VALUE

These functions return zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL Some arguments were invalid.

EXAMPLES

Example:

```
#include <math.h>
#include <flux/core.h>
#include <inttypes.h>
#include "src/common/libutil/log.h"

int main (int argc, char **argv)
{
    flux_t *h;
    uint32_t rank, n;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("flux_open");
    if (flux_get_rank (h, &rank) < 0)
        log_err_exit ("flux_get_rank");
    if (flux_get_size (h, &n) < 0)
        log_err_exit ("flux_get_size");
    flux_close (h);
    return (0);
}
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

RFC 3: CMB1 - Flux Comms Message Broker Protocol

1.2.17 flux_get_reactor(3)

SYNOPSIS

#include <flux/core.h>

flux_reactor_t *flux_get_reactor (flux_t *h);

int flux_set_reactor (flux_t *h, flux_reactor_t *r);

DESCRIPTION

`flux_get_reactor()` retrieves a flux_reactor_t object previously associated with the broker handle *h* by a call to `flux_set_reactor()`. If one has not been previously associated, a flux_reactor_t object is created on demand. If the flux_reactor_t object is created on demand, it will be destroyed when the handle is destroyed, otherwise it is the responsibility of the owner to destroy it after the handle is destroyed.

`flux_set_reactor()` associates a flux_reactor_t object *r* with a broker handle *h*. A flux_reactor_t object may be obtained from another handle, for example when events from multiple handles are to be managed using a common flux_reactor_t, or one may be created directly with `flux_reactor_create(3)`. `flux_set_reactor()` should be called immediately after `flux_open(3)` to avoid conflict with other API calls which may internally call `flux_get_reactor()`.

RETURN VALUE

`flux_get_reactor()` returns a flux_reactor_t object on success. On error, NULL is returned, and errno is set appropriately.

`flux_set_reactor()` returns 0 on success, or -1 on failure with errno set appropriately.

ERRORS

ENOMEM Out of memory.

EEXIST Handle already has a reactor association.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_reactor_create(3), flux_reactor_destroy(3)

1.2.18 flux_handle_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
flux_watcher_t *flux_handle_watcher_create (flux_reactor_t *r,
                                            flux_t *h, int events,
                                            flux_watcher_f callback,
                                            void *arg);
```

```
flux_t *flux_handle_watcher_get_flux (flux_watcher_t *w);
```

DESCRIPTION

`flux_handle_watcher_create()` creates a flux_watcher_t object which monitors for events on a Flux broker handle *h*. When events occur, the user-supplied *callback* is invoked.

The *events* and *revents* arguments are a bitmask containing a logical OR of the following bits. If a bit is set in *events*, it indicates interest in this type of event. If a bit is set in *revents*, it indicates that this event has occurred.

FLUX_POLLIN The handle is ready for reading.

FLUX_POLLOUT The handle is ready for writing.

FLUX_POLLERR The handle has encountered an error. This bit is ignored if it is set in *events*.

Events are processed in a level-triggered manner. That is, the callback will continue to be invoked as long as the event has not been fully consumed or cleared, and the watcher has not been stopped.

`flux_handle_watcher_get_flux()` is used to obtain the handle from within the callback.

RETURN VALUE

`flux_handle_watcher_create()` returns a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

`flux_handle_watcher_get_flux()` returns the handle associated with the watcher.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3), flux_recv(3), flux_send(3).

1.2.19 flux_idle_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
flux_watcher_t *flux_prepare_watcher_create (flux_reactor_t *r,
                                             flux_watcher_f callback,
                                             void *arg);
```

```
flux_watcher_t *flux_check_watcher_create (flux_reactor_t *r,
                                           flux_watcher_f callback,
                                           void *arg);
```

```
flux_watcher_t *flux_idle_watcher_create (flux_reactor_t *r,
                                          flux_watcher_f callback,
                                          void *arg);
```

DESCRIPTION

`flux_prepare_watcher_create()`, `flux_check_watcher_create()`, and `flux_idle_watcher_create()` create specialized reactor watchers with the following properties:

The prepare watcher is called by the reactor loop immediately before blocking, while the check watcher is called by the reactor loop immediately after blocking.

The idle watcher is always run when no other events are pending, excluding other idle watchers, prepare and check watchers. While it is active, the reactor loop does not block waiting for new events.

The callback *revents* argument should be ignored.

Note: the Flux reactor is based on libev. For additional information on the behavior of these watchers, refer to the libev documentation on `ev_idle`, `ev_prepare`, and `ev_check`.

RETURN VALUE

These functions return a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3)

libev home page

1.2.20 flux_kvs_commit(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_kvs_commit (flux_t *h,
                                const char *ns,
                                int flags,
                                flux_kvs_txn_t *txn);
```

```
flux_future_t *flux_kvs_fence (flux_t *h,
                               const char *ns,
                               int flags,
                               const char *name,
                               int nprocs,
                               flux_kvs_txn_t *txn);
```

```
int flux_kvs_commit_get_treeobj (flux_future_t *f,
                                 const char **treeobj);
```

```
int flux_kvs_commit_get_sequence (flux_future_t *f,
                                  int *seq);
```

DESCRIPTION

`flux_kvs_commit()` sends a request via handle *h* to the KVS service to commit a transaction *txn*. *txn* is created with `flux_kvs_txn_create(3)` and after commit completion, is destroyed with `flux_kvs_txn_destroy()`. A `flux_future_t` object is returned, which acts as handle for synchronization and container for the response. The *txn* will operate in the namespace specified by *ns*. If *ns* is NULL, `flux_kvs_commit()` will operate on the default namespace, or if set, the namespace from the FLUX_KVS_NAMESPACE environment variable. Note that all transactions operate on the same namespace.

`flux_kvs_fence()` is a "collective" version of `flux_kvs_commit()` that supports multiple callers. Each caller uses the same *flags*, *name*, and *nprocs* arguments. Once *nprocs* requests are received by the KVS service for the named operation, the transactions are combined and committed together as one transaction. *name* must be unique across the Flux session and should not be reused, even after the fence is complete.

`flux_future_then(3)` may be used to register a reactor callback (continuation) to be called once the response to the commit/fence request has been received. `flux_future_wait_for(3)` may be used to block until the response has been received. Both accept an optional timeout.

`flux_future_get()`, `flux_kvs_commit_get_treeobj()`, or `flux_kvs_commit_get_sequence()` can decode the response. A return of 0 indicates success and the entire transaction was committed. A return of -1 indicates failure, none of the transaction was committed. All can be used on the `flux_future_t` returned by `flux_kvs_commit()` or `flux_kvs_fence()`.

In addition to checking for success or failure, `flux_kvs_commit_get_treeobj()` and `flux_kvs_commit_get_sequence()` can return information about the root snapshot that the commit or fence has completed its transaction on.

`flux_kvs_commit_get_treeobj()` obtains the root hash in the form of an RFC 11 *dirref* treeobj, suitable to be passed to `flux_kvs_lookupat(3)`.

`flux_kvs_commit_get_sequence()` retrieves the monotonic sequence number for the root.

FLAGS

The following are valid bits in a *flags* mask passed as an argument to `flux_kvs_commit()` or `flux_kvs_fence()`.

FLUX_KVS_NO_MERGE The KVS service may merge contemporaneous commit transactions as an optimization. However, if the combined transactions modify the same key, a watch on that key may only be notified of the last-in value. This flag can be used to disable that optimization for this transaction.

RETURN VALUE

`flux_kvs_commit()` and `flux_kvs_fence()` return a `flux_future_t` on success, or NULL on failure with errno set appropriately.

ERRORS

EINVAL One of the arguments was invalid.

ENOMEM Out of memory.

EPROTO A request was malformed.

ENOSYS The KVS module is not loaded.

ENOTSUP An unknown namespace was requested.

EOVERFLOW `flux_kvs_fence()` has been called too many times and *nprocs* has been exceeded.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_future_get(3), flux_kvs_txn_create(3), flux_kvs_set_namespace(3)

1.2.21 flux_kvs_copy(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_kvs_copy (flux_t *h,
                              const char *srckey,
                              const char *dstkey,
                              int commit_flags);
```

```
flux_future_t *flux_kvs_move (flux_t *h,
                              const char *srckey,
                              const char *dstkey,
                              int commit_flags);
```

DESCRIPTION

`flux_kvs_copy()` sends a request via handle *h* to the KVS service to look up the directory entry of *srckey*. Upon receipt of the response, it then sends another request to commit a duplicate at *dstkey*. *commit_flags* are passed through to the commit operation. See the FLAGS section of flux_kvs_commit(3).

The net effect is that all content below *srckey* is copied to *dstkey*. Due to the hash tree organization of the KVS name space, only the directory entry needs to be duplicated to create a new, fully independent deep copy of the original data.

`flux_kvs_move()` first performs a `flux_kvs_copy()`, then sends a commit request to unlink *srckey*. *commit_flags* are passed through to the commit within `flux_kvs_copy()`, and to the commit which performs the unlink.

`flux_kvs_copy()` and `flux_kvs_move()` are capable of working across namespaces. See `flux_kvs_commit(3)` for info on how to select a namespace other than the default.

CAVEATS

`flux_kvs_copy()` and `flux_kvs_commit()` are implemented as aggregates of multiple KVS operations. As such they do not have the "all or nothing" guarantee of a being carried out within a single KVS transaction.

In the unlikely event that the copy phase of a `flux_kvs_move()` succeeds but the unlink phase fails, `flux_kvs_move()` may return failure without cleaning up the new copy. Since the copy phase already validated that the unlink target key exists by copying from it, the source of such a failure would be a transient error such as out of memory or communication failure.

RETURN VALUE

`flux_kvs_copy ()` and `flux_kvs_move ()` return a `flux_future_t` on success, or NULL on failure with errno set appropriately.

ERRORS

EINVAL One of the arguments was invalid.

ENOMEM Out of memory.

EPROTO A request was malformed.

ENOSYS The KVS module is not loaded.

ENOTSUP An unknown namespace was requested.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_future_get(3), flux_kvs_commit(3)

1.2.22 flux_kvs_getroot(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_kvs_getroot (flux_t *h,
                                 const char *ns,
                                 int flags);
```

```
int flux_kvs_getroot_get_treeobj (flux_future_t *f,
                                  const char **treeobj);
```

```
int flux_kvs_getroot_get_blobref (flux_future_t *f,
                                  const char **blobref);
```

```
int flux_kvs_getroot_get_sequence (flux_future_t *f,
                                   int *seq);
```

```
int flux_kvs_getroot_get_owner (flux_future_t *f,
                                uint32_t *owner);
```

DESCRIPTION

`flux_kvs_getroot()` sends a request via handle *h* to the `kvs` service to look up the current root hash for namespace *ns*. A `flux_future_t` object is returned, which acts as handle for synchronization and container for the response. *flags* is currently unused and should be set to 0.

Upon future fulfillment, these functions can decode the result:

`flux_kvs_getroot_get_treeobj()` obtains the root hash in the form of an RFC 11 *dirref* treeobj, suitable to be passed to `flux_kvs_lookupat(3)`.

`flux_kvs_getroot_get_blobref()` obtains the RFC 10 blobref, suitable to be passed to `flux_content_load(3)`.

`flux_kvs_getroot_get_sequence()` retrieves the monotonic sequence number for the root.

`flux_kvs_getroot_get_owner()` retrieves the namespace owner.

FLAGS

The *flags* mask is currently unused and should be set to 0.

RETURN VALUE

`flux_kvs_getroot()` returns a `flux_future_t` on success, or NULL on failure with errno set appropriately.

The other functions return zero on success, or -1 on failure with errno set appropriately.

ERRORS

EINVAL One of the arguments was invalid.

ENOMEM Out of memory.

EPROTO A request was malformed.

ENOSYS The kvs module is not loaded.

ENOTSUP An unknown namespace was requested or namespace was deleted.

EPERM The requesting user is not permitted to access the requested namespace.

ENODATA A stream of responses has been terminated by a call to `flux_kvs_getroot_cancel()`.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_kvs_lookup (3), flux_future_get (3), flux_content_load (3).

1.2.23 flux_kvs_lookup(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_kvs_lookup (flux_t *h, const char *ns, int flags,
                                const char *key);
```

```
flux_future_t *flux_kvs_lookupat (flux_t *h, int flags,
                                  const char *key, const char *treeobj);
```

```
int flux_kvs_lookup_get (flux_future_t *f, const char **value);
```

```
int flux_kvs_lookup_get_unpack (flux_future_t *f, const char *fmt, ...);
```

```
int flux_kvs_lookup_get_raw (flux_future_t *f,
                             const void **data, int *len);
```

```
int flux_kvs_lookup_get_dir (flux_future_t *f,
                             const flux_kvsdir_t **dir);
```

```
int flux_kvs_lookup_get_treeobj (flux_future_t *f, const char **treeobj);
```

```
int flux_kvs_lookup_get_symlink (flux_future_t *f, const char **ns,
                                 const char **target);
```

```
const char *flux_kvs_lookup_get_key (flux_future_t *f);
```

```
int flux_kvs_lookup_cancel (flux_future_t *f);
```

DESCRIPTION

The Flux Key Value Store is a general purpose distributed storage service used by Flux services.

`flux_kvs_lookup()` sends a request to the KVS service to look up *key* in namespace *ns*. It returns a `flux_future_t` object which acts as handle for synchronization and container for the result. The namespace *ns* is optional. If set to NULL, `flux_kvs_lookup()` uses the default namespace, or if set, the namespace from the FLUX_KVS_NAMESPACE environment variable. *flags* modifies the request as described below.

`flux_kvs_lookupat()` is identical to `flux_kvs_lookup()` except *treeobj* is a serialized RFC 11 object that references a particular static set of content within the KVS, effectively a snapshot. See `flux_kvs_lookup_get_treeobj()` below.

All the functions below are variations on a common theme. First they complete the lookup RPC by blocking on the response, if not already received. Then they interpret the result in different ways. They may be called more than once

on the same future, and they may be intermixed to probe a result or interpret it in different ways. Results remain valid until `flux_future_destroy()` is called.

`flux_kvs_lookup_get()` interprets the result as a value. If the value has length greater than zero, a NULL is appended and it is assigned to *value*, otherwise NULL is assigned to *value*.

`flux_kvs_lookup_get_unpack()` interprets the result as a value, which it decodes as JSON according to variable arguments in Jansson `json_unpack()` format.

`flux_kvs_lookup_get_raw()` interprets the result as a value. If the value has length greater than zero, the value and its length are assigned to *buf* and *len*, respectively. Otherwise NULL and zero are assigned.

`flux_kvs_lookup_get_dir()` interprets the result as a directory, e.g. in response to a lookup with the FLUX_KVS_READDIR flag set. The directory object is assigned to *dir*.

`flux_kvs_lookup_get_treeobj()` interprets the result as any RFC 11 object. The object in JSON-encoded form is assigned to *treeobj*. Since all lookup requests return an RFC 11 object of one type or another, this function should work on all.

`flux_kvs_lookup_get_symlink()` interprets the result as a symlink target, e.g. in response to a lookup with the FLUX_KVS_READLINK flag set. The result is parsed and symlink namespace is assigned to *ns* and the symlink target is assigned to *target*. If a namespace was not assigned to the symlink, *ns* is set to NULL.

`flux_kvs_lookup_get_key()` accesses the key argument from the original lookup.

`flux_kvs_lookup_cancel()` cancels a stream of lookup responses requested with FLUX_KVS_WATCH or a waiting lookup response with FLUX_KVS_WAITCREATE. See FLAGS below for additional information.

These functions may be used asynchronously. See `flux_future_then(3)` for details.

FLAGS

The following are valid bits in a *flags* mask passed as an argument to `flux_kvs_lookup()` or `flux_kvs_lookupat()`.

FLUX_KVS_READDIR Look up a directory, not a value. The lookup fails if the key does not refer to a directory object.

FLUX_KVS_READLINK If key is a symlink, read the link value. The lookup fails if the key does not refer to a symlink object.

FLUX_KVS_TREEOBJ All KVS lookups return an RFC 11 tree object. This flag requests that they be returned without conversion. That is, a "valref" will not be converted to a "val" object, and a "dirref" will not be converted to a "dir" object. This is useful for obtaining a snapshot reference that can be passed to `flux_kvs_lookupat()`.

FLUX_KVS_WATCH After the initial response, continue to send responses to the lookup request each time *key* is mentioned verbatim in a committed transaction. After receiving a response, `flux_future_reset()` should be used to consume a response and prepare for the next one. Responses continue until the namespace is removed, the key is removed, the lookup is canceled with `flux_kvs_lookup_cancel()`, or an error occurs. After calling `flux_kvs_lookup_cancel()`, callers should wait for the future to be fulfilled with an ENODATA error to ensure the cancel request has been received and processed.

FLUX_KVS_WATCH_UNIQ Specified along with FLUX_KVS_WATCH, this flag will alter watch behavior to only respond when *key* is mentioned verbatim in a committed transaction and the value of the key has changed.

FLUX_KVS_WATCH_APPEND Specified along with FLUX_KVS_WATCH, this flag will alter watch behavior to only respond when *key* is mentioned verbatim in a committed transaction and the key has been appended to. The response will only contain the additional appended data. Note that only data length is considered for appends and no guarantee is made that prior data hasn't been overwritten.

FLUX_KVS_WATCH_FULL Specified along with FLUX_KVS_WATCH, this flag will alter watch behavior to respond when the value of the key being watched has changed. Unlike FLUX_KVS_WATCH_UNIQ, the key being watched need not be mentioned in a transaction. This may occur under several scenarios, such as a parent directory being altered.

FLUX_KVS_WAITCREATE If a KVS key does not exist, wait for it to exist before returning. This flag can be specified with or without FLUX_KVS_WATCH. The lookup can be canceled with `flux_kvs_lookup_cancel()`. After calling `flux_kvs_lookup_cancel()`, callers should wait for the future to be fulfilled with an ENODATA error to ensure the cancel request has been received and processed.

RETURN VALUE

`flux_kvs_lookup()` and `flux_kvs_lookupat()` return a `flux_future_t` on success, or NULL on failure with errno set appropriately.

`flux_kvs_lookup_get()`, `flux_kvs_lookup_get_unpack()`, `flux_kvs_lookup_get_raw()`, `flux_kvs_lookup_get_dir()`, `flux_kvs_lookup_get_treeobj()`, `flux_kvs_lookup_get_symlink()`, and `flux_kvs_lookup_cancel()` return 0 on success, or -1 on failure with errno set appropriately.

`flux_kvs_lookup_get_key()` returns key on success, or NULL with errno set to EINVAL if its future argument did not come from a KVS lookup.

ERRORS

EINVAL One of the arguments was invalid, or FLUX_KVS_READLINK was used but the key does not refer to a symlink.

ENOMEM Out of memory.

ENOENT An unknown key was requested.

ENOTDIR FLUX_KVS_READDIR flag was set and key does NOT point to a directory.

EISDIR FLUX_KVS_READDIR flag was NOT set and key points to a directory.

EPROTO A request or response was malformed.

EFBIG; ENOSYS The KVS module is not loaded.

ENOTSUP An unknown namespace was requested or namespace was deleted.

ENODATA A stream of responses requested with FLUX_KVS_WATCH was terminated with `flux_kvs_lookup_cancel()`.

EPERM The user does not have instance owner capability, and a lookup was attempted against a KVS namespace owned by another user.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_rpc(3), flux_future_then(3), flux_kvs_set_namespace(3)

RFC 11: Key Value Store Tree Object Format v1

1.2.24 flux_kvs_namespace_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_kvs_namespace_create (flux_t *h,
                                          const char *namespace,
                                          uint32_t owner,
                                          int flags);
```

```
flux_future_t *flux_kvs_namespace_remove (flux_t *h,
                                          const char *namespace);
```

DESCRIPTION

`flux_kvs_namespace_create()` creates a KVS namespace. Within a namespace, users can get/put KVS values completely independent of other KVS namespaces. An owner of the namespace other than the instance owner can be chosen by setting *owner*. Otherwise, *owner* can be set to FLUX_USERID_UNKNOWN.

`flux_kvs_namespace_remove()` removes a KVS namespace.

FLAGS

The *flags* mask is currently unused and should be set to 0.

RETURN VALUE

`flux_kvs_namespace_create()` and `flux_kvs_namespace_remove()` return a `flux_future_t` on success, or NULL on failure with errno set appropriately.

ERRORS

EINVAL One of the arguments was invalid.

ENOMEM Out of memory.

EPROTO A request was malformed.

ENOSYS The KVS module is not loaded.

EEXIST The namespace already exists.

ENOTSUP Attempt to remove illegal namespace.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_kvs_lookup(3), flux_kvs_commit(3)

1.2.25 flux_kvs_txn_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_kvs_txn_t *flux_kvs_txn_create (void);
```

```
void flux_kvs_txn_destroy (flux_kvs_txn_t *txn);
```

```
int flux_kvs_txn_put (flux_kvs_txn_t *txn, int flags,
                      const char *key, const char *value);
```

```
int flux_kvs_txn_pack (flux_kvs_txn_t *txn, int flags,
                       const char *key, const char *fmt, ...);
```

```
int flux_kvs_txn_vpack (flux_kvs_txn_t *txn, int flags,
                        const char *key, const char *fmt, va_list ap);
```

```
int flux_kvs_txn_mkdir (flux_kvs_txn_t *txn, int flags,
                        const char *key);
```

```
int flux_kvs_txn_unlink (flux_kvs_txn_t *txn, int flags,
                         const char *key);
```

```
int flux_kvs_txn_symlink (flux_kvs_txn_t *txn, int flags,
                          const char *key, const char *ns,
                          const char *target);
```

```
int flux_kvs_txn_put_raw (flux_kvs_txn_t *txn, int flags,
                          const char *key, const void *data, int len);
```

```
int flux_kvs_txn_put_treeobj (flux_kvs_txn_t *txn, int flags,
                              const char *key, const char *treeobj);
```

DESCRIPTION

The Flux Key Value Store is a general purpose distributed storage service used by Flux services.

`flux_kvs_txn_create()` creates a KVS transaction object that may be passed to `flux_kvs_commit(3)` or `flux_kvs_fence(3)`. The transaction consists of a list of operations that are applied to the KVS together, in order. The entire transaction either succeeds or fails. After commit or fence, the object must be destroyed with `flux_kvs_txn_destroy()`.

Each function below adds a single operation to *txn*. *key* is a hierarchical path name with period (".") used as path separator. When the transaction is committed, any existing keys or path components that are in conflict with the requested operation are overwritten. *flags* can modify the request as described below.

`flux_kvs_txn_put()` sets *key* to a NULL terminated string *value*. *value* may be NULL indicating that an empty value should be stored.

`flux_kvs_txn_pack()` sets *key* to a NULL terminated string encoded from a JSON object built with `json_pack()` style arguments (see below). `flux_kvs_txn_vpack()` is a variant that accepts a *va_list* argument.

`flux_kvs_txn_mkdir()` sets *key* to an empty directory.

`flux_kvs_txn_unlink()` removes *key*. If *key* is a directory, all its contents are removed as well.

`flux_kvs_txn_symlink()` sets *key* to a symbolic link pointing to a namespace *ns* and a *target* key within that namespace. Neither *ns* nor *target* must exist. The namespace *ns* is optional, if set to NULL the *target* is assumed to be in the key's current namespace.

`flux_kvs_txn_put_raw()` sets *key* to a value containing raw data referred to by *data* of length *len*.

`flux_kvs_txn_put_treeobj()` sets *key* to an RFC 11 object, encoded as a JSON string.

FLAGS

The following are valid bits in a *flags* mask passed as an argument to `flux_kvs_txn_put()` or `flux_kvs_txn_put_raw()`.

FLUX_KVS_APPEND Append value instead of overwriting it. If the key does not exist, it will be created with the value as the initial value.

ENCODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_pack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a null terminated UTF-8 string to a JSON string.

s# (string)['const char *', 'int'] Convert a UTF-8 buffer of a given length to a JSON string.

s% (string)['const char *', 'size_t'] Like **s#** but the length argument is of type size_t.

+ ['const char *'] Like **s**, but concatenate to the previous string. Only valid after a string.

+# ['const char *', 'int'] Like **s#**, but concatenate to the previous string. Only valid after a string.

+% ['const char *', 'size_t'] Like **+#**, but the length argument is of type size_t.

n (null) Output a JSON null value. No argument is consumed.

b (boolean)['int'] Convert a C int to JSON boolean value. Zero is converted to *false* and non-zero to *true*.

i (integer)['int'] Convert a C int to JSON integer.

I (integer)['int64_t'] Convert a C int64_t to JSON integer. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert a C double to JSON real.

o (any value)['json_t *'] Output any given JSON value as-is. If the value is added to an array or object, the reference to the value passed to **o** is stolen by the container.

O (any value)['json_t *'] Like **o**, but the argument's reference count is incremented. This is useful if you pack into an array or object and want to keep the reference for the JSON value consumed by **O** to yourself.

[fmt] (array) Build an array with contents from the inner format string. **fmt** may contain objects and arrays, i.e. recursive value building is supported.

{fmt} (object) Build an object with contents from the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must be a string as object keys are always strings. The second, fourth, etc. format specifier represent a value. Any value may be an object or array, i.e. recursive value building is supported.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Building Values

RETURN VALUE

`flux_kvs_txn_create()` returns a `flux_kvs_txn_t` object on success, or NULL on failure with errno set appropriately.

`flux_kvs_txn_put()`, `flux_kvs_txn_pack()`, `flux_kvs_txn_mkdir()`, `flux_kvs_txn_unlink()`, `flux_kvs_txn_symlink()`, and `flux_kvs_txn_put_raw()` returns 0 on success, or -1 on failure with errno set appropriately.

ERRORS

EINVAL One of the arguments was invalid.

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_kvs_commit(3)

RFC 11: Key Value Store Tree Object Format v1

1.2.26 flux_log(3)

SYNOPSIS

#include <flux/core.h>

int flux_vlog (flux_t *h, int level, const char *fmt, va_list ap);

int flux_log (flux_t *h, int level, const char *fmt, . . .);

void flux_log_set_appname (flux_t *h, const char *s);

void flux_log_set_procid (flux_t *h, const char *s);

DESCRIPTION

`flux_log()` creates RFC 5424 format log messages. The log messages are sent to the Flux message broker on *h* for handling if it is specified. If *h* is NULL, the log message is output to stderr.

The *level* parameter should be set to one of the syslog(3) severity levels, which are, in order of decreasing importance:

LOG_EMERG system is unusable

LOG_ALERT action must be taken immediately

LOG_CRIT critical conditions

LOG_ERR error conditions

LOG_WARNING warning conditions

LOG_NOTICE normal, but significant, condition

LOG_INFO informational message

LOG_DEBUG debug-level message

When *h* is specified, log messages are are added to the broker's circular buffer which can be accessed with flux-dmesg(3). From there, a message's disposition is up to the broker's log configuration.

`flux_log_set_procid()` may be used to override the default procid, which is initialized to the calling process's PID.

`flux_log_set_appname()` may be used to override the default application name, which is initialized to the value of the *__progname* symbol (normally the argv[0] program name).

MAPPING TO SYSLOG

A Flux log message is formatted as a Flux request with a "raw" payload, as defined by Flux RFC 3. The raw payload is formatted according to Internet RFC 5424.

If the Flux handle *h* is specified, the following Syslog header fields are set in a Flux log messages when it is created within `flux_log()`:

PRI Set to the user-specified severity level combined with the facility, which is hardwired to *LOG_USER* in Flux log messages.

VERSION Set to 1.

TIMESTAMP Set to the current UTC wallclock time.

HOSTNAME Set to the broker rank associated with *h*.

APP-NAME Set to the user-defined application name, truncated to 48 characters, excluding terminating NULL.

PROCID Set to the PID of the calling process.

MSGID Set to the NIL string "-".

The STRUCTURED-DATA portion of the message is empty, and reserved for future use by Flux.

The MSG portion is post-processed to ensure it contains no NULL's or non-ASCII characters. At this time non-ASCII UTF-8 is not supported by `flux_log()`.

RETURN VALUE

`flux_log()` normally returns 0 on success, or -1 if there was a problem building or sending the log message, with errno set.

ERRORS

EPERM The user does not have permission to log messages to this Flux instance.

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-dmesg(1), flux-logger(1), RFC 5424 The Syslog Protocol

1.2.27 flux_msg_cmp(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
struct flux_match {
    int typemask;
    uint32_t matchtag;
    char *topic_glob;
};
```

```
bool flux_msg_cmp (const flux_msg_t *msg, struct flux_match match);
```

DESCRIPTION

`flux_msg_cmp()` compares *msg* to *match* criteria.

If *match.typemask* is nonzero, the type of the message must match one of the types in the mask.

If *match.matchtag* is not FLUX_MATCHTAG_NONE, the message matchtag must match *match.matchtag*.

If *match.topic_glob* is not NULL or an empty string, then the message topic string must match *match.topic_glob* according to the rules of shell wildcards.

RETURN VALUE

`flux_msg_cmp()` returns true on a match, otherwise false.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

fnmatch(3)

1.2.28 flux_msg_encode(3)

SYNOPSIS

#include <flux/core.h>

int flux_msg_encode (const flux_msg_t *msg, void **buf, size_t *size);

flux_msg_t *flux_msg_decode (void *buf, size_t size);

DESCRIPTION

`flux_msg_encode()` converts *msg* to a serialized representation, allocated internally and assigned to *buf*, number of bytes to *size*. The caller must release *buf* with free(3).

`flux_msg_decode()` performs the inverse, creating *msg* from *buf* and *size*. The caller must destroy *msg* with flux_msg_destroy().

RETURN VALUE

`flux_msg_encode()` returns 0 on success. On error, -1 is returned, and errno is set appropriately.

`flux_msg_decode()` the decoded message on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

EINVAL Some arguments were invalid.

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

1.2.29 flux_msg_handler_addvec(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
struct flux_msg_handler_spec {
    int typemask;
    const char *topic_glob;
    flux_msg_handler_f cb;
    uint32_t rolemask;
};
```

```
int flux_msg_handler_addvec (flux_t *h,
                             const struct flux_msg_handler_spec tab[],
                             void *arg,
                             flux_msg_handler_t **handlers[]);
```

```
void flux_msg_handler_delvec (flux_msg_handler_t *handlers[]);
```

DESCRIPTION

`flux_msg_handler_addvec()` creates and starts an array of message handlers, terminated by FLUX_MSGHANDLER_TABLE_END. The new message handler objects are assigned to an internally allocated array, returned in *handlers*. The last entry in the array is set to NULL.

`flux_msg_handler_delvec()` stops and destroys an array of message handlers returned from `flux_msg_handler_addvec()`.

These functions are convenience functions which call `flux_msg_handler_create(3)`, `flux_msg_handler_start(3)`; and `flux_msg_handler_stop(3)`, `flux_msg_handler_destroy(3)` on each element of the array, respectively.

If `flux_msg_handler_addvec()` encounters an error creating a message handler, all previously created message handlers in the array are destroyed before an error is returned.

RETURN VALUE

`flux_msg_handler_addvec()` returns zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_msg_handler_create(3)

1.2.30 flux_msg_handler_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_msg_handler_f)(flux_t *h,
                                   flux_msg_handler_t *mh,
                                   const flux_msg_t *msg,
                                   void *arg);
```

```
flux_msg_handler_t *
flux_msg_handler_create (flux_t *h,
                         const struct flux_match match,
                         flux_msg_handler_f callback,
                         void *arg);
```

```
void flux_msg_handler_destroy (flux_msg_handler_t *mh);
```

```
void flux_msg_handler_start (flux_msg_handler_t *mh);
```

```
void flux_msg_handler_stop (flux_msg_handler_t *mh);
```

DESCRIPTION

`flux_msg_handler_create()` registers *callback* to be invoked when a message meeting *match* criteria, as described in `flux_msg_cmp(3)`, is received on Flux broker handle *h*.

The message handler must be started with `flux_msg_handler_start()` in order to receive messages. Conversely, `flux_msg_handler_stop()` causes the message handler to stop receiving messages. Starting and stopping are idempotent operations.

The handle *h* is monitored for FLUX_POLLIN events on the flux_reactor_t associated with the handle as described in `flux_set_reactor(3)`. This internal "handle watcher" is started when the first message handler is started, and stopped when the last message handler is stopped.

Messages arriving on *h* are internally read and dispatched to matching message handlers. If multiple handlers match the message, the following rules apply:

FLUX_MSGTYPE_REQUEST Requests are first delivered to a message handler whose match.topic_glob is set to an exact string match of the message topic glob. The most recently registered of these takes precedence. If an exact match is unavailable, the message is delivered to the most recently registered message handler for which `flux_msg_cmp()` returns true. If there is no match, an ENOSYS response is automatically generated by the dispatcher.

FLUX_MSGTYPE_RESPONSE Responses are first delivered to a matching RPC response handler (match.matchtag != FLUX_MATCHTAG_NONE). If an RPC response handler does not match, responses are delivered to the most recently registered message handler for which `flux_msg_cmp()` returns true. If there is no match, the response is discarded.

FLUX_MSGTYPE_EVENT Events are delivered to *all* matching message handlers.

`flux_msg_handler_destroy()` destroys a handler, after internally stopping it.

CAVEATS

Although it is possible to register a message handler in a given *flux_t* handle for any topic string, `flux-broker(1)` does not automatically route matching requests or events to the handle.

Requests are only routed if the handle has registered a matching service with `flux_service_register(3)`, or for broker modules only, the service matches the module name.

Events are only routed if the topic matches a subscription registered with `flux_event_subscribe(3)`.

RETURN VALUE

`flux_msg_handler_create()` returns a flux_msg_handler_t object on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_get_reactor(3), flux_reactor_start(3), flux_msg_cmp(3)

1.2.31 flux_open(3)

SYNOPSIS

#include <flux/core.h>

flux_t *flux_open (const char *uri, int flags);

void flux_close (flux_t *h);

flux_t *flux_clone (flux_t *orig);

DESCRIPTION

`flux_open()` creates a `flux_t` handle, used to communicate with the Flux message broker.

The *uri* scheme (before "://") specifies the "connector" that will be used to establish the connection. The *uri* path (after "://") is parsed by the connector. If *uri* is NULL, the value of $FLUX_URI is used, if set.

flags is the logical "or" of zero or more of the following flags:

FLUX_O_TRACE Dumps message trace to stderr.

FLUX_O_MATCHDEBUG Prints diagnostic to stderr when matchtags are leaked, for example when a streaming RPC is destroyed without receiving a error response as end-of-stream, or a regular RPC is destroyed without being fulfilled at all.

FLUX_O_NONBLOCK The `flux_send()` and `flux_recv()` functions should never block.

`flux_clone()` creates another reference to a `flux_t` handle that is identical to the original in all respects except that it does not inherit a copy of the original handle's "aux" hash, or its reactor and message dispatcher references. By creating a clone, and calling `flux_set_reactor()` on it, one can create message handlers on the cloned handle that run on a different reactor than the one associated with the original handle.

`flux_close()` destroys a `flux_t` handle, closing its connection with the Flux message broker.

RETURN VALUE

`flux_open()` and `flux_clone()` return a `flux_t` handle on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

EINVAL *uri* was NULL and $FLUX_URI was not set, or other arguments were invalid.

ENOMEM Out of memory.

EXAMPLES

This example opens the Flux broker using the default connector and path, requests the broker rank, and finally closes the broker handle.

```
#include <inttypes.h>
#include <flux/core.h>
#include "src/common/libutil/log.h"

int main (int argc, char **argv)
{
    flux_t *h;
    uint32_t rank;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("flux_open");
    if (flux_get_rank (h, &rank) < 0)
        log_err_exit ("flux_get_rank");
    printf ("My rank is %"PRIu32"\n", rank);
    flux_close (h);
    return (0);
}
```

RESOURCES

Github: http://github.com/flux-framework

1.2.32 flux_periodic_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
typedef double (*flux_reschedule_f) (flux_watcher_t *w, double now, void *arg);
```

```
flux_watcher_t *flux_periodic_watcher_create (flux_reactor_t *r,
                                              double offset, double interval,
                                              flux_reschedule_f reschedule_cb,
                                              flux_watcher_f callback,
                                              void *arg);
```

```
void flux_periodic_watcher_reset (flux_watcher_t *w,
                                  double offset, double interval);
```

DESCRIPTION

`flux_periodic_watcher_create()` creates a flux_watcher_t object which monitors for periodic events. The periodic watcher will trigger when the wall clock time *offset* has elapsed, and optionally again ever *interval* of wall clock time thereafter. If the *reschedule_cb* parameter is used, then *offset* and *interval* are ignored, and instead each time the periodic watcher is scheduled the reschedule callback will be called with the current time, and is expected to return the next absolute time at which the watcher should be scheduled.

Unlike timer events, a periodic watcher monitors wall clock or system time, not the actual time that passes. Thus, a periodic watcher can be used to run a callback when system time reaches a certain point. For example, if a periodic watcher is set to run with an *offset* of 10 seconds, and then system time is set back by 1 hour, it will take approximately 1 hour, 10 seconds for the watcher to execute.

If a periodic watcher is running in manual reschedule mode (*reschedule_cb* is non-NULL), and the user-provided reschedule callback returns a time that is before the current time, the watcher will be silently stopped.

Note: the Flux reactor is based on libev. For additional important information on the behavior of periodic, refer to the libev documentation on `ev_periodic`.

RETURN VALUE

`flux_periodic_watcher_create()` returns a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3), flux_timer_watcher_create(3)

libev home page

1.2.33 flux_pollevents(3)

SYNOPSIS

#include <flux/core.h>

int flux_pollevents (flux_t *h);

int flux_pollfd (flux_t *h);

DESCRIPTION

`flux_pollevents()` returns a bitmask of poll flags for handle *h*.

`flux_pollfd()` obtains a file descriptor that becomes readable, in an edge triggered fashion, when `flux_pollevents()` has poll flags raised.

Valid poll flags are:

FLUX_POLLIN Handle is ready for reading.

FLUX_POLLOUT Handle is ready for writing.

FLUX_POLLERR Handle has experienced an error.

These functions can be used to integrate a `flux_t` handle into an external event loop. They are analogous to the ZMQ_FD and ZMQ_EVENTS socket options provided by ZeroMQ.

RETURN VALUE

`flux_pollevents()` returns flags on success. On error, -1 is returned, and errno is set appropriately.

`flux_pollfd()` returns a file descriptor on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL Some arguments were invalid.

EXAMPLES

Here is an example of a libev "composite watcher" for a Flux handle using the hooks provided above. This code, more or less, is used internally to integrate flux handles into the Flux reactor, which is based on libev. Refer to the libev documentation for background on how libev works.

There are a total of four different types of libev watcher in the composite watcher. libev "prepare" and "check" callbacks are executed just before and just after libev blocks internally in the poll(2) system call. Here they are used to test `flux_pollevents()`, make user callbacks, and enable/disable no-op "io" and "idle" watchers. The io watcher watches for EV_READ on `flux_pollfd()` file descriptor. The idle watcher, if enabled, is always ready and thus causes the event loop to spin.

When `flux_pollevents()` has poll flags asserted, the idle watcher is enabled. When `flux_pollevents()` has no poll flags asserted, the idle watcher is disabled and the io watcher is enabled. While the idle and io watchers have no callbacks, if either is enabled and ready, the event loop must execute the prepare and check callbacks.

The net results are 1) the edge-triggered notification provided by `flux_pollfd()` is integrated with libev's level-triggered watcher processing; 2) the handle is able to give control back to the event loop between handle event callbacks to preserve fairness, i.e. it doesn't have to consume events until they they are gone in one callback; and 3) the event loop is able to sleep when there are no handle events pending.

```
// ev_flux.h
#include <ev.h>

struct ev_flux;

typedef void (*ev_flux_f)(struct ev_loop *loop,
                          struct ev_flux *w, int revents);
```

(continues on next page)

(continued from previous page)

```
struct ev_flux {
    ev_io       io_w;
    ev_prepare  prepare_w;
    ev_idle     idle_w;
    ev_check    check_w;
    flux_t      *h;
    int         pollfd;
    int         events;
    ev_flux_f   cb;
    void        *data;
};
```

```
// ev_flux.c
static int get_pollevents (flux_t *h)
{
    int e = flux_pollevents (h);
    int events = 0;
    if (e < 0 || (e & FLUX_POLLERR))
        events |= EV_ERROR;
    if ((e & FLUX_POLLIN))
        events |= EV_READ;
    if ((e & FLUX_POLLOUT))
        events |= EV_WRITE;
    return events;
}

static void prepare_cb (struct ev_loop *loop, ev_prepare *w,
                        int revents)
{
    struct ev_flux *fw = (struct ev_flux *)((char *)w
                        - offsetof (struct ev_flux, prepare_w));
    int events = get_pollevents (fw->h);

    if ((events & fw->events) || (events & EV_ERROR))
        ev_idle_start (loop, &fw->idle_w);
    else
        ev_io_start (loop, &fw->io_w);
}

static void check_cb (struct ev_loop *loop, ev_check *w,
                      int revents)
{
    struct ev_flux *fw = (struct ev_flux *)((char *)w
                        - offsetof (struct ev_flux, check_w));
    int events = get_pollevents (fw->h);

    ev_io_stop (loop, &fw->io_w);
    ev_idle_stop (loop, &fw->idle_w);

    if ((events & fw->events) || (events & EV_ERROR))
        fw->cb (loop, fw, events);
}

int ev_flux_init (struct ev_flux *w, ev_flux_f cb,
                  flux_t *h, int events)
{
```

(continues on next page)

(continued from previous page)

```
    w->cb = cb;
    w->h = h;
    w->events = events;
    if ((w->pollfd = flux_pollfd (h)) < 0)
        return -1;

    ev_prepare_init (&w->prepare_w, prepare_cb);
    ev_check_init (&w->check_w, check_cb);
    ev_idle_init (&w->idle_w, NULL);
    ev_io_init (&w->io_w, NULL, w->pollfd, EV_READ);

    return 0;
}

void ev_flux_start (struct ev_loop *loop, struct ev_flux *w)
{
    ev_prepare_start (loop, &w->prepare_w);
    ev_check_start (loop, &w->check_w);
}

void ev_flux_stop (struct ev_loop *loop, struct ev_flux *w)
{
    ev_prepare_stop (loop, &w->prepare_w);
    ev_check_stop (loop, &w->check_w);
    ev_io_stop (loop, &w->io_w);
    ev_idle_stop (loop, &w->idle_w);
}
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod

http://api.zeromq.org/4-0:zmq-getsockopt

http://funcptr.net/2013/04/20/embedding-zeromq-in-the-libev-event-loop

1.2.34 flux_reactor_create(3)

SYNOPSIS

#include <flux/core.h>

flux_reactor_t *flux_reactor_create (int flags);

void flux_reactor_destroy (flux_reactor_t *r);

int flux_reactor_run (flux_reactor_t *r, int flags);

void flux_reactor_stop (flux_reactor_t *r);

void flux_reactor_stop_error (flux_reactor_t *r);

void flux_reactor_active_incref (flux_reactor_t *r);

void flux_reactor_active_decref (flux_reactor_t *r);

DESCRIPTION

`flux_reactor_create()` creates a flux_reactor_t object which can be used to monitor for events on file descriptors, ZeroMQ sockets, timers, and flux_t broker handles.

There is currently only one possible flag for reactor creation:

FLUX_REACTOR_SIGCHLD The reactor will internally register a SIGCHLD handler and be capable of handling flux child watchers (see flux_child_watcher_create(3)).

For each event source and type that is to be monitored, a flux_watcher_t object is created using a type-specific create function, and started with flux_watcher_start(3).

For each event source and type that is to be monitored, a flux_watcher_t object is created and associated with a specific reactor using a type-specific create function, and started with flux_watcher_start(3). To receive events, control must be transferred to the reactor event loop by calling `flux_reactor_run()`.

The full list of flux reactor run flags is as follows:

FLUX_REACTOR_NOWAIT Run one reactor loop iteration without blocking.

FLUX_REACTOR_ONCE Run one reactor loop iteration, blocking until at least one event is handled.

flux_reactor_run() processes events until one of the following conditions is met:

- There are no more active watchers.
- The `flux_reactor_stop()` or `flux_reactor_stop_error()` functions are called by one of the watchers.
- Flags include FLUX_REACTOR_NOWAIT and one reactor loop iteration has been completed.
- Flags include FLUX_REACTOR_ONCE, at least one event has been handled, and one reactor loop iteration has been completed.

If `flux_reactor_stop_error()` is called, this will cause `flux_reactor_run()` to return -1 indicating that an error has occurred. The caller should ensure that a valid error code has been assigned to errno(3) before calling this function.

`flux_reactor_destroy()` releases an internal reference taken at `flux_reactor_create()` time. Freeing of the underlying resources will be deferred if there are any remaining watchers associated with the reactor.

`flux_reactor_active_decref()` and `flux_reactor_active_incref()` manipulate the reactor's internal count of active watchers. Each active watcher takes a reference count on the reactor, and the reactor returns when this count reaches zero. It is useful sometimes to have a watcher that can remain active without preventing the reactor from exiting. To achieve this, call `flux_reactor_active_decref()` after the watcher is started, and `flux_reactor_active_incref()` before the watcher is stopped. Remember that destroying an active reactor internally stops it, so be sure to stop/incref such a watcher first.

RETURN VALUE

`flux_reactor_create()` returns a flux_reactor_t object on success. On error, NULL is returned, and errno is set appropriately.

`flux_reactor_run()` returns the number of active watchers on success. On failure, it returns -1 with errno set. A failure return is triggered when the application sets errno and calls `flux_reactor_stop_error()`.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_fd_watcher_create(3), flux_handle_watcher_create(3), flux_timer_watcher_create(3), flux_watcher_start(3)

libev home page

1.2.35 flux_watcher_now(3)

SYNOPSIS

double flux_reactor_now (flux_reactor_t *r);

void flux_reactor_now_update (flux_reactor_t *r);

double flux_reactor_time (void);

DESCRIPTION

`flux_reactor_now()` returns the current reactor time, which is the time the reactor began processing events. The time will not be updated until the reactor runs out of events and wakes up again. This is a lighter weight alternative to system calls when only coarse event timing is needed, e.g. when all events processed in a given wakeup can be considered simultaneous.

`flux_reactor_now_update()` forces an update to reactor time. This may be useful when the reactor has not run for a while and timing calculations relative to reactor time need to be made, for example when creating timer watchers.

`flux_reactor_time()` returns the system time as a double. Reactor time is a snapshot of `flux_reactor_time()`.

Note: the Flux reactor is based on libev. For additional information on the behavior of reactor time, refer to the libev documentation on `ev_now` and `ev_now_update()`.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_reactor_create (3)

libev home page

1.2.36 flux_recv(3)

SYNOPSIS

#include <flux/core.h>

flux_msg_t *flux_recv (flux_t *h, struct flux_match match, int flags);

DESCRIPTION

`flux_recv()` receives a message using the Flux Message broker, previously opened with `flux_open()` on handle *h*. The message should eventually be destroyed with `flux_msg_destroy()`.

match is a message match structure which limits which messages can be received.

```
struct flux_match {
    int typemask;      // bitmask of matching message types
    uint32_t matchtag; // matchtag
    char *topic_glob;  // glob matching topic string
};
```

The following initializers are available for *match*:

FLUX_MATCH_ANY Match any message.

FLUX_MATCH_EVENT Match any event message.

For additional details on how to use *match*, see flux_msg_cmp(3).

flags is the logical "or" of zero or more of the following flags:

FLUX_O_TRACE Dumps *msg* to stderr.

FLUX_O_NONBLOCK If unable to receive a matching message, return an error rather than block.

Internally, flags are the logical "or" of *flags* and the flags provided to `flux_open()` when the handle was created.

Messages that do not meet *match* criteria, are requeued with `flux_requeue()` for later consumption.

RETURN VALUE

`flux_recv()` returns a message on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOSYS Handle has no recv operation.

EINVAL Some arguments were invalid.

EAGAIN `FLUX_O_NONBLOCK` was selected and `flux_send()` would block.

EXAMPLES

This example opens the Flux broker and displays event messages as they arrive.

```
#include <flux/core.h>
#include "src/common/libutil/log.h"

int main (int argc, char **argv)
{
    flux_t *h;
    flux_msg_t *msg;
    const char *topic;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("flux_open");
    if (flux_event_subscribe (h, "") < 0)
        log_err_exit ("flux_event_subscribe");
    for (;;) {
        if ((msg = flux_recv (h, FLUX_MATCH_EVENT, 0)))
            log_err_exit ("flux_recv");
        if (flux_msg_get_topic (msg, &topic) < 0)
            log_err_exit ("flux_msg_get_topic");
        printf ("Event: %s\n", topic);
        flux_msg_destroy (msg);
    }
    flux_close (h);
    return (0);
}
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_open(3), flux_send(3), flux_requeue(3), flux_msg_cmp(3)

1.2.37 flux_request_decode(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
int flux_request_decode (const flux_msg_t *msg,
                         const char **topic,
                         const char **s);
```

```
int flux_request_unpack (const flux_msg_t *msg,
                         const char **topic,
                         const char *fmt, ...);
```

```
int flux_request_decode_raw (const flux_msg_t *msg,
                             const char **topic,
                             const void **data, int *len);
```

DESCRIPTION

`flux_request_decode()` decodes a request message *msg*.

topic, if non-NULL, will be set the message's topic string. The storage for this string belongs to *msg* and should not be freed.

s, if non-NULL, will be set to the message's NULL-terminated string payload. If no payload exists, it is set to NULL. The storage for this string belongs to *msg* and should not be freed.

`flux_request_unpack()` decodes a request message with a JSON payload as above, parsing the payload using variable arguments with a format string in the style of jansson's `json_unpack()` (used internally). Decoding fails if the message doesn't have a JSON payload.

`flux_request_decode_raw()` decodes a request message with a raw payload, setting *data* and *len* to the payload data and length. The storage for the raw payload belongs to *msg* and should not be freed.

DECODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_unpack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a JSON string to a pointer to a null terminated UTF-8 string. The resulting string is extracted by using 'json_string_value()' internally, so it exists as long as there are still references to the corresponding JSON string.

n (null) Expect a JSON null value. Nothing is extracted.

b (boolean)['int'] Convert a JSON boolean value to a C int, so that *true* is converted to 1 and *false* to 0.

i (integer)['int'] Convert a JSON integer to a C int.

I (integer)['int64_t'] Convert a JSON integer to a C int64_t. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert JSON real to a C double.

F (real)['double'] Convert JSON number (integer or real) to a C double.

o (any value)['json_t *'] Store a JSON value, with no conversion, to a json_t pointer.

O (any value)['json_t *'] Like **o**, but the JSON value's reference count is incremented.

[fmt] (array) Convert each item in the JSON array according to the inner format string. **fmt** may contain objects and arrays, i.e. recursive value extraction is supported.

{fmt} (object) Convert each item in the JSON object according to the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must by **s**. The corresponding argument to unpack functions is read as the object key. The second, fourth, etc. format specifier represent a value and is written to the address given as the corresponding argument. Note that every other argument is read from and every other is written to. **fmt** may contain objects and arrays as values, i.e. recursive value extraction is supported. Any **s** representing a key may be suffixed with **?** to make the key optional. If the key is not found, nothing is extracted.

! This special format specifier is used to enable the check that all object and array items are accessed, on a per-value basis. It must appear inside an array or object as the last format specifier before the closing bracket or brace.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Parsing and Validating Values

RETURN VALUE

These functions return 0 on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL The *msg* argument was NULL.

EPROTO Message decoding failed, such as due to incorrect message type, missing topic string, etc..

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_respond(3), flux_rpc(3)

1.2.38 flux_request_encode(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_msg_t *flux_request_encode (const char *topic,
                                 const char *s);
```

```
flux_msg_t *flux_request_encode_raw (const char *topic,
                                     void *data, int len);
```

DESCRIPTION

`flux_request_encode()` encodes a request message with topic string *topic* and optional NULL terminated string payload *s*. The newly constructed message that is returned must be destroyed with `flux_msg_destroy()`.

`flux_request_encode_raw()` encodes a request message with topic string *topic*. If *data* is non-NULL its contents will be used as the message payload, and the payload type set to raw.

RETURN VALUE

These functions return a message on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

EINVAL The *topic* argument was NULL or *s* is not NULL terminated.

ENOMEM Memory was unavailable.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_response_decode(3), flux_rpc(3)

1.2.39 flux_requeue(3)

SYNOPSIS

#include <flux/core.h>

int flux_requeue (flux_t *h, const flux_msg_t *msg, int flags);

DESCRIPTION

`flux_requeue()` requeues a *msg* in handle *h*. The message can be received with `flux_recv()` as though it arrived from the broker.

flags must be set to one of the following values:

FLUX_RQ_TAIL *msg* is placed at the tail of the message queue.

FLUX_RQ_TAIL *msg* is placed at the head of the message queue.

RETURN VALUE

`flux_requeue()` return zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL Some arguments were invalid.

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_open(3), flux_recv(3), flux_send(3)

1.2.40 flux_respond(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
int flux_respond (flux_t *h, const flux_msg_t *request,
                  const char *s);
```

```
int flux_respond_pack (flux_t *h, const flux_msg_t *request,
                       const char *fmt, ...);
```

```
int flux_respond_raw (flux_t *h, const flux_msg_t *request,
                      const void *data, int length);
```

```
int flux_respond_error (flux_t *h, const flux_msg_t *request,
                        int errnum, const char *errmsg);
```

DESCRIPTION

`flux_respond()`, `flux_respond_pack()`, `flux_respond_raw()`, and `flux_respond_error()` encode and send a response message on handle *h*, deriving topic string, matchtag, and route stack from the provided *request*.

`flux_respond()` sends a response to *request*. If *s* is non-NULL, `flux_respond()` will send it as the response payload, otherwise there will be no payload.

`flux_respond_raw()` is identical except if *data* is non-NULL, `flux_respond_raw()` will send it as the response payload.

`flux_respond_pack()` encodes a response message with a JSON payload, building the payload using variable arguments with a format string in the style of jansson's `json_pack()` (used internally).

`flux_respond_error()` returns an error response to the sender. *errnum* must be non-zero. If *errmsg* is non-NULL, an error string payload is included in the response. The error string may be used to provide a more detailed error message than can be conveyed via *errnum*.

STREAMING SERVICES

Per RFC 6, a "streaming" service must return zero or more non-error responses to a request and a final error response. If the requested operation was successful, the final error response may use ENODATA as the error number. Clients should interpret ENODATA as a non-error end-of-stream marker.

It is essential that services which return multiple responses verify that requests were made with the FLUX_RPC_STREAMING flag by testing the FLUX_MSGFLAG_STREAMING flag, e.g. using `flux_msg_is_streaming()`. If the flag is not set, the service must return an immediate EPROTO error.

ENCODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_pack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a null terminated UTF-8 string to a JSON string.

s# (string)['const char *', 'int'] Convert a UTF-8 buffer of a given length to a JSON string.

s% (string)['const char *', 'size_t'] Like **s#** but the length argument is of type size_t.

+ ['const char *'] Like **s**, but concatenate to the previous string. Only valid after a string.

+# ['const char *', 'int'] Like **s#**, but concatenate to the previous string. Only valid after a string.

+% ['const char *', 'size_t'] Like **+#**, but the length argument is of type size_t.

n (null) Output a JSON null value. No argument is consumed.

b (boolean)['int'] Convert a C int to JSON boolean value. Zero is converted to *false* and non-zero to *true*.

i (integer)['int'] Convert a C int to JSON integer.

I (integer)['int64_t'] Convert a C int64_t to JSON integer. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert a C double to JSON real.

o (any value)['json_t *'] Output any given JSON value as-is. If the value is added to an array or object, the reference to the value passed to **o** is stolen by the container.

O (any value)['json_t *'] Like **o**, but the argument's reference count is incremented. This is useful if you pack into an array or object and want to keep the reference for the JSON value consumed by **O** to yourself.

[fmt] (array) Build an array with contents from the inner format string. **fmt** may contain objects and arrays, i.e. recursive value building is supported.

{fmt} (object) Build an object with contents from the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must be a string as object keys are always strings. The second, fourth, etc. format specifier represent a value. Any value may be an object or array, i.e. recursive value building is supported.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Building Values

RETURN VALUE

These functions return zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

ENOSYS Handle has no send operation.

EINVAL Some arguments were invalid.

EPROTO A protocol error was encountered.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_rpc(3), flux_rpc_raw(3)

RFC 6: Flux Remote Procedure Call Protocol

RFC 3: CMB1 - Flux Comms Message Broker Protocol

1.2.41 flux_response_decode(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
int flux_response_decode (const flux_msg_t *msg,
                          const char **topic,
                          const char **s);
```

```
int flux_response_decode_raw (const flux_msg_t *msg,
                              const char **topic,
                              const void **data, int *len);
```

```
int flux_response_decode_error (const flux_msg_t *msg,
                                const char *errstr);
```

DESCRIPTION

`flux_response_decode()` decodes a response message *msg*.

topic, if non-NULL, will be set to the message's topic string. The storage for this string belongs to *msg* and should not be freed.

s, if non-NULL, will be set to the message's NULL-terminated string payload. If no payload exists, it is set to NULL. The storage for this string belongs to *msg* and should not be freed.

`flux_response_decode_raw()` decodes a response message with a raw payload, setting *data* and *len* to the payload data and length. The storage for the raw payload belongs to *msg* and should not be freed.

`flux_response_decode_error()` decodes an optional error string included with an error response. This fails if the response is not an error, or does not include an error string payload.

RETURN VALUE

These functions return 0 on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL The *msg* argument was NULL.

EPROTO Message decoding failed, such as due to incorrect message type, missing topic string, etc..

ENOENT `flux_response_decode_error()` was called on a message with no error response payload.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_request_encode(3), flux_rpc(3)

1.2.42 flux_rpc(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_rpc (flux_t *h, const char *topic,
                         const char *s,
                         uint32_t nodeid, int flags);
```

```
flux_future_t *flux_rpc_pack (flux_t *h, const char *topic,
                              uint32_t nodeid, int flags,
                              const char *fmt, ...);
```

```
flux_future_t *flux_rpc_raw (flux_t *h, const char *topic,
                             const void *data, int len,
                             uint32_t nodeid, int flags);
```

```
flux_future_t *flux_rpc_message (flux_t *h,
                                 const flux_msg_t *msg,
                                 uint32_t nodeid, int flags);
```

```
int flux_rpc_get (flux_future_t *f, const char **s);
```

```
int flux_rpc_get_unpack (flux_future_t *f, const char *fmt, ...);
```

```
int flux_rpc_get_raw (flux_future_t *f,
                      const void **data, int *len);
```

DESCRIPTION

A remote procedure call (RPC) consists of a matched request and response message exchanged with a Flux service. `flux_rpc()`, `flux_rpc_pack()`, and `flux_rpc_raw()` encode and send a request message via Flux broker handle *h* to a Flux service identified by *topic* and *nodeid*. A `flux_future_t` object is returned which acts as a handle for synchronization and a container for the response message which in turn contains the RPC result.

A lower-level variant of `flux_rpc()`, `flux_rpc_message()` accepts a pre-created request message, assigning *nodeid* and matchtag according to *flags*.

`flux_future_then(3)` may be used to register a reactor callback (continuation) to be called once the response has been received. `flux_future_wait_for(3)` may be used to block until the response has been received. Both accept an optional timeout.

`flux_rpc_get()`, `flux_rpc_get_unpack()`, and `flux_rpc_get_raw()` decode the RPC result. Internally, they call `flux_future_get()` to access the response message stored in the future. If the response message has not yet been received, these functions block until it is, or an error occurs.

REQUEST OPTIONS

The request message is encoded and sent with or without a payload using one of the three `flux_rpc()` variants.

`flux_rpc()` attaches *s*, a NULL terminated string, as request payload. If NULL, the request is encoded without a payload.

`flux_rpc_pack()` attaches a JSON payload encoded as a NULL terminated string using Jansson `json_pack()` style arguments (see below).

`flux_rpc_raw()` attaches a raw payload *data* of length *len*, in bytes. If *data* is NULL, the request is encoded without a payload.

nodeid affects request routing, and must be set to one of the following values:

FLUX_NODEID_ANY The request is routed to the first matching service instance.

FLUX_NODEID_UPSTREAM The request is routed to the first matching service instance, skipping over the sending rank.

integer The request is routed to a specific rank.

flags may be zero or:

FLUX_RPC_NORESPONSE No response is expected. The request will not be assigned a matchtag, and the returned flux_future_t is immediately fulfilled, and may simply be destroyed.

FLUX_RPC_STREAMING The RPC is for a service that may send zero or more non-error responses, and a final error response. ENODATA should be interpreted as a non-error end-of-stream sentinel.

RESPONSE OPTIONS

The response message is stored in the future when the future is fulfilled. At that time it is decoded with `flux_response_decode(3)`. If it cannot be decoded, or if the service returned an error, the future is fulfilled with an error. Otherwise it is fulfilled with the response message. If there was an error, `flux_future_get()` or the `flux_rpc_get()` variants return an error.

`flux_rpc_get()` sets *s* (if non-NULL) to the NULL-terminated string payload contained in the RPC response. If there was no payload, *s* is set to NULL.

`flux_rpc_get_unpack()` decodes the NULL-terminated string payload as JSON using Jansson `json_unpack()` style arguments (see below). It is an error if there is no payload, or if the payload is not JSON.

`flux_rpc_get_raw()` assigns the raw payload of the RPC response message to *data* and its length to *len*. If there is no payload, this function will fail.

PREMATURE DESTRUCTION

If a regular RPC future is destroyed before its response is received, the matchtag allocated to it is not immediately returned to the pool for reuse. If an unclaimed response subsequently arrives with that matchtag, it is returned to the pool then.

If a **streaming** RPC future is destroyed before its terminating response is received, its matchtag is only returned to the pool when an unclaimed **error** response is received. Non-error responses are ignored.

It is essential that services which return multiple responses verify that requests were made with the FLUX_RPC_STREAMING flag and return an immediate EPROTO error if they were not. See flux_respond(3).

CANCELLATION

Flux RFC 6 does not currently specify a cancellation protocol for an individual RPC, but does stipulate that an RPC may be canceled if a disconnect message is received, as is automatically generated by the local connector upon client disconnection.

ENCODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_pack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a null terminated UTF-8 string to a JSON string.

s# (string)['const char *', 'int'] Convert a UTF-8 buffer of a given length to a JSON string.

s% (string)['const char *', 'size_t'] Like **s#** but the length argument is of type size_t.

+ ['const char *'] Like **s**, but concatenate to the previous string. Only valid after a string.

+# ['const char *', 'int'] Like **s#**, but concatenate to the previous string. Only valid after a string.

+% ['const char *', 'size_t'] Like **+#**, but the length argument is of type size_t.

n (null) Output a JSON null value. No argument is consumed.

b (boolean)['int'] Convert a C int to JSON boolean value. Zero is converted to *false* and non-zero to *true*.

i (integer)['int'] Convert a C int to JSON integer.

I (integer)['int64_t'] Convert a C int64_t to JSON integer. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert a C double to JSON real.

o (any value)['json_t *'] Output any given JSON value as-is. If the value is added to an array or object, the reference to the value passed to **o** is stolen by the container.

O (any value)['json_t *'] Like **o**, but the argument's reference count is incremented. This is useful if you pack into an array or object and want to keep the reference for the JSON value consumed by **O** to yourself.

[fmt] (array) Build an array with contents from the inner format string. **fmt** may contain objects and arrays, i.e. recursive value building is supported.

{fmt} (object) Build an object with contents from the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must be a string as object keys are always strings. The second, fourth, etc. format specifier represent a value. Any value may be an object or array, i.e. recursive value building is supported.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Building Values

DECODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_unpack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a JSON string to a pointer to a null terminated UTF-8 string. The resulting string is extracted by using 'json_string_value()' internally, so it exists as long as there are still references to the corresponding JSON string.

n (null) Expect a JSON null value. Nothing is extracted.

b (boolean)['int'] Convert a JSON boolean value to a C int, so that *true* is converted to 1 and *false* to 0.

i (integer)['int'] Convert a JSON integer to a C int.

I (integer)['int64_t'] Convert a JSON integer to a C int64_t. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert JSON real to a C double.

F (real)['double'] Convert JSON number (integer or real) to a C double.

o (any value)['json_t *'] Store a JSON value, with no conversion, to a json_t pointer.

O (any value)['json_t *'] Like **o**, but the JSON value's reference count is incremented.

[fmt] (array) Convert each item in the JSON array according to the inner format string. **fmt** may contain objects and arrays, i.e. recursive value extraction is supported.

{fmt} (object) Convert each item in the JSON object according to the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must by **s**. The corresponding argument to unpack functions is read as the object key. The second, fourth, etc. format specifier represent a value and is written to the address given as the corresponding argument. Note that every other argument is read from and every other is written to. **fmt** may contain objects and arrays as values, i.e. recursive value extraction is supported. Any **s** representing a key may be suffixed with **?** to make the key optional. If the key is not found, nothing is extracted.

! This special format specifier is used to enable the check that all object and array items are accessed, on a per-value basis. It must appear inside an array or object as the last format specifier before the closing bracket or brace.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Parsing and Validating Values

RETURN VALUE

`flux_rpc()`, `flux_rpc_pack()`, and `flux_rpc_raw()` return a flux_future_t object on success. On error, NULL is returned, and errno is set appropriately.

`flux_rpc_get()`, `flux_rpc_get_unpack()`, and `flux_rpc_get_raw()` return zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

ENOSYS Service is not available (misspelled topic string, module not loaded, etc), or flux_t handle has no send operation.

EINVAL Some arguments were invalid.

EPROTO A request was malformed, the FLUX_RPC_STREAMING flag was omitted on a request to a service that may send multiple responses, or other protocol error occurred.

EXAMPLES

This example performs a synchronous RPC with the broker's "attr.get" service to obtain the broker's rank.

```
#include <flux/core.h>
#include "src/common/libutil/log.h"

int main (int argc, char **argv)
{
    flux_t *h;
    flux_future_t *f;
    const char *rankstr;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("flux_open");

    if (!(f = flux_rpc_pack (h, "attr.get", FLUX_NODEID_ANY, 0,
                             "{s:s}", "name", "rank")))
        log_err_exit ("flux_rpc_pack");

    if (flux_rpc_get_unpack (f, "{s:s}", "value", &rankstr) < 0)
        log_err_exit ("flux_rpc_get_unpack");

    printf ("rank is %s\n", rankstr);
    flux_future_destroy (f);

    flux_close (h);
    return (0);
}
```

This example registers a continuation to do the same thing asynchronously.

```
#include <flux/core.h>
#include "src/common/libutil/log.h"

void continuation (flux_future_t *f, void *arg)
{
    const char *rankstr;

    if (flux_rpc_get_unpack (f, "{s:s}", "value", &rankstr) < 0)
        log_err_exit ("flux_rpc_get_unpack");

    printf ("rank is %s\n", rankstr);
    flux_future_destroy (f);
}

int main (int argc, char **argv)
{
    flux_t *h;
    flux_future_t *f;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("flux_open");

```

(continues on next page)

(continued from previous page)

```
    if (!(f = flux_rpc_pack (h, "attr.get", FLUX_NODEID_ANY, 0,
                             "{s:s}", "name", "rank")))
        log_err_exit ("flux_rpc_pack");

    if (flux_future_then (f, -1., continuation, NULL) < 0)
        log_err_exit ("flux_future_then");

    if (flux_reactor_run (flux_get_reactor (h), 0) < 0)
        log_err_exit ("flux_reactor_run");

    flux_close (h);
    return (0);
}
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_future_get(3), flux_respond(3)

RFC 6: Flux Remote Procedure Call Protocol

1.2.43 flux_send(3)

SYNOPSIS

#include <flux/core.h>

int flux_send (flux_t *h, const flux_msg_t *msg, int flags);

DESCRIPTION

`flux_send()` sends *msg* using the Flux Message broker, previously opened with `flux_open()` on handle *h*.

flags is the logical "or" of zero or more of the following flags:

FLUX_O_TRACE Dumps *msg* to stderr.

FLUX_O_NONBLOCK If unable to send, return an error rather than block.

Internally, flags are the logical "or" of *flags* and the flags provided to `flux_open()` when the handle was created.

The message type, topic string, and nodeid affect how the message will be routed by the broker. These attributes are pre-set in the message.

RETURN VALUE

`flux_send()` returns zero on success. On error, -1 is returned, and errno is set appropriately.

ERRORS

ENOSYS Handle has no send operation.

EINVAL Some arguments were invalid.

EAGAIN `FLUX_O_NONBLOCK` was selected and `flux_send()` would block.

EXAMPLES

This example opens the Flux broker and publishes an event message.

```
#include <flux/core.h>
#include "src/common/libutil/log.h"

int main (int argc, char **argv)
{
    flux_t *h;
    flux_msg_t *msg;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("flux_open");
    if (!(msg = flux_event_encode ("snack.bar.closing", NULL)))
        log_err_exit ("flux_event_encode");
    if (flux_send (h, msg, 0) < 0)
        log_err_exit ("flux_send");
    flux_msg_destroy (msg);
    flux_close (h);
    return (0);
}
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_open(3), flux_recv(3), flux_requeue(3)

1.2.44 flux_shell_add_completion_ref(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
int flux_shell_add_completion_ref (flux_shell_t *shell,
                                   const char *fmt,
                                   ...)
```

```
int flux_shell_remove_completion_ref (flux_shell_t *shell,
                                      const char *fmt,
                                      ...)
```

DESCRIPTION

`flux_shell_add_completion_ref` creates a named "completion reference" on the shell object `shell` so that the shell will not consider a job "complete" until the reference is released with `flux_shell_remove_completion_ref`. Once all references have been removed, the shells reactor is stopped with `flux_reactor_stop(shell->r)`.

RETURN VALUE

`flux_shell_add_completion_ref` returns the reference count for the particular name, or -1 on error.

`flux_shell_remove_completion_ref` returns 0 on success, -1 on failure.

ERRORS

EINVAL Either `shell` or `fmt` are NULL.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_reactor_stop(3)

1.2.45 flux_shell_add_event_context(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
int flux_shell_add_event_context (flux_shell_t *shell,
                                  const char *name,
                                  int flags,
                                  const char *fmt,
                                  ...);
```

DESCRIPTION

Add extra context that will be emitted with shell standard event `name` using Jansson `json_pack()` style arguments. The `flags` parameter is currently unused.

RETURN VALUE

Returns 0 on success, -1 if `shell`, `name` or `fmt` are NULL.

ERRORS

EINVAL `shell`, `name` or `fmt` are NULL.

RESOURCES

Github: http://github.com/flux-framework

1.2.46 flux_shell_add_event_handler(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
int flux_shell_add_event_handler (flux_shell_t *shell,
                                  const char *subtopic,
                                  flux_msg_handler_f cb,
                                  void *arg);
```

DESCRIPTION

When the shell initializes, it subscribes to all events with the substring `shell-JOBID.`, where `JOBID` is the jobid under which the shell is running. `flux_shell_add_event_handler()` registers a handler to be run for a **subtopic** within the shell's event namespace, e.g. registering a handler for `subtopic "kill"` will invoke the handler `cb` whenever an event named `shell-JOBID.kill` is generated.

RETURN VALUE

Returns -1 if `shell`, `shell->h`, `subtopic` or `cb` are NULL, or if underlying calls to `asprintf()` or `flux_msg_handler_create()` fail.

ERRORS

EINVAL `shell`, `shell->h`, `subtopic` or `cb` are NULL.

RESOURCES

Github: http://github.com/flux-framework

1.2.47 flux_shell_aux_set(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
typedef void (*flux_free_f)(void *arg);
```

```
int flux_shell_aux_set (flux_shell_t *shell,
                        const char *name,
                        void *aux,
                        flux_free_f free_fn);
```

```
void * flux_shell_aux_get (flux_shell_t *shell,
                           const char *key);
```

DESCRIPTION

flux_shell_aux_set() attaches application-specific data to the parent object. It stores data aux by key name, with optional destructor destroy. The destructor, if non-NULL, is called when the parent object is destroyed, or when key is overwritten by a new value. If aux is NULL, the destructor for a previous value, if any is called, but no new value is stored. If name is NULL, aux is stored anonymously.

flux_shell_aux_get() retrieves application-specific data by name. If the data was stored anonymously, it cannot be retrieved.

The implementation (as opposed to the header file) uses the variable names `shell`, `key`, `val` and `free_fn`, which may be more intuitive.

In most cases the key, value and free function will be non-null. Several exceptions are supported.

First, if `key` and `val` are non-NULL but `free_fn` is null, the caller is responsible for memory management associated with the value.

Second, if `key` is NULL but `val` and `free_fun` are not NULL, the lifetime of the object is tied to the lifetime of the underlying aux object; the object will be destroyed during the destruction of the aux. The value cannot be retrieved.

Third, a non-null `key` and a null `val` deletes the value previously associated with the key by calling its previously-associated `free_fn`, if the destructor exists.

RETURN VALUE

`flux_aux_set()` returns 0 on success, or -1 on failure, with errno set.

`flux_shell_aux_get()` returns data on success, or NULL on failure, with errno set.

ERRORS

EINVAL

`shell` is null; or
both `name` (aka `key`) and `aux` (aka `val`) are null; or
`free_fn` is not null but `aux` is; or
`free_fn` and `name` are both null.

ENOMEM Out of memory.

ENOENT `flux_aux_get()` could not find an entry for *key*.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_aux_get(3), flux_aux_set(3)

1.2.48 flux_shell_current_task(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
flux_shell_task_t *flux_shell_current_task (flux_shell_t *shell);
```

```
flux_shell_task_t *flux_shell_task_first (flux_shell_t *shell);
```

```
flux_shell_task_t *flux_shell_task_next (flux_shell_t *shell);
```

DESCRIPTION

flux_shell_task_first and flux_shell_task_next are used to iterate over all current tasks known to the shell.

`flux_shell_current_task` returns the current task for `task_init`, `task_exec` and `task_exec` callbacks and NULL in any other context.

`flux_shell_task_first` and `flux_shell_task_next` return the first and next tasks, respectively.

RETURN VALUE

The relevant `flux_shell_task_t*` value, or NULL on error.

ERRORS

EINVAL `shell` is NULL.

EAGAIN There are no tasks.

RESOURCES

Github: http://github.com/flux-framework

1.2.49 flux_shell_get_flux(3)

SYNOPSIS

```
#include <flux/shell.h>
```

```
flux_t *flux_shell_get_flux (flux_shell_t *shell);
```

DESCRIPTION

Returns the Flux handle.

RETURN VALUE

Returns the Flux handle.

ERRORS

No error conditions are possible.

EXAMPLE

```
// Set a timer in flux_plugin_init().
```

```
void flux_plugin_init (flux_plugin_t *p){
```

```
// Get the shell handle,
flux_shell_t *shell = flux_plugin_get_shell( p );
```

```
// use that to get the flux handle,
flux_t *flux = flux_shell_get_flux( shell );
```

```
// and use that to get the reactor handle.
flux_reactor_t *reactor = flux_get_reactor( flux );
```

```
flux_watcher_t* timer   = flux_timer_watcher_create( reactor, 0.1, 0.1, timer_cb,␣
↪NULL );
flux_watcher_start(timer);
```

```
....
```

RESOURCES

Github: http://github.com/flux-framework

1.2.50 flux_shell_get_info(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
int flux_shell_get_info (flux_shell_t *shell,
                         char **json_str);
```

```
int flux_shell_info_unpack (flux_shell_t *shell,
                            const char *fmt,
                            ...);
```

```
int flux_shell_get_rank_info (flux_shell_t *shell,
                              int shell_rank,
                              char **json_str);
```

```
int flux_shell_rank_info_unpack (flux_shell_t *shell,
                                 int shell_rank,
                                 const char *fmt,
                                 ...);
```

DESCRIPTION

`flux_shell_get_info()` returns shell information as a json string with the following layout:

```
"jobid":I,
"rank":i,
"size":i,
"ntasks";i,
"service";s,
"options": { "verbose":b, "standalone":b },
"jobspec":o,
"R":o
```

`flux_shell_get_rank_info()` returns shell rank information as a json string with the following layout:

```
"broker_rank":i,
"ntasks":i
"taskids":s
"resources": { "cores":s, ... }
```

where `broker_rank` is the broker rank on which the target shell rank of the query is running, `ntasks` is the number of tasks running under that shell rank, `taskids` is a list of task id assignments for those tasks (an RFC 22 idset string), and `resources` is a dictionary of resource name to resource ids assigned to the shell rank.

`flux_shell_info_unpack()` and `flux_shell_rank_info_unpack()` accomplished the same thing with Jansson-style formatting arguments.

If `shell_rank` is set to -1, the current shell rank is used.

RETURN VALUE

All functions return 0 on success and -1 on error.

ERRORS

EINVAL if `shell` is NULL, or either `json_str` or `fmt` are NULL, or if `shell_rank` is less than -1.

SEE ALSO

For an overview of the Jansson API, see https://jansson.readthedocs.io/en/2.8/apiref.html.

RESOURCES

Github: http://github.com/flux-framework

1.2.51 flux_shell_get_jobspec_info(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
int flux_shell_get_jobspec_info (flux_shell_t *shell,
                                 char **json_str);
```

```
int flux_shell_jobspec_info_unpack (flux_shell_t *shell,
                                    const char *fmt,
                                    ...);
```

DESCRIPTION

`flux_shell_get_jobspec_info()` returns jobspec summary information from the flux job shell as a json string. The only key guaranteed to be in the returned JSON object is the jobspec `version`, e.g.

:: {"version": 1}

For jobspec version 1, the following keys are also available:

```
{
  "ntasks":i,          # number of tasks requested
  "nslots":i,          # number of task slots
  "cores_per_slot":i   # number of cores per task slot
  "nnodes":i           # number of nodes requested, -1 if unset
  "slots_per_node":i   # number of slots per node, -1 if unavailable
 }
```

This summary information is derived from the jobspec by the shell and is shared with plugins in order to avoid duplication of effort.

Currently only version 1 jobspec is supported.

`flux_shell_jobspec_info_unpack()` accomplishes the same thing with Jansson-style formatting arguments.

RETURN VALUE

All functions return 0 on success and -1 on error.

ERRORS

EINVAL if `shell` is NULL, or either `json_str` or `fmt` are NULL, or if `shell_rank` is less than -1.

SEE ALSO

For an overview of the Jansson API, see https://jansson.readthedocs.io/en/2.8/apiref.html.

RESOURCES

Github: http://github.com/flux-framework

1.2.52 flux_shell_getenv(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
const char * flux_shell_getenv (flux_shell_t *shell,
                                const char *name);
```

```
int flux_shell_get_environ (flux_shell_t *shell,
                            char **json_str);
```

```
int flux_shell_setenvf (flux_shell_t *shell,
                        int overwrite,
                        const char *name,
                        const char *fmt,
                        ...)
```

```
int flux_shell_unsetenv (flux_shell_t *shell,
                         const char *name);
```

DESCRIPTION

`flux_shell_getenv()` returns the value of an environment variable from the global job environment. `flux_shell_get_environ()` returns 0 on success with `*json_str` set to an allocated JSON string, or -1 on failure with `errno` set. `flux_shell_setenvf()` sets an environment variable in the global job environment using `printf(3)` style format arguments. `flux_shell_unsetenv()` unsets the specified environment variable in the global job environment.

RETURN VALUE

`flux_shell_getenv()` returns NULL if either `shell` or `name` is NULL, or if the variable is not found.

`flux_shell_get_environ()` returns a json string on success or NULL on failure.

`flux_shell_setenvf()` and `flux_shell_unsetenv()` return 0 on success and -1 on failure.

ERRORS

EINVAL `shell`, `name` or `fmt` is NULL.

EEXIST The variable already exists and `overwrite` was not non-zero (`flux_shell_setenvf()`).

RESOURCES

Github: http://github.com/flux-framework

1.2.53 flux_shell_getopt(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
int flux_shell_getopt (flux_shell_t *shell,
                       const char *name,
                       char **json_str);
```

```
int flux_shell_getopt_unpack (flux_shell_t *shell,
                              const char *name,
                              const char *fmt,
                              ...);
```

```
int flux_shell_setopt (flux_shell_t *shell,
                       const char *name,
                       const char *json_str);
```

```
int flux_shell_setopt_pack (flux_shell_t *shell,
                            const char *name,
                            const char *fmt,
                            ...);
```

DESCRIPTION

`flux_shell_getopt()` gets shell option `name` as a JSON string from jobspec `attributes.system.shell.options.name`.

`flux_shell_setopt()` sets shell option `name`, making it available to subsequent calls from `flux_shell_getopt()`. If `json_str` is NULL, the option is unset.

`flux_shell_getopt_unpack()` and `flux_shell_setopt_unpack()` use Jansson format strings to accomplish the same functionality.

RETURN VALUE

`flux_shell_getopt()` and `flux_shell_getopt_unpack()` return 1 on success, 0 if `name` was not set, and -1 on error,

`flux_shell_setopt()` and `flux_shell_setopt_pack` return 0 on success and -1 on error.

ERRORS

EINVAL `name` or `shell` is NULL.

ENOMEM The process has exhausted its memory.

RESOURCES

Github: http://github.com/flux-framework

1.2.54 flux_shell_killall(3)

SYNOPSIS

```
#include <flux/shell.h>
```

```
void flux_shell_killall (flux_shell_t *shell,
                         int sig);
```

DESCRIPTION

Sends the signal `sig` to all processes running in `shell`. No errors are set, but the call returns immediately if `shell` is NULL or if `sig` is zero or negative.

RETURN VALUE

None.

ERRORS

None.

RESOURCES

Github: http://github.com/flux-framework

1.2.55 flux_shell_log(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
void flux_shell_log (const char *component,
                     int level,
                     const char *file,
                     int line,
                     const char *fmt,
                     ...)
```

```
int flux_shell_err (const char *component,
                    const char *file,
                    int line,
                    int errnum,
                    const char *fmt,
                    ...)
```

```
void flux_shell_fatal (const char *component,
                       const char *file,
                       int line,
                       int errnum,
                       int exit_code,
                       const char *fmt,
                       ...)
```

```
void flux_shell_raise (const char *type,
                       int severity,
                       const char *fmt,
                       ...)
```

```
int flux_shell_log_setlevel (int level,
                             const char *dest);
```

DESCRIPTION

`flux_shell_log()` logs a message at for shell component or plugin `component` at `level` to all loggers registered to receive messages at that severity or greater. See `flux_log` for a list of supported levels.

The following macros handle common levels. For external shell plugins, the required macro `FLUX_SHELL_PLUGIN_NAME` is automatically substituted for the `component` in all macros.

```
#define shell_trace(...) \
```

```
#define shell_debug(...) \
```

```
#define shell_log(...) \
```

```
#define shell_warn(...) \
```

```
#define shell_log_error(...) \
```

`flux_shell_err()` logs a message at FLUX_SHELL_ERROR level, additionally appending the result of strerror(`errnum`) for convenience. Macros include:

```
#define shell_log_errn(errn, ...) \
```

```
#define shell_log_errno(...) \
```

Note that `errno` is the standard global value defined in `errno.h` and `errn` is a user-provided error code.

`flux_shell_fatal()` logs a message at FLUX_SHELL_FATAL level and schedules termination of the job shell. This may generate an exception if tasks are already running. Exits with `exit_code`. While the macro names are similar to those using `flux_shell_err()`, note that the choices of `errnum` are either 0 or `errno`.

```
#define shell_die(code,...) \
```

```
#define shell_die_errno(code,...) \
```

`flux_shell_raise()` explicitly raises an exception for the current job of the given `type` and `severity`. Exceptions of severity 0 will result in termination of the job by the execution system.

`flux_shell_log_setlevel()` sets default severity of logging destination `dest` to `level`. If `dest` is NULL then the internal log dispatch level is set (i.e. no messages above severity level will be logged to any log destination). Macros include:

```
#define shell_set_verbose(n) \
flux_shell_log_setlevel(FLUX_SHELL_NOTICE+n, NULL)
```

```
#define shell_set_quiet(n) \
flux_shell_log_setlevel(FLUX_SHELL_NOTICE-n, NULL)
```

RETURN VALUE

`flux_shell_err()` returns -1 with errno = errnum, so that the function can be used as: return flux_shell_err(...);

`flux_shell_log_setlevel()` will return -1 and set `errno` to EINVAL if the requested `level` is not valid or if `dest` is not a valid pointer to a logger shell.

ERRORS:

EINVAL `level` or `dest` is not valid.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_log(3)

1.2.56 flux_shell_plugstack_call(3)

SYNOPSIS

```
#include <flux/shell.h>
```

```
int flux_shell_plugstack_call (flux_shell_t *shell,
                               const char *topic,
                               flux_plugin_arg_t *args);
```

DESCRIPTION

The job shell implements a flexible plugin architecture which allows registration of one or more callback functions on arbitrary topic names. The stack of functions "listening" on a given topic string is called the "plugin stack". `flux_shell_plugstack_call()` exports the ability to call into the plugin stack so that plugins can invoke callbacks from other plugins.

RETURN VALUE

Returns 0 on success and -1 on failure, setting errno.

ERRORS:

EINVAL `shell` or `topic` are NULL.

RESOURCES

Github: http://github.com/flux-framework

1.2.57 flux_shell_rpc_pack(3)

SYNOPSIS

```
#include <flux/shell.h>
#include <errno.h>
```

```
flux_future_t *flux_shell_rpc_pack (flux_shell_t *shell,
                                    const char *method,
                                    int shell_rank,
                                    int flags,
                                    const char *fmt,
                                    ...);
```

DESCRIPTION

Send a remote procedure call `method` to another shell in the same job at shell rank `shell_rank`.

RETURN VALUE

Returns NULL on failure.

ERRORS

EINVAL `shell`, `method` or `fmt` are NULL, or if `rank` is less than 0.

RESOURCES

Github: http://github.com/flux-framework

1.2.58 flux_shell_service_register(3)

SYNOPSIS

```
#include <flux/shell.h>
```

```
int flux_shell_service_register (flux_shell_t *shell,
                                 const char *method,
                                 flux_msg_handler_f cb,
                                 void *arg);
```

DESCRIPTION

The job shell registers a unique service name with the flux broker on startup, and posts the topic string for this service in the context of the `shell.init` event. `flux_shell_service_register()` allows registration of a request handler `cb` for subtopic `method` on this service endpoint, allowing other job shells and/or flux commands to interact with arbitrary services within a job.

RETURN VALUE

Returns -1 on failure, 0 on success.

ERRORS

EINVAL `shell`, `method` or `cb` is NULL.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

`flux_msg_handler_create(3)`

1.2.59 flux_shell_task_channel_subscribe(3)

SYNOPSIS

```
#include <flux/shell.h>
```

```
int flux_shell_task_channel_subscribe (flux_shell_task_t *task,
                                       const char *channel,
                                       flux_shell_task_io_f cb,
                                       void *arg);
```

DESCRIPTION

Call `cb` when shell task output channel `name` is ready for reading.

Callback can then call `flux_shell_task_get_subprocess()` and use `flux_subprocess_read()` or `getline()` on the result to get available data. Only one subscriber per stream is allowed.

RETURN VALUE

Returns 0 on success and -1 on error.

Not yet implemented.

ERRORS

EEXIST `flux_shell_task_channel_subscribe()` is called on a stream with an existing subscriber

RESOURCES

Github: http://github.com/flux-framework

1.2.60 flux_shell_task_get_info(3)

SYNOPSIS

```
#include <flux/shell.h>
```

```
int flux_shell_task_get_info (flux_shell_task_t *task,
                              char **json_str);
```

```
int flux_shell_task_info_unpack (flux_shell_task_t *task,
                                 const char *fmt, ...);
```

DESCRIPTION

Returns task info either as a json string (specified below) or using Jansson-style parameters. The structure of the former is:

```
"localid":i,
"rank":i,
"state":s,
"pid":I,
"wait_status":i,
"exitcode":i,
"signaled":i
```

RETURN VALUE

Returns 0 on success and -1 on failure. A failure will not necessarily set errno.

ERRORS

EINVAL If `task` or `json_str` is NULL.

RESOURCES

Github: http://github.com/flux-framework

1.2.61 flux_shell_task_subprocess(3)

SYNOPSIS

```
#include <flux/shell.h>
```

```
flux_subprocess_t *flux_shell_task_subprocess (flux_shell_task_t *task)
```

```
flux_cmd_t *flux_shell_task_cmd (flux_shell_task_t *task)
```

DESCRIPTION

`flux_shell_task_subprocess` returns the subprocess for a shell task in `task_fork` and `task_exit` callbacks.

`flux_shell_task_cmd` returns the cmd structure for a shell task.

RETURN VALUE

`flux_shell_task_subprocess` returns the `proc` field of the `task`, and `flux_shell_task_cmd` returns the `cmd` field, or NULL on error.

ERRORS

EINVAL `task` is NULL.

RESOURCES

Github: http://github.com/flux-framework

1.2.62 flux_signal_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
flux_watcher_t *flux_signal_watcher_create (flux_reactor_t *r,
                                            int signum,
                                            flux_watcher_f callback,
                                            void *arg);
```

```
int flux_signal_watcher_get_signum (flux_watcher_t *w);
```

DESCRIPTION

`flux_signal_watcher_create()` creates a reactor watcher that monitors for receipt of signal *signum*.

The callback *revents* argument should be ignored.

When one *callback* is shared by multiple watchers, the signal number that triggered the event can be obtained with `flux_signal_watcher_get_signum()`.

RETURN VALUE

flux_signal_watcher_create() returns a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3)

libev home page

1.2.63 flux_stat_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
flux_watcher_t *flux_stat_watcher_create (flux_reactor_t *r,
                                          const char *path,
                                          double interval,
                                          flux_watcher_f callback,
                                          void *arg);
```

```
void flux_stat_watcher_get_rstat (flux_watcher_t *w,
                                  struct stat *stat,
                                  struct stat *prev);
```

DESCRIPTION

`flux_stat_watcher_create()` creates a reactor watcher that monitors for changes in the status of the file system object represented by *path*. If the file system object exists, inotify(2) is used, if available; otherwise the reactor polls the file every *interval* seconds. A value of zero selects a conservative default (currently five seconds).

The callback *revents* argument should be ignored.

`flux_stat_watcher_get_rstat ()` may be used to obtain the status within *callback*. If non-NULL, *stat* receives the current status. If non-NULL, *prev* receives the previous status.

If the object does not exist, stat->st_nlink will be zero and other status fields are undefined. The appearance/disappearance of a file is considered a status change like any other.

RETURN VALUE

flux_stat_watcher_create() returns a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3), stat(2)

libev home page

1.2.64 flux_sync_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
flux_future_t *flux_sync_create (flux_t *h, double minimum);
```

DESCRIPTION

`flux_sync_create()` creates a future that is fulfilled when the system heartbeat message is received. System heartbeats are event messages published periodically at a configurable interval. Synchronizing Flux internal overhead to the heartbeat can, in theory, reduce disruption to bulk synchronous applications.

If *minimum* is greater than zero, it establishes a minimum time in seconds between fulfillments. Heartbeats that arrive too soon after the last one are ignored. This may be used to protect from thrashing if the heartbeat period is set too fast, or if heartbeats arrive close to one another in time due to overlay network congestion.

A maximum time between fulfillments may be established by specifying a continuation timeout with `flux_future_then()`. If the timeout expires, the future is fulfilled with an error (ETIMEDOUT), as usual.

On each fulfillment, `flux_future_reset()` should be called to enable the future to be fulfilled again, and to re-start any timeout.

RETURN VALUE

`flux_sync_create()` returns a future, or NULL on failure with errno set.

ERRORS

EINVAL One or more arguments were invalid.

ENOMEM Out of memory.

EXAMPLE

Set up a continuation callback for each heartbeat that arrives at least *sync_min* seconds from the last, with a timeout of *sync_max* seconds:

```
#include <flux/core.h>
#include "src/common/libutil/log.h"

const double sync_min = 1.0;
const double sync_max = 60.0;

void sync_continuation (flux_future_t *f, void *arg)
```

(continues on next page)

(continued from previous page)

```
{
    // do work here
    flux_future_reset (f);
}

int main (int argc, char **argv)
{
    flux_t *h;
    flux_future_t *f;

    if (!(h = flux_open (NULL, 0)))
        log_err_exit ("could not connect to broker");

    if (!(f = flux_sync_create (h, sync_max)))
        log_err_exit ("error creating future");

    if (flux_future_then (f, sync_min, sync_continuation, NULL) < 0)
        log_err_exit ("error registering continuation");

    if (flux_reactor_run (flux_get_reactor (h), 0) < 0)
        log_err_exit ("reactor returned wtih error");

    flux_future_destroy (f);

    flux_close (h);
    return (0);
}
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_future_then(3), flux_future_get(3), flux_future_reset(3)

1.2.65 flux_timer_watcher_create(3)

SYNOPSIS

```
#include <flux/core.h>
```

```
typedef void (*flux_watcher_f)(flux_reactor_t *r,
                               flux_watcher_t *w,
                               int revents, void *arg);
```

```
flux_watcher_t *flux_timer_watcher_create (flux_reactor_t *r,
                                           double after, double repeat,
                                           flux_watcher_f callback,
                                           void *arg);
```

```
void flux_timer_watcher_reset (flux_watcher_t *w,
                               double after, double repeat);
```

DESCRIPTION

`flux_timer_watcher_create()` creates a flux_watcher_t object which monitors for timer events. A timer event occurs when *after* seconds have elapsed, and optionally again every *repeat* seconds. When events occur, the user-supplied *callback* is invoked.

If *after* is 0., the flux_watcher_t will be immediately ready when the reactor is started. If *repeat* is 0., the flux_watcher_t will automatically be stopped when *after* seconds have elapsed.

Note that *after* is internally referenced to reactor time, which is only updated when the reactor is run/created, and therefore can be out of date. Use `flux_reactor_now_update(3)` to manually update reactor time before creating timer watchers in such cases. Refer to "The special problem of time updates" in the libev manual for more information.

To restart a timer that has been automatically stopped, you must reset the *after* and *repeat* values with `flux_timer_watcher_reset()` before calling `flux_watcher_start()`.

The callback *revents* argument should be ignored.

Note: the Flux reactor is based on libev. For additional information on the behavior of timers, refer to the libev documentation on `ev_timer`.

RETURN VALUE

`flux_timer_watcher_create()` returns a flux_watcher_t object on success. On error, NULL is returned, and errno is set appropriately.

ERRORS

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_watcher_start(3), flux_reactor_start(3), flux_reactor_now(3)

libev home page

1.2.66 flux_watcher_start(3)

SYNOPSIS

void flux_watcher_start (flux_watcher_t *w);

void flux_watcher_stop (flux_watcher_t *w);

void flux_watcher_destroy (flux_watcher_t *w);

double flux_watcher_next_wakeup (flux_watcher_t *w);

DESCRIPTION

`flux_watcher_start()` activates a flux_watcher_t object *w* so that it can receive events. If *w* is already active, the call has no effect. This may be called from within a flux_watcher_f callback.

`flux_watcher_stop()` deactivates a flux_watcher_t object *w* so that it stops receiving events. If *w* is already inactive, the call has no effect. This may be called from within a flux_watcher_f callback.

`flux_watcher_destroy()` destroys a flux_watcher_t object *w*, after stopping it. It is not safe to destroy a watcher object within a flux_watcher_f callback.

`flux_watcher_next_wakeup()` returns the absolute time that the watcher is supposed to trigger next. This function only works for *timer* and *periodic* watchers, and will return a value less than zero with errno set to `EINVAL` otherwise.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux_reactor_create (3)

1.2.67 idset_create(3)

SYNOPSIS

```
#include <flux/idset.h>
```

```
struct idset *idset_create (size_t slots, int flags);
```

```
void idset_destroy (struct idset *idset);
```

```
struct idset *idset_copy (const struct idset *idset);
```

```
int idset_set (struct idset *idset, unsigned int id);
```

```
int idset_range_set (struct idset *idset,
                     unsigned int lo, unsigned int hi);
```

```
int idset_clear (struct idset *idset, unsigned int id);
```

```
int idset_range_clear (struct idset *idset,
                       unsigned int lo, unsigned int hi)
```

```
bool idset_test (const struct idset *idset, unsigned int id);
```

```
unsigned int idset_first (const struct idset *idset);
```

```
unsigned int idset_next (const struct idset *idset, unsigned int prev);
```

```
unsigned int idset_last (const struct idset *idset)
```

```
size_t idset_count (const struct idset *idset);
```

```
bool idset_equal (const struct idset *set1, const struct idset *set2);
```

USAGE

cc [flags] files -lflux-idset [libraries]

DESCRIPTION

An idset is a set of numerically sorted, non-negative integers. It is internally represented as a van Embde Boas (or vEB) tree. Functionally it behaves like a bitmap, and has space efficiency comparable to a bitmap, but performs operations (insert, delete, lookup, findNext, findPrevious) in O(log(m)) time, where pow (2,m) is the number of slots in the idset.

`idset_create()` creates an idset. *slots* specifies the highest numbered *id* it can hold, plus one. The size is fixed unless *flags* specify otherwise (see FLAGS below).

`idset_destroy()` destroys an idset.

`idset_copy()` copies an idset.

`idset_set()` and `idset_clear()` set or clear *id*.

`idset_range_set()` and `idset_range_clear()` set or clear an inclusive range of ids, from *lo* to *hi*.

`idset_test()` returns true if *id* is set, false if not.

`idset_first()` and `idset_next()` can be used to iterate over ids in the set, returning IDSET_INVALID_ID at the end. `idset_last()` returns the last (highest) id, or IDSET_INVALID_ID if the set is empty.

`idset_count()` returns the number of ids in the set.

`idset_equal()` returns true if the two idset objects *set1* and *set2* are equal sets, i.e. the sets contain the same set of integers.

FLAGS

IDSET_FLAG_AUTOGROW Valid for `idset_create()` only. If set, the idset will grow to accommodate any id inserted into it. The internal vEB tree is doubled in size until until the new id can be inserted. Resizing is a costly operation that requires all ids in the old tree to be inserted into the new one.

RETURN VALUE

`idset_copy()` returns an idset on success which must be freed with `idset_destroy()`. On error, NULL is returned with errno set.

`idset_first()`, `idset_next()`, and `idset_last()` return an id, or IDSET_INVALID_ID if no id is available.

`idset_equal()` returns true if *set1* and *set2* are equal sets, or false if they are not equal, or either argument is *NULL*.

Other functions return 0 on success, or -1 on error with errno set.

ERRORS

EINVAL One or more arguments were invalid.

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

idset_encode(3), idset_add(3)

RFC 22: Idset String Representation

1.2.68 idset_encode(3)

SYNOPSIS

```
#include <flux/idset.h>
```

```
char *idset_encode (const struct idset *idset, int flags);
```

```
struct idset *idset_decode (const char *s);
```

```
struct idset *idset_ndecode (const char *s, size_t len);
```

USAGE

cc [flags] files -lflux-idset [libraries]

DESCRIPTION

Refer to `idset_create(3)` for a general description of idsets.

`idset_encode()` creates a string from *idset*. The string contains a comma-separated list of ids, potentially modified by *flags* (see FLAGS below).

`idset_decode()` creates an idset from a string *s*. The string may have been produced by `idset_encode()`. It must consist of comma-separated non-negative integer ids, and may also contain hyphenated ranges. If enclosed in square brackets, the brackets are ignored. Some examples of valid input strings are:

```
1,2,5,4
```

```
1-4,7,9-10
```

```
42
```

```
[99-101]
```

`idset_ndecode()` creates an idset from a sub-string *s* defined by length *len*.

FLAGS

IDSET_FLAG_BRACKETS Valid for `idset_encode()` only. If set, the encoded string will be enclosed in brackets, unless the idset is a singleton (contains only one id).

IDSET_FLAG_RANGE Valid for `idset_encode()` only. If set, any consecutive ids are compressed into hyphenated ranges in the encoded string.

RETURN VALUE

`idset_decode()` and `idset_ndecode()` return idset on success which must be freed with `idset_destroy(3)`. On error, NULL is returned with errno set.

`idset_encode()` returns a string on success which must be freed with `free()`. On error, NULL is returned with errno set.

ERRORS

EINVAL One or more arguments were invalid.

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

idset_create(3)

RFC 22: Idset String Representation

1.2.69 idset_add(3)

SYNOPSIS

```
#include <flux/idset.h>
```

```
struct idset *idset_union (const struct idset *a,
                           const struct idset *b);
```

```
struct idset *idset_difference (const struct idset *a,
                                const struct idset *b);
```

```
struct idset *idset_intersect (const struct idset *a,
                               const struct idset *b);
```

```
int idset_add (struct idset *a,
               const struct idset *b);
```

```
int idset_subtract (struct idset *a,
                    const struct idset *b);
```

```
bool idset_has_intersection (const struct idset *a,
                             const struct idset *b);
```

```
#define idset_clear_all (x) idset_subtract (x, x)
```

USAGE

cc [flags] files -lflux-idset [libraries]

DESCRIPTION

Refer to `idset_create(3)` for a general description of idsets.

`idset_union()` creates a new idset that is the union of *a* and *b*.

`idset_difference()` creates a new idset that is *a* with the members of *b* removed.

`idset_intersect()` creates a new idset containing only members of *a* and *b* that are in both sets.

`idset_add()` adds the members of *b* to *a*.

`idset_subtract()` removes the members of *b* from *a*.

`idset_has_intersection()` tests whether *a* and *b* have any members in common.

`idset_clear_all()` removes all members of *x*

RETURN VALUE

`idset_union()`, `idset_difference()`, and `idset_intersect()` return an idset on success which must be freed with `idset_destroy()`. On error, NULL is returned with errno set.

`idset_add()`, `idset_subtract()`, and `idset_clear_all()` return 0 on success. On error, -1 is returned with errno set.

`idset_has_intersection()` returns true or false.

ERRORS

EINVAL One or more arguments were invalid.

ENOMEM Out of memory.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

idset_encode(3), idset_encode(3)

RFC 22: Idset String Representation

1.2.70 flux_jobtap_get_flux(3)

SYNOPSIS

```
#include <flux/core.h>
#include <flux/jobtap.h>
```

```
flux_t *flux_jobtap_get_flux (flux_plugin_t *p);
```

```
int flux_jobtap_service_register (flux_plugin_t *p,
                                  const char *method,
                                  flux_msg_handler_f cb,
                                  void *arg);
```

```
int flux_jobtap_reprioritize_all (flux_plugin_t *p);
```

```
int flux_jobtap_reprioritize_job (flux_plugin_t *p,
                                  flux_jobid_t id,
                                  unsigned int priority);
```

```
int flux_jobtap_priority_unavail (flux_plugin_t *p,
                                  flux_plugin_arg_t *args);
```

```
int flux_jobtap_reject_job (flux_plugin_t *p,
                            flux_plugin_arg_t *args,
                            const char *fmt, ...);
```

DESCRIPTION

These interfaces are used by Flux *jobtap* plugins which are used to extend the job manager broker module.

`flux_jobtap_get_flux()` returns the job manager's Flux handle given the plugin's `flux_plugin_t *`. This can be used by a *jobtap* plugin to send RPCs, schedule timer watchers, or other asynchronous work.

`flux_jobtap_service_register()` registers a service name `method` under the job manager which will be handled by the provided message handler `cb`. The constructed service name will be `job-manager.<name>.<method>` where `name` is the name of the plugin as returned by `flux_plugin_get_name(3)`. As such, this call may fail if the *jobtap* plugin has not yet set a name for itself using `flux_plugin_set_name(3)`.

`flux_jobtap_reprioritize_all()` requests that the job manager begin reprioritization of all pending jobs, i.e. jobs in the PRIORITY and SCHED states. This will result on each job having a `job.priority.get` callback invoked on it.

`flux_jobtap_reprioritize_job()` allows a *jobtap* plugin to asynchronously assign the priority of a job.

`flux_jobtap_priority_unavail()` is a convenience function which may be used by a plugin in the `job.state.priority` priority callback to indicate that a priority for the job is not yet available. It can be called as:

```
return flux_jobtap_priority_unavail (p, args);
```

`flux_jobtap_reject_job()` is a convenience function which may be used by a plugin from the `job.validate` callback to reject a job before its submission is fully complete. The error and optional message supplied in `fmt` will be returned to the originating job submission request. This function returns `-1` so that it may be conveniently called as:

```
return flux_jobtap_reject_job (p, args,
                               "User exceeded %d jobs",
                               limit);
```

RETURN VALUE

`flux_jobtap_get_flux()` returns a `flux_t *` handle on success. `NULL` is returned with errno set to `EINVAL` if the supplied `flux_plugin_t *p` is not a jobtap plugin handle.

`flux_jobtap_reject_job()` always returns `-1` so that it may be used to exit the `job.validate` callback.

The remaining functions return 0 on success, -1 on failure.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-jobtap-plugins(7)

1.2.71 ENCODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_pack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a null terminated UTF-8 string to a JSON string.

s# (string)['const char *', 'int'] Convert a UTF-8 buffer of a given length to a JSON string.

s% (string)['const char *', 'size_t'] Like **s#** but the length argument is of type size_t.

+ ['const char *'] Like **s**, but concatenate to the previous string. Only valid after a string.

+# ['const char *', 'int'] Like **s#**, but concatenate to the previous string. Only valid after a string.

+% ['const char *', 'size_t'] Like **+#**, but the length argument is of type size_t.

n (null) Output a JSON null value. No argument is consumed.

b (boolean)['int'] Convert a C int to JSON boolean value. Zero is converted to *false* and non-zero to *true*.

i (integer)['int'] Convert a C int to JSON integer.

I (integer)['int64_t'] Convert a C int64_t to JSON integer. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert a C double to JSON real.

o (any value)['json_t *'] Output any given JSON value as-is. If the value is added to an array or object, the reference to the value passed to **o** is stolen by the container.

O (any value)['json_t *'] Like **o**, but the argument's reference count is incremented. This is useful if you pack into an array or object and want to keep the reference for the JSON value consumed by **O** to yourself.

[fmt] (array) Build an array with contents from the inner format string. **fmt** may contain objects and arrays, i.e. recursive value building is supported.

{fmt} (object) Build an object with contents from the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must be a string as object keys are always strings. The second, fourth, etc. format specifier represent a value. Any value may be an object or array, i.e. recursive value building is supported.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Building Values

1.2.72 DECODING JSON PAYLOADS

Flux API functions that are based on Jansson's json_unpack() accept the following tokens in their format string. The type in parenthesis denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

s (string)['const char *'] Convert a JSON string to a pointer to a null terminated UTF-8 string. The resulting string is extracted by using 'json_string_value()' internally, so it exists as long as there are still references to the corresponding JSON string.

n (null) Expect a JSON null value. Nothing is extracted.

b (boolean)['int'] Convert a JSON boolean value to a C int, so that *true* is converted to 1 and *false* to 0.

i (integer)['int'] Convert a JSON integer to a C int.

I (integer)['int64_t'] Convert a JSON integer to a C int64_t. Note: Jansson expects a json_int_t here without committing to a size, but Flux guarantees that this is a 64-bit integer.

f (real)['double'] Convert JSON real to a C double.

F (real)['double'] Convert JSON number (integer or real) to a C double.

o (any value)['json_t *'] Store a JSON value, with no conversion, to a json_t pointer.

O (any value)['json_t *'] Like **o**, but the JSON value's reference count is incremented.

[fmt] (array) Convert each item in the JSON array according to the inner format string. **fmt** may contain objects and arrays, i.e. recursive value extraction is supported.

{fmt} (object) Convert each item in the JSON object according to the inner format string **fmt**. The first, third, etc. format specifier represent a key, and must by **s**. The corresponding argument to unpack functions is read as the object key. The second, fourth, etc. format specifier represent a value and is written to the address given as the corresponding argument. Note that every other argument is read from and every other is written to. **fmt** may contain objects and arrays as values, i.e. recursive value extraction is supported. Any **s** representing a key may be suffixed with **?** to make the key optional. If the key is not found, nothing is extracted.

! This special format specifier is used to enable the check that all object and array items are accessed, on a per-value basis. It must appear inside an array or object as the last format specifier before the closing bracket or brace.

Whitespace, **:** (colon) and **,** (comma) are ignored.

These descriptions came from the Jansson 2.6 manual.

See also: Jansson API: Parsing and Validating Values

1.3 man5

1.3.1 flux-config-bootstrap(5)

DESCRIPTION

The broker discovers the size of the Flux instance, the broker's rank, and overlay network wireup information either dynamically using a PMI service, such as when being launched by Flux or another resource manager, or statically using the `bootstrap` section of the Flux configuration, such as when being launched by systemd.

The default bootstrap mode is PMI. To select config file bootstrap, specify the config directory with the `--config-path=PATH` broker command line option or set `FLUX_CONF_DIR` in the broker's environment. Ensure that this directory contains a file that defines the `bootstrap` section.

CONFIG FILES

Flux uses the TOML configuration file format. The `bootstrap` section is a TOML table containing the following keys. Each node in a cluster is expected to bootstrap from an identical config file.

KEYWORDS

enable_ipv6 (optional) Boolean value for enabling IPv6. By default only IPv4 is enabled. Note that setting this to true prevents binding to a named interface that only supports IPv4.

curve_cert (optional) Path to a CURVE certificate generated with flux-keygen(1). The certificate should be identical on all broker ranks. It is required for instance sizes > 1.

default_port (optional) The value is an integer port number that is substituted for the token `%p` in the other keys.

default_bind (optional) The value is a ZeroMQ endpoint URI that is used for host entries that do not explicitly set a bind address. The tokens `%p` and `%h` are replaced with the default port and the host for the current host entry.

default_connect (optional) The value is a ZeroMQ endpoint URI that is used for host entries that do not explicitly set a connect address. The tokens `%p` and `%h` are replaced with the default port and the host for the current host entry.

hosts (optional) A rank-ordered array of host entries. Each host entry is a TOML table containing at minimum the `host` key. The broker determines its rank by matching its hostname in the hosts array and taking the array index. An empty or missing hosts array implies a standalone (single broker) instance. The entry for a broker with downstream peers must either assign the `bind` key to a ZeroMQ endpoint URI, or the `default_bind` URI described above is used. The entry for a broker with downstream peers must also either assign the `connect` key to a ZeroMQ endpoint URI, or the `default_connect` URI described above is used. The same `%h` and `%p` substitutions work here as well.

COMPACT HOSTS

Since it would be tedious to repeat host entries for every compute node in a large cluster, the `hosts` array may be abbreviated using RFC29 hostlists. For example, the list of hosts foo0, foo1, foo2, foo3, foo18, foo4, foo20 can be represented as "foo[0-3,18,4,20]".

EXAMPLE

```
[bootstrap]

default_port = 8050
default_bind = "tcp://en0:%p"
default_connect = "tcp://e%h:%p"

hosts = [
    {
        host="fluke0",
        bind="tcp://en4:9001",
        connect="tcp://fluke-mgmt:9001"
    },
    { host = "fluke[1-1023]" },
]
```

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-getattr(1), flux_attr_get(3)

1.4 man7

1.4.1 flux-broker-attributes(7)

DESCRIPTION

Flux broker attributes are parameters that affect how different broker systems behave. Attributes can be listed and manipulated with flux-getattr(1), flux-setattr(1), and flux-lsattr(1).

The broker currently exports the following attributes:

SESSION ATTRIBUTES

rank The rank of the local broker.

size The number of broker ranks in the flux instance

rundir A temporary directory available for scratch storage within the session. By default, a temporary directory is created for each broker rank, but if rundir is set on the command line, this directory may be shared by all broker ranks running on the same node. If rundir is created by the broker, it is removed during session exit.

content.backing-path The path to the content backing store file(s). If this is set on the broker command line, the backing store uses this path instead of a temporary one, and content is preserved on instance exit. If file exists, its content is imported into the instance. If it doesn't exist, it is created.

TOPOLOGY ATTRIBUTES

tbon.fanout Branching factor of the tree based overlay network.

tbon.descendants Number of descendants "below" this node of the tree based overlay network, not including this node.

tbon.level The level of this node in the tree based overlay network. Root is level 0.

tbon.maxlevel The maximum level number in the tree based overlay network. Maxlevel is 0 for a size=1 instance.

tbon.endpoint The endpoint for the tree based overlay network to communicate over.

tbon.zmqdebug If set to an non-zero integer value, 0MQ socket event logging is enabled, if available. This is potentially useful for debugging overlay connectivity problems. The attribute may not be changed during runtime.

tbon.prefertcp If set to an integer value other than zero, and the broker is bootstrapping with PMI, tcp:// endpoints will be used instead of ipc://, even if all brokers are on a single node.

SOCKET ATTRIBUTES

tbon.parent-endpoint The URI of the ZeroMQ endpoint this rank is connected to in the tree based overlay network. This attribute will not be set on rank zero.

local-uri The Flux URI that should be passed to flux_open(1) to establish a connection to the local broker rank. By default, local-uri is created as "local://<broker.rank>/local".

parent-uri The Flux URI that should be passed to flux_open(1) to establish a connection to the enclosing instance.

LOGGING ATTRIBUTES

log-ring-used The number of log entries currently stored in the ring buffer.

log-ring-size The maximum number of log entries that can be stored in the ring buffer.

log-count The number of log entries ever stored in the ring buffer.

log-forward-level Log entries at syslog(3) level at or below this value are forwarded to rank zero for permanent capture.

log-critical-level Log entries at syslog(3) level at or below this value are copied to stderr on the logging rank, for capture by the enclosing instance.

log-filename (rank zero only) If set, session log entries, as filtered by log-forward-level, are directed to this file.

log-stderr-mode If set to "leader" (default), broker rank 0 emits forwarded logs from other ranks to stderr, subject to the constraints of log-forward-level and log-stderr-level. If set to "local", each broker emits its own logs to stderr, subject to the constraints of log-stderr-level.

log-stderr-level Log entries at syslog(3) level at or below this value to stderr, subject to log-stderr-mode.

log-level Log entries at syslog(3) level at or below this value are stored in the ring buffer.

CONTENT ATTRIBUTES

content.acct-dirty The number of dirty cache entries on this rank.

content.acct-entries The total number of cache entries on this rank.

content.acct-size The estimated total size in bytes consumed by cache entries on this rank, excluding overhead.

content.acct-valid The number of valid cache entries on this rank.

content.backing-module The selected backing store module, if any. This attribute is only set on rank 0 where the content backing store is active.

content.blob-size-limit The maximum size of a blob, the basic unit of content storage.

content.flush-batch-count The current number of outstanding store requests, either to the backing store (rank 0) or upstream (rank > 0).

content.flush-batch-limit The maximum number of outstanding store requests that will be initiated when handling a flush or backing store load operation.

content.hash The selected hash algorithm, default sha1.

content.purge-old-entry When the cache size footprint needs to be reduced, only consider purging entries that are older than this number of seconds.

content.purge-target-size If possible, the cache size purged periodically so that the total size of the cache stays at or below this value.

WIREUP ATTRIBUTES

hello.timeout The reduction timeout (in seconds) for the broker wireup protocol. Before the timeout, a topology-based high water mark is applied at each node of the tree based overlay network. After the timeout, new wireup information is forwarded upstream without delay. Set to 0 to disable the timeout.

hello.hwm The reduction high water mark for the broker wireup protocol, normally calculated based on the topology. Set to 0 to disable the high water mark.

CONFIG ATTRIBUTES

config.hostlist The rank-ordered hosts specified in the `bootstrap` section of the Flux configuration. Hosts are listed in RFC29 hostlist format.

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-getattr(1), flux_attr_get(3)

1.4.2 flux-jobtap-plugins(7)

DESCRIPTION

The *jobtap* interface supports loading of builtin and external plugins into the job manager broker module. These plugins can be used to assign job priorities using algorithms other than the default, assign job dependencies, aid in debugging of the flow of job states, or generically extend the functionality of the job manager.

Jobtap plugins are defined using the Flux standard plugin format. Therefore a jobtap plugin should export the single symbol: `flux_plugin_init()`, from which calls to `flux_plugin_add_handler(3)` should be used to register functions which will be called for the callback topic strings described in the *CALLBACK TOPICS* section below.

Each callback function uses the Flux standard plugin callback form, e.g.:

```
int callback (flux_plugin_t *p,
              const char *topic,
              flux_plugin_arg_t *args,
              void *arg);
```

where `p` is the handle for the current *jobtap* plugin, `topic` is the *topic string* for the currently invoked callback, `args` contains a set of plugin arguments which may be unpacked with the `flux_plugin_arg_unpack(3)` call, and `arg` is any opaque argument passed along when registering the handler.

Multiple plugins may be loaded in the job-manager simultaneously. In this case, all matching handlers are called in all loaded plugins in the order in which they were loaded. For more information about loading plugins see the *CONFIGURATION* section below or the `flux-jobtap(1)` manpage.

JOBTAP PLUGIN NAMES

Jobtap plugins are loaded into the job-manager and referenced in the output of `flux jobtap list` by file name. If a plugin is loaded by a fully qualified path, the plugin name is shortened to the basename, such that all dynamically loaded plugins have names such as `plugin-name.so`.

Builtin plugins, on the other hand, are named with a leading `.`, and are hidden in `flux jobtap list`, do not match the glob(7) `*` or "all" keyword, etc. (similar to hidden filesystem files). To list builtin plugins, use the `-a, --all` option to `flux jobtap list`, and to remove them use the name explicitly or include the leading `.` in any pattern.

A plugin may optionally assign a name with `flux_plugin_set_name(3)`, however this name is not displayed in `flux jobtap list` or used in matching. The internal plugin name is only used as part of the service name generated by `flux_jobtap_service_register()`, i.e. the service name will be `job-manager.<name>.<method>`. If a plugin does not set a name with `flux_plugin_set_name(3)`, then the basename of the plugin file will be used with the trailing `.so` removed.

JOBTAP PLUGIN ARGUMENTS

For job-specific callbacks, all job data is passed to the plugin via the `flux_plugin_arg_t *args`, and return data is sent back to the job manager via the same `args`. Incoming arguments may be unpacked using `flux_plugin_arg_unpack(3)`, e.g.:

```
rc = flux_plugin_arg_unpack (args, FLUX_PLUGIN_ARG_IN,
                             "{s{s:o}, s:I}",
                             "jobspec", "resources", &resources,
                             "id", &id);
```

will unpack the `resources` section of jobspec and the jobid into `resources` and `id` respectively.

The full list of available args includes the following:

name	type	description
jobspec	o	jobspec with environment redacted
id	I	jobid
state	i	current job state
prev_state	i	previous state (`job.state.*` callbacks)
userid	i	userid
urgency	i	current urgency
priority	I	current priority
t_submit	f	submit timestamp in floating point seconds
entry	o	posted eventlog entry, including context

Return arguments can be packed using the `FLUX_PLUGIN_ARG_OUT` and optionally `FLUX_PLUGIN_ARG_REPLACE` flags. For example to return a priority:

```
rc = flux_plugin_arg_pack (args, FLUX_PLUGIN_ARG_OUT,
                           "{s:I}",
                           "priority", (int64_t) priority);
```

While a job is pending, *jobtap* plugin callbacks may also add job annotations by returning a value for the `annotations` key:

```
flux_plugin_arg_pack (args, FLUX_PLUGIN_ARG_OUT,
                      "{s:{s:s}}",
                      "annotations", "test", value);
```

CALLBACK TOPICS

The following callback "topic strings" are currently provided by the *jobtap* interface:

job.validate The `job.validate` topic allows a plugin to reject a job before it is introduced to the job manager. A rejected job will result in a job submission error in the submitting client, and any job data in the KVS will be purged. No further callbacks will be made for rejected jobs. Note: If a job is not rejected, then the `job.new` callback will be invoked immediately after `job.validate`. This allows limits or other validation to be implemented in the `job.validate` callback, but accounting for those limits should be confined to the `job.new` callback, since `job.new` may also be called during job-manager restart or plugin reload.

job.dependency.* The `job.dependency.*` topic allows a dependency plugin to notify the job-manager that it handles a given dependency _scheme_. The job-manager will scan the `attirbutes.system.dependencies` array, if provided, and issue a `job.dependency.SCHEME` callback for each listed dependency. If no plugin has registered for `SCHEME`, then the job is rejected. The plugin should then call `flux_jobtap_dependency_add(3)` to add a new named dependency to the job (if necessary). Jobs with dependencies will remain in the `DEPEND` state until all dependencies are removed with a corresponding call to `flux_jobtap_dependency_remove(3)`. See `job.state.depend` below for more information about dependencies. If there is an error in the dependency specification, the job may be rejected with `flux_jobtap_reject_job(3)` and a negative return code from the callback.

job.new The `job.new` topic is used by the job manager to notify a jobtap plugin about a newly introduced job. This call may be made in three different situations:

1. on job job submission
2. when the job manager is restarted and has reloaded a job from the KVS

3. when a new jobtap plugin is loaded

In case 1 above, the job state will always be `FLUX_JOB_STATE_NEW`, while jobs in cases 2 and 3 can be in any state except `FLUX_JOB_STATE_INACTIVE`.

job.state.* The `job.state.*` callbacks are made just after a job state transition. The callback is made after the state has been published to the job's eventlog, but before any action has been taken on that state (since the action could involve immediately transitioning to a new state)

job.event.* The `job.event.*` callbacks are only made for plugins that have explicitly subscribed to a job with `flux_jobtap_job_subscribe()`. In this case, all job events result in this callback being invoked on all subscribed plugins. This may be useful for plugins to get notification of events that do not necessarily result in a state transition, e.g. the `start` event or a non-fatal `exception`.

job.state.depend The callback for `FLUX_JOB_STATE_DEPEND` is the final place from which a plugin may add dependencies to a job. Dependencies are added via the `flux_jobtap_dependency_add()` function. This function allows a named dependency to be attached to a job. Jobs with dependencies will remain in the `DEPEND` state until all dependencies are removed with a corresponding call the `flux_jobtap_dependency_remove()`. A dependency may only be used once. A second call to `flux_jobtap_dependency_add()` with the same dependency description will return `EEXIST`, even if the dependency was subsequently removed. (This allows idempotent operation of plugin-managed dependencies for job-manager or plugin restart).

job.state.priority The callback for `FLUX_JOB_STATE_PRIORITY` is special, in that a plugin must return a priority at the end of the callback (if the plugin is a priority-managing plugin). If the job priority is not available, the plugin should use `flux_jobtap_priority_unavail()` to indicate that the priority cannot be set. Jobs that do not have a priority due to unavailable priority or when no current priority plugin is loaded will remain in the PRIORITY state until a priority is assigned. Therefore, a plugin should arrange for the priority to be set asynchronously using `flux_jobtap_reprioritize_job()`. See the *PRIORITY* section for more detailed information about plugin management of job priority.

job.priority.get The job manager calls the `job.priority.get` topic whenever it wants to update the job priority of a single job. The plugin should return a priority immediately, but if one is not available when a job is in the PRIORITY state, the plugin may use `flux_jobtap_priority_unavail()` to indicate the priority is not available. Returning an unavailable priority in the SCHED state is an error and it will be logged, but otherwise ignored. A call of `job.priority.get` can be requested for all jobs by calling `flux_jobtap_reprioritize_all()`. See the *PRIORITY* section for more information about plugin management of job priority.

PRIORITY

Custom assignment of job priority values is one of the core features supported by the jobtap plugin interface. A builtin `.priority-default` plugin is always loaded in the job-manager to ensure that jobs move past the PRIORITY state when no other priority plugin is loaded. The default plugin simply assigns the priority to the same value as the current job urgency.

When loading a new jobtap plugin that assigns priority, it is important to be cognizant of the fact that the `.priority-default` plugin may still be loaded. This will result in the `priority` set in the return arguments to always be initialized to the job urgency. However, since plugin `job.state.priority` and `job.priority.get` callbacks are run in order, any subsequently loaded plugin that assigns a priority will overwrite the returned default `priority` and thus the last loaded priority plugin will be active.

To ensure the default priority is always overridden priority plugins should therefore make sure to always set a priority, or use `flux_jobtap_priority_unavail()` if the priority is not available, in any callback in which a priority is expected to be returned, i.e. `job.state.priority` and `job.priority.get`.

To fully ensure priority plugins do not conflict, the builtin priority plugin may explicitly be removed with

```
flux jobtap remove .priority-default
```

or via configuration (See *CONFIGURATION* below)

```
[job-manager]
plugins = [
  { remove = ".priority-default",
    load = "complex-priority.so"
  },
]
```

CONFIGURATION

Job-manager plugin configuration is defined in the `job-manager.plugins` section of the Flux TOML configuration file. This section is an array of plugin directives which include the following keys:

load Load a plugin matching the given filename into the job-manager. If the path is not absolute, then the first plugin matching the job-manager searchpath will be loaded.

conf With load only, pass an optional configuration table to the loaded plugin.

remove Remove all plugins matching the value. The value may be a glob(7). If `remove` appears with `load`, plugin removal is always handled first. The special value `all` is a synonym for `*`, but will not error when no plugins match.

For example

```
[job-manager]
plugins = [
   {
     load = "priority-custom.so",
     conf = {
        job-limit = 100,
        size-limit = 128
     }
   }
]
```

The list of loaded jobtap plugins may also be queried and controlled at runtime with the `flux-jobtap(1)` command

RESOURCES

Github: http://github.com/flux-framework

SEE ALSO

flux-jobtap(1)

CHAPTER 2

flux-core Python Bindings

2.1 python

2.1.1 flux package

python bindings to flux-core, the main core of the flux resource manager

`flux.`**`constants`**
Used by autodoc_mock_imports.

`flux.`**`Flux`**(**args*, ***kwargs*)

Subpackages

flux.core package

Submodules

flux.core.handle module

class `flux.core.handle.`**`Flux`**(*url=<sphinx.ext.autodoc.importer._MockObject object>*, *flags=0*, *handle=None*)

Bases: *flux.wrapper.Wrapper*

The general Flux handle class, create one of these to connect to the nearest enclosing flux instance

Example

```
>>> flux.Flux()
<flux.core.Flux object at 0x...>
```

attr_get(*attr_name*)

barrier(*name*, *nprocs*)

close()
The underlying flux handle is automatically closed when a Flux instance is deconstructed. Prevent users from manually closing, the handle, leading to a double free.

event_create(*topic*, *payload=None*)
Create a new event message.

Parameters

- **topic** – A string, the event's topic
- **payload** – If a string, the payload is used unmodified, if it is another type json.dumps() is used to stringify it

event_recv(*topic=None*)

event_send(*topic*, *payload=None*)
Create and send a new event in one step

event_subscribe(*topic*)
Subscribe to events

Parameters **topic** (*str, bytes, or unicode*) – The event's topic to subscribe to

Raises

- **EnvironmentError** – if the topic is None or NULL
- **TypeError** – if the topic is not a str, bytes, or unicode

fd_watcher_create(*fd_int*, *callback*, *events=None*, *args=None*)

get_rank()

in_reactor()

log(*level*, *fstring*)
Log to the flux logging facility

Parameters

- **level** – A syslog log-level, check the syslog module for possible values
- **fstring** – A string to log, C-style formatting is *not* supported

msg_watcher_create(*callback*, *type_mask=<bound method Wrapper.__getattr__.<locals>.wrap_class of <flux.core.inner.Core object>>*, *topic_glob='*'*, *args=None*, *match_tag=<bound method Wrapper.__getattr__.<locals>.wrap_class of <flux.core.inner.Core object>>*)

classmethod raise_if_exception()
Re-raise any class global exception if set

If a global exception is currently set for the Flux handle class, re-raise it and reset the exception state to None.

The exception is raised `from None` to preserve the original stack trace.

reactor_decref(*reactor=None*)

classmethod reactor_enter()

classmethod reactor_exit()

reactor_incref(*reactor=None*)

reactor_run(*reactor=None*, *flags=0*)
Run reactor associated with this Flux handle or reactor argument if it is provided. Sets a signal watcher for SIGINT to return from the reactor on Ctrl-C, and raise KeyboardInterrupt.

classmethod reactor_running()
Return True if this thread is running the Flux reactor

reactor_stop(*reactor=None*)

reactor_stop_error(*reactor=None*)

recv(*type_mask=<bound method ? of <flux.core.inner.Core object>>*, *match_tag=<bound method ? of <flux.core.inner.Core object>>*, *topic_glob=None*, *flags=0*)
Receive a message, returns a flux.Message containing the result or None

respond(*message*, *payload=None*)
Respond to a flux rpc

Parameters

- **message** (*Message*) – The message to respond to
- **payload** (*None, str, bytes, unicode, or json-serializable*) – The (optional) payload to include in the response

rpc(*topic*, *payload=None*, *nodeid=<bound method ? of <flux.core.inner.Core object>>*, *flags=0*)
Create a new RPC object

send(*message*, *flags=0*)
Send a pre-constructed flux message

service_register(*name*)

service_unregister(*name*)

classmethod set_exception(*exception*)
Set a global, per-thread exception for Flux

This class method allows Python callbacks called from the Flux reactor to set a global exception which can be re-thrown after the return to Python (when reactor_run() returns). This is implemented as a class attribute since the Flux handle object which is available in a Python callback from C will be a different instantiation than the Flux handle object which started the reactor (with the same underlying flux_t however)

Parameters **exception** (*Exception*) – A reference to the exception thrown.

Returns The previously set exception, or None

Return type Exception

signal_watcher_create(*signum*, *callback*, *args=None*)

timer_watcher_create(*after*, *callback*, *repeat=0.0*, *args=None*)

tls = <_thread._local object>

flux.core.inner module

class flux.core.inner.**Core**
Bases: *flux.wrapper.Wrapper*

Generic Core wrapper, you probably do not want or need one of these.

FLUX_MATCHTAG_NONE (**args*, ***kwargs*)

FLUX_MSGTYPE_ANY (**args*, ***kwargs*)

FLUX_NODEID_ANY (**args*, ***kwargs*)

flux_future_destroy (**args*, ***kwargs*)

flux.core.trampoline module

flux.core.trampoline.**mod_main_trampoline**(*name*, *int_handle*, *args*)

flux.core.watchers module

class flux.core.watchers.**TimerWatcher**(*flux_handle*, *after*, *callback*, *repeat=0*, *args=None*)
Bases: flux.core.watchers.Watcher

class flux.core.watchers.**FDWatcher**(*flux_handle*, *fd_int*, *events*, *callback*, *args=None*)
Bases: flux.core.watchers.Watcher

class flux.core.watchers.**SignalWatcher**(*flux_handle*, *signal_int*, *callback*, *args=None*)
Bases: flux.core.watchers.Watcher

flux.job package

Subpackages

flux.job.validator package

Submodules

flux.job.validator.validator module

class flux.job.validator.validator.**JobValidator**(*argv*, *pluginpath=None*, *parser=None*)
Bases: object

A plugin-based job validator class

JobValidator loads plugins that implenment the ValidatorPlugin interface from the 'flux.job.validator.plugins' namespace. Plugins may be configured at runtime by passing in a `--plugins=LIST` option

default_validators = ['jobspec']

plugin_namespace = 'flux.job.validator.plugins'

start()
Select and configure plugins, start executor, etc.

validate(*jobinfo*)
Validate jobinfo using all loaded validators

Parameters **jobinfo** (*ValidatorJobInfo*) – A ValidatorJobInfo object which describes the job to be validated.

Returns

ValidatorResult

If any one validator plugin fails, then result will indicate failure.

class flux.job.validator.validator.**ValidatorJobInfo**(*jobinfo*)

Bases: object

An instance of a Flux job specification used by job validators

jobspec

Submitted jobspec in Python dict form

Type dict

userid

Submitting user id

Type int

flags

Job flags supplied during submission

Type int

urgency

Job urgency

Type int

flux

On-demand, per-thread Flux handle

Type Flux

tls = <_thread._local object>

class flux.job.validator.validator.**ValidatorPlugin**(*parser*)

Bases: abc.ABC

Base class for Validator Plugins

configure(*args*)

Configure a ValidatorPlugin. Run after argparse.parse_args()

Parameters

- **args** (Namespace) – The resulting Namespace after calling
- **argparse.parse_args()** –

validate(*job*)

Validate a job. A ValidatorPlugin must implement this method

If a job fails validation, this method should either throw an exception, which will be caught by the calling script, or a (errnum, errmsg) tuple may optionally be returned, if that is more convenient.

On success, this method should return nothing or explicitly:

```
(0, None)
```

Parameters **job** (*ValidatorJobInfo*) – the job to validate

Returns None or (errnum, errmsg) tuple.

class flux.job.validator.validator.**ValidatorResult**
Bases: object

Container for result or results from the JobValidator validate method

errmsg
comma-separated string list of all error messages

Type str

push_result(*errnum*, *errmsg=None*)
Add a result from one validator to a ValidatorResult

Parameters

- **errnum** (int) – error number (0 for success)
- **errmsg** (str, optional) – An optional error messsage for a failed result.

success
True if job validated successfully, False otherwise

Type bool

flux.job.validator.validator.**import_path**(*file_path*)

flux.job.validator.validator.**import_plugins**(*pkg_name*, *pluginpath=None*)
Load plugins from a namespace package and optional additional paths

A plugin in pluginpath with the same name as an existing plugin will take precedence

flux.job.validator.validator.**import_plugins_pkg**(*ns_pkg*)
Import all modules found in the namespace package ns_pkg

Submodules

flux.job.JobID module

class flux.job.JobID.**JobID**
Bases: int

Class used to represent a Flux JOBID

JobID is a subclass of *int*, so may be used in place of integer. However, a JobID may be created from any valid RFC 19 FLUID encoding, including:

- decimal integer (no prefix)
- hexidecimal integer (prefix 0x)
- dotted hex (dothex) (xxxx.xxxx.xxxx.xxxx)
- kvs dir (dotted hex with *job.* prefix)
- RFC19 F58: (Base58 encoding with prefix *ƒ* or *f*)

A JobID object also has properties for encoding a JOBID into each of the above representations, e.g. jobid.f85, jobid.words, jobid.dothex...

dec
Return decimal integer representation of a JobID

dothex
Return dotted hexidecimal representation of a JobID

encode(*encoding='dec'*)
Encode a JobID to alternate supported format

f58
Return RFC19 F58 representation of a JobID

hex
Return 0x-prefixed hexidecimal representation of a JobID

kvs
Return KVS directory path of a JobID

orig
Return the original string used to create the JobID

words
Return words (mnemonic) representation of a JobID

flux.job.JobID.**id_encode**(*jobid*, *encoding='f58'*)
returns: Jobid encoded in encoding :rtype str

flux.job.JobID.**id_parse**(*jobid_str*)
returns: An integer jobid :rtype int

flux.job.Jobspec module

class flux.job.Jobspec.**Jobspec**(*resources*, *tasks*, ***kwargs*)
Bases: object

attributes

cwd
Get working directory of job.

dumps(***kwargs*)

duration

environment
Get (entire) environment of job.

classmethod from_yaml_file(*filename*)

classmethod from_yaml_stream(*yaml_stream*)

resource_counts()
Compute the counts of each resource type in the jobspec

The following jobspec would return `{ "slot": 12, "core": 18, "memory": 242 }`

```
- type: slot
  count: 2
  with:
    - type: core
      count: 4
    - type: memory
      count: 1
      unit: GB
- type: slot
  count: 10
  with:
```

(continues on next page)

(continued from previous page)

```
    - type: core
      count: 1
    - type: memory
      count: 24
      unit: GB
```

Note: the current implementation ignores the *unit* label and assumes they are consist across resources

resource_walk()
Traverse the resources in the *resources* section of the jobspec.

Performs a depth-first, pre-order traversal. Yields a tuple containing (parent, resource, count). *parent* is None when *resource* is a top-level resource. *count* is the number of that resource including the multiplicative effects of the *with* clause in ancestor resources. For example, the following resource section, will yield a count of 2 for the *slot* and a count of 8 for the *core* resource:

```
- type: slot
  count: 2
  with:
    - type: core
      count: 4
```

resources

setattr(*key*, *val*)
set job attribute

setattr_shell_option(*key*, *val*)
set job attribute: shell option

stderr

stdin

stdout

tasks

top_level_keys = {'attributes', 'resources', 'tasks', 'version'}

version

class flux.job.Jobspec.**JobspecV1**(*resources*, *tasks*, ***kwargs*)
Bases: *flux.job.Jobspec.Jobspec*

classmethod from_batch_command(*script*, *jobname*, *args=None*, *num_slots=1*, *cores_per_slot=1*, *gpus_per_slot=None*, *num_nodes=None*, *broker_opts=None*)
Create a Jobspec describing a nested Flux instance controlled by a script.

The nested Flux instance will execute the script with the given command-line arguments after copying it and setting the executable bit. Conceptually, this differs from the *from_nest_command*, which also creates a nested Flux instance, in that it a) requires the initial program of the new instance to be an executable text file and b) creates the initial program from a string rather than using an executable existing somewhere on the filesystem.

Use setters to assign additional properties.

Parameters

- **script** – contents of the script to execute, as a string. The script should have a shebang (e.g. *#!/bin/sh*) at the top.
- **jobname** (*str*) – name to use as the argv[0] for this job. This will be the default job name reported by Flux. (Note the actual argv is overridden by the job shell when executed.)
- **args** (iterable of *str*) – arguments to pass to *script*
- **num_slots** – number of resource slots to create. Slots are an abstraction, and are only used (along with *cores_per_slot* and *gpus_per_slot*) to determine the nested instance's allocation size and layout.
- **cores_per_slot** – number of cores to allocate per slot
- **gpus_per_slot** – number of GPUs to allocate per slot
- **num_nodes** – distribute allocated resource slots across N individual nodes
- **broker_opts** (*iterable of str*) – options to pass to the new Flux broker

classmethod from_command(*command*, *num_tasks=1*, *cores_per_task=1*, *gpus_per_task=None*, *num_nodes=None*)

Factory function that builds the minimum legal v1 jobspec.

Use setters to assign additional properties.

Parameters

- **command** (*iterable of str*) – command to execute
- **num_tasks** – number of MPI tasks to create
- **cores_per_task** – number of cores to allocate per task
- **gpus_per_task** – number of GPUs to allocate per task
- **num_nodes** – distribute allocated tasks across N individual nodes

classmethod from_nest_command(*command*, *num_slots=1*, *cores_per_slot=1*, *gpus_per_slot=None*, *num_nodes=None*, *broker_opts=None*)

Create a Jobspec describing a nested Flux instance controlled by *command*.

Conceptually, this differs from the *from_batch_command* method in that a) the initial program of the nested Flux instance can be any executable on the file system, not just a text file and b) the executable is not copied at submission time.

Use setters to assign additional properties.

Parameters

- **command** (*iterable of str*) – initial program for the nested Flux instance
- **num_slots** – number of resource slots to create. Slots are an abstraction, and are only used (along with *cores_per_slot* and *gpus_per_slot*) to determine the nested instance's allocation size and layout.
- **cores_per_slot** – number of cores to allocate per slot
- **gpus_per_slot** – number of GPUs to allocate per slot
- **num_nodes** – distribute allocated resource slots across N individual nodes
- **broker_opts** (*iterable of str*) – options to pass to the new Flux broker

flux.job.Jobspec.**validate_jobspec**(*jobspec*, *require_version=None*)

Validates the jobspec by attempting to construct a Jobspec object. If no exceptions are thrown during construction, then the jobspec is assumed to be valid and this function returns True. If the jobspec is invalid, the relevant exception is thrown (i.e., TypeError, ValueError, EnvironmentError)

By default, the validation function will read the *version* key in the jobspec to determine which Jobspec object to instantiate. An optional *require_version* is included to override this behavior and force a particular class to be used.

Parameters

- **jobspec** – a Jobspec object or JSON string
- **require_version** – jobspec version to use, if not provided, the value of jobspec['version'] is used

Raises

- **ValueError** –
- **TypeError** –
- **EnvironmentError** –

flux.job.event module

class flux.job.event.**EventLogEvent**(*event*)

Bases: object

wrapper class for a single KVS EventLog entry

context

name

timestamp

class flux.job.event.**JobEventWatchFuture**(*future_handle*)

Bases: *flux.future.Future*

A future returned from job.event_watch_async(). Adds get_event() method to return an EventLogEntry event

cancel()

Cancel a streaming job.event_watch_async() future

get_event(*autoreset=True*)

Return the next event from a JobEventWatchFuture, or None if the event stream has terminated.

The future is auto-reset unless autoreset=False, so a subsequent call to get_event() will try to fetch the next event and thus may block.

exception flux.job.event.**JobException**(*event*)

Bases: Exception

Represents an 'exception' event occurring to a job.

Instances expose a few public attributes.

Variables

- ***timestamp*** – the timestamp of the 'exception' event.
- ***type*** – A string identifying the type of job exception.
- **note** – Brief human-readable explanation of the exception.

- **severity** – the severity of the exception. Exceptions with a severity of 0 are fatal to the job; any other severity is non-fatal.

flux.job.event.**event_wait**(*flux_handle*, *jobid*, *name*, *eventlog='eventlog'*, *raiseJobException=True*)

Wait for a job eventlog entry 'name'

Wait synchronously for an eventlog entry named "name" and return the entry to caller, raises OSError with ENODATA if event never occurred

See also:

21/Job States and Events Documentation for the events in the main eventlog

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobid** – the job ID on which to wait for eventlog events
- **name** – The event name for which to wait
- **eventlog** – eventlog path in job kvs directory (default: eventlog)
- **raiseJobException** – if True, watch for job exception events and raise a JobException if one is seen before event 'name' (default=True)

Returns an EventLogEntry object, or raises OSError if eventlog ended before matching event was found

Return type EventLogEntry

flux.job.event.**event_watch**(*flux_handle*, *jobid*, *eventlog='eventlog'*)

Python generator to watch all events for a job

Synchronously watch events a job eventlog via a simple generator.

Example

```
>>> for event in job.event_watch(flux_handle, jobid):
...     # do something with event
```

See also:

21/Job States and Events Documentation for the events in the main eventlog

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobid** – the job ID on which to watch events
- **eventlog** – eventlog path in job kvs directory (default: eventlog)

flux.job.event.**event_watch_async**(*flux_handle*, *jobid*, *eventlog='eventlog'*)

Asynchronously get eventlog updates for a job

Asynchronously watch the events of a job eventlog.

Returns a JobEventWatchFuture. Call .get_event() from the then callback to get the currently returned event from the Future object.

See also:

21/Job States and Events Documentation for the events in the main eventlog

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobid** – the job ID on which to watch events
- **eventlog** – eventlog path in job kvs directory (default: eventlog)

Returns a JobEventWatchFuture object

Return type *JobEventWatchFuture*

flux.job.executor module

This module defines the FluxExecutor and FluxExecutorFuture classes.

class flux.job.executor.**FluxExecutor**(*threads=1*, *thread_name_prefix=''*, *poll_interval=0.1*, *handle_args=()*, *handle_kwargs={}*)

Bases: object

Provides a method to submit and monitor Flux jobs asynchronously.

Forks threads to complete futures and fetch event updates in the background.

Inspired by the concurrent.futures.Executor class, with the following interface differences:

- the submit method takes a flux.job.Jobspec instead of a callable and its arguments, and returns a FluxExecutorFuture representing that job.
- the map method is not supported, given that the executor consumes Jobspecs rather than callables.

Otherwise, the FluxExecutor is faithful to its inspiration. In addition to methods and behavior defined by concurrent.futures, FluxExecutor provides its futures with event updates and the jobid of the underlying job.

Futures returned by submit have their jobid set as soon as it is available, which is always before the future completes.

The executor can also monitor existing jobs through the attach method, which takes a job ID and returns a future representing the job.

Futures may receive event updates even after they complete. The names of valid events are contained in the EVENTS class attribute.

The result of a future is the highest process exit status of the underlying job (in which case the result is an integer greater than or equal to 0), or -signum where signum is the number of the signal that caused the process to terminate (in which case the result is an integer less than 0).

A future is marked as "running" (and can no longer be canceled using the .cancel() method) once it reaches a certain point in the Executor—a point which is completely unrelated to the status of the underlying Flux job. The underlying Flux job may still be canceled at any point before it terminates, however, using the flux.job.cancel and flux.job.kill functions, in which case a JobException will be set.

If the jobspec is invalid, an OSError is set.

Parameters

- **threads** – the number of worker threads to fork.

- **thread_name_prefix** – used to control the names of `threading.Thread` objects created by the executor, for easier debugging.
- **poll_interval** – the interval (in seconds) in which to break out of the flux event loop to check for new job submissions.
- **handle_args** – positional arguments to the `flux.Flux` instances used by the executor.
- **handle_kwargs** – keyword arguments to the `flux.Flux` instances used by the executor.

EVENTS = frozenset({'urgency', 'start', 'depend', 'flux-restart', 'clean', 'submit', '
A set containing valid event names for attaching to futures.

attach(*jobid*)
Attach a `FluxExecutorFuture` to an existing job ID and return it.

Returned futures will behave identically to futures returned by the `FluxExecutor.submit` method. If the job ID is not accepted by Flux an exception will be set on the future.

This method is primarily useful for monitoring jobs that have been submitted through other mechanisms.

Parameters **jobid** (*int*) – jobid to attach to.

Raises **RuntimeError** – if `shutdown` has been called or if an error has occurred and new jobs cannot be submitted (e.g. a remote Flux instance can no longer be communicated with).

shutdown(*wait=True*, *, *cancel_futures=False*)
Clean-up the resources associated with the Executor.

It is safe to call this method several times. Otherwise, no other methods can be called after this one.

Parameters

- **wait** – If `True`, then this method will not return until all running futures have finished executing and the resources used by the executor have been reclaimed.
- **cancel_futures** – If `True`, this method will cancel all pending futures that the executor has not started running. Any futures that are completed or running won't be cancelled, regardless of the value of `cancel_futures`.

submit(**args*, ***kwargs*)
Submit a jobspec to Flux and return a `FluxExecutorFuture`.

Accepts the same positional and keyword arguments as `flux.job.submit`, except for the `flux.job.submit` function's first argument, `flux_handle`.

Parameters

- **jobspec** (*Jobspec or its string encoding*) – jobspec defining the job request
- **urgency** (*int*) – job urgency 0 (lowest) through 31 (highest) (default is 16). Priorities 0 through 15 are restricted to the instance owner.
- **waitable** (*bool*) – allow result to be fetched with `flux.job.wait()` (default is False). Waitable=True is restricted to the instance owner.
- **debug** (*bool*) – enable job manager debugging events to job eventlog (default is False)
- **pre_signed** (*bool*) – jobspec argument is already signed (default is False)

Raises **RuntimeError** – if `shutdown` has been called or if an error has occurred and new jobs cannot be submitted (e.g. a remote Flux instance can no longer be communicated with).

class flux.job.executor.**FluxExecutorFuture**(*owning_thread_id*, **args*, ***kwargs*)
Bases: concurrent.futures._base.Future

A concurrent.futures.Future subclass that represents a single Flux job.

In addition to all of the concurrent.futures.Future functionality, FluxExecutorFuture instances offer:

- The jobid and add_jobid_callback methods for retrieving the Flux jobid of the underlying job.
- The add_event_callback method to invoke callbacks when particular job-state events occur.

Valid events are contained in the EVENTS class attribute.

EVENTS = frozenset({'urgency', 'start', 'depend', 'flux-restart', 'clean', 'submit', '
A set containing the names of valid events.

add_done_callback(**args*, ***kwargs*)
Attaches a callable that will be called when the future finishes.

Parameters **fn** – A callable that will be called with this future as its only argument when the future completes or is cancelled. The callable will always be called by a thread in the same process in which it was added. If the future has already completed or been cancelled then the callable will be called immediately. These callables are called in the order that they were added.

Returns self

add_event_callback(*event*, *callback*)
Add a callback to be invoked when an event occurs.

The callback will be invoked, with the future as the first argument and the flux.job.EventLogEvent as the second, whenever the event occurs. If the event occurs multiple times, the callback will be invoked with each different *EventLogEvent* instance. If the event never occurs, the callback will never be invoked.

Added callables are called in the order that they were added and may be called in another thread. If the callable raises an Exception subclass, it will be logged and ignored. If the callable raises a BaseException subclass, the behavior is undefined.

If the event has already occurred, the callback will be called immediately.

Parameters

- **event** – the name of the event to add the callback to.
- **callback** – a callable taking the future and the event as arguments.

Returns self

add_jobid_callback(*callback*)
Attaches a callable that will be called when the jobid is ready.

Added callables are called in the order that they were added and may be called in another thread. If the callable raises an Exception subclass, it will be logged and ignored. If the callable raises a BaseException subclass, the behavior is undefined.

Parameters **callback** – a callable taking the future as its only argument.

Returns self

cancel(**args*, ***kwargs*)
Cancel the future if possible.

Returns True if the future was cancelled, False otherwise. A future cannot be cancelled if it is running or has already completed.

exception(**args*, ***kwargs*)
Return the exception raised by the call that the future represents.

Parameters **timeout** – The number of seconds to wait for the exception if the future isn't done. If None, then there is no limit on the wait time.

Returns The exception raised by the call that the future represents or None if the call completed without raising.

Raises

- `CancelledError` – If the future was cancelled.
- `TimeoutError` – If the future didn't finish executing before the given timeout.

jobid(*timeout=None*)
Return the jobid of the Flux job that the future represents.

Parameters **timeout** – The number of seconds to wait for the jobid. If None, then there is no limit on the wait time.

Returns a positive integer jobid.

Raises

- **concurrent.futures.TimeoutError** – If the jobid is not available before the given timeout.
- **concurrent.futures.CancelledError** – If the future was cancelled.
- **RuntimeError** – If the job could not be submitted (e.g. if the jobspec was invalid).

result(**args*, ***kwargs*)
Return the result of the call that the future represents.

Parameters **timeout** – The number of seconds to wait for the result if the future isn't done. If None, then there is no limit on the wait time.

Returns The result of the call that the future represents.

Raises

- `CancelledError` – If the future was cancelled.
- `TimeoutError` – If the future didn't finish executing before the given timeout.
- `Exception` – If the call raised then that exception will be raised.

set_exception(*exception*)
Sets the result of the future as being the given exception.

Should only be used by Executor implementations and unit tests.

flux.job.info module

class flux.job.info.**AnnotationsInfo**(*annotationsDict*)
Bases: `object`

class flux.job.info.**ExceptionInfo**(*occurred*, *severity*, *_type*, *note*)
Bases: `object`

class flux.job.info.**InfoList**
Bases: `list`

Extend list with string representation appropriate for JobInfo format

class flux.job.info.**JobInfo**(*info_resp*)
Bases: object

JobInfo class: encapsulate job-list.list response in an object that implements a getattr interface to job information with memoization. Better for use with output formats since results are only computed as-needed.

defaults = {'expiration': 0.0, 'nnodes': '', 'nodelist': '', 'priority': '', 'r

get_remaining_time()

get_runtime()

result

result_abbrev

returncode
The job return code if the job has exited, or an empty string if the job is still active. The return code of a job is the highest job shell exit code, or the negative signal number if the job shell was terminated by a signal. For jobs that were canceled before the RUN state, the return code will be set to -128.

runtime

state

state_single

status

status_abbrev

t_remaining

username

class flux.job.info.**JobInfoFormat**(*fmt*)
Bases: flux.util.OutputFormat

Store a parsed version of an output format string for JobInfo objects, allowing the fields to iterated without modifiers, building a new format suitable for headers display, etc...

class HeaderFormatter
Bases: flux.job.info.JobFormatter

Custom formatter for flux-jobs(1) header row.

Override default formatter behavior of calling getattr() on dotted field names. Instead look up header literally in kwargs. This greatly simplifies header name registration as well as registration of "valid" fields.

get_field(*field_name*, *args*, *kwargs*)
Override get_field() so we don't do the normal gettatr thing

class JobFormatter
Bases: string.Formatter

convert_field(*value*, *conv*)
Flux job-specific field conversions. Avoids the need to create many different format field names to represent different conversion types. (mainly used for time-specific fields for now).

format_field(*value*, *spec*)

format(*obj*)
format object with our JobFormatter

header()
format header with custom HeaderFormatter

headings = {'annotations': 'ANNOTATIONS', 'annotations.sched.reason_pending': 'REASO

flux.job.info.**fsd**(*secs*)

flux.job.info.**get_username**(*userid*)

flux.job.info.**resulttostr**(*resultid*, *singlechar=False*)

flux.job.info.**statetostr**(*stateid*, *singlechar=False*)

flux.job.info.**statustostr**(*stateid*, *resultid*, *abbrev=False*)

flux.job.kill module

flux.job.kill.**cancel**(*flux_handle: flux.core.handle.Flux, jobid: Union[flux.job.JobID.JobID, int], reason: Optional[str] = None*)
Cancel a pending or or running job

Parameters

- **flux_handle** – handle for Flux broker from flux.Flux()
- **jobid** – the job ID of the job to cancel
- **reason** – the textual reason associated with the cancelation

flux.job.kill.**cancel_async**(*flux_handle: flux.core.handle.Flux, jobid: Union[flux.job.JobID.JobID, int], reason: Optional[str] = None*)
Cancel a pending or or running job asynchronously

Parameters

- **flux_handle** – handle for Flux broker from flux.Flux()
- **jobid** – the job ID of the job to cancel
- **reason** – the textual reason associated with the cancelation

Returns a future fulfilled when the cancelation completes

Return type *Future*

flux.job.kill.**kill**(*flux_handle: flux.core.handle.Flux, jobid: Union[flux.job.JobID.JobID, int], signum: Optional[int] = None*)
Send a signal to a running job.

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobid** – the job ID of the job to kill
- **signum** – signal to send (default SIGTERM)

flux.job.kill.**kill_async**(*flux_handle: flux.core.handle.Flux, jobid: Union[flux.job.JobID.JobID, int], signum: Optional[int] = None*)
Send a signal to a running job asynchronously

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobid** – the job ID of the job to kill

- **signum** – signal to send (default SIGTERM)

Returns a Future

Return type *Future*

flux.job.kvs module

flux.job.kvs.**job_kvs**(*flux_handle*, *jobid*)

Returns The KVS directory of the given job

Return type *KVSDir*

flux.job.kvs.**job_kvs_guest**(*flux_handle*, *jobid*)

Returns The KVS guest directory of the given job

Return type *KVSDir*

flux.job.list module

class flux.job.list.**JobList**(*flux_handle, attrs=['userid', 'urgency', 'priority', 't_submit', 't_depend', 't_run', 't_cleanup', 't_inactive', 'state', 'name', 'ntasks', 'nnodes', 'ranks', 'nodelist', 'waitstatus', 'success', 'exception_occurred', 'exception_type', 'exception_severity', 'exception_note', 'result', 'expiration', 'annotations', 'dependencies'], filters=[], ids=[], user=None, max_entries=1000*)

Bases: object

User friendly class for querying lists of jobs from Flux

By default a JobList will query the last max_entries jobs for all users. Other filter parameters can be passed to the constructor or the set_user() and add_filter() methods.

Flux_handle A Flux handle obtained from flux.Flux()

Attrs Optional list of job attributes to fetch. (default is all attrs)

Filters List of strings defining the results or states to filter. E.g., ["pending", "running"].

Ids List of jobids to return. Other filters are ignored if ids is not empty.

User Username or userid for which to fetch jobs. Default is all users.

Max_entries Maximum number of jobs to return

RESULTS = {'canceled': <sphinx.ext.autodoc.importer._MockObject object>, 'completed':

STATES = {'active': <sphinx.ext.autodoc.importer._MockObject object>, 'cleanup': <sp

add_filter(*fname*)

Append a state or result filter to JobList query

fetch_jobs()

Initiate the JobList query to the Flux job-info module

JobList.fetch_jobs() returns a JobListRPC or JobListIdsFuture, either of which will be fulfilled when the job data is available.

Once the Future has been fulfilled, a list of JobInfo objects can be obtained via JobList.jobs(). If JobList.errors is non-empty, then it will contain a list of errors returned via the query.

jobs()
Synchronously fetch a list of JobInfo objects from JobList query

If the Future object returned by JobList.fetch_jobs has not yet been fulfilled (e.g. is_ready() returns False), then this call may block. Otherwise, returns a list of JobInfo objects for all jobs returned from the underlying job listing RPC.

set_user(*user*)
Only return jobs for user (may be a username or userid)

class flux.job.list.**JobListIdRPC**(**args*, ***kwargs*)
Bases: *flux.rpc.RPC*

get_job()

get_jobinfo()

class flux.job.list.**JobListIdsFuture**
Bases: *flux.future.WaitAllFuture*

Simulate interface of JobListRPC for listing multiple jobids

get_jobinfos()
get all successful results as list of JobInfo objects

Any errors are appended to self.errors.

get_jobs()
get all successful results, appending errors into self.errors

class flux.job.list.**JobListRPC**(*flux_handle*, *topic*, *payload=None*, *nodeid=<sphinx.ext.autodoc.importer._MockObject object>*, *flags=0*)
Bases: *flux.rpc.RPC*

get_jobinfos()

get_jobs()

flux.job.list.**job_list**(*flux_handle*, *max_entries=1000*, *attrs=[]*, *userid=1005*, *states=0*, *results=0*)

flux.job.list.**job_list_id**(*flux_handle*, *jobid*, *attrs=[]*)

flux.job.list.**job_list_inactive**(*flux_handle*, *since=0.0*, *max_entries=1000*, *attrs=[]*, *name=None*)

flux.job.stats module

class flux.job.stats.**JobStats**(*handle*)
Bases: object

Container for job statistics as returned by job-list.job-stats

depend
Count of jobs current in DEPEND state

priority
Count of jobs in PRIORITY state

sched
Count of jobs in SCHED state

run
Count of jobs in RUN state

cleanup
Count of jobs in CLEANUP state

inactive
Count of INACTIVE jobs

active
Total number of active jobs (all states but INACTIVE)

failed
Total number of jobs that did not exit with zero status

successful
Total number of jobs completed with zero exit code

canceled
Total number of jobs that were canceled

timeout
Total number of jobs that timed out

pending
Sum of "depend", "priority", and "sched"

running
Sum of "run" and "cleanup"

update(*callback=None*, ***kwargs*)
Asynchronously fetch job statistics and update this object.

Requires that the reactor for this handle be running in order to process the result.

Parameters

- **callback** – Optional: a callback to call when asynchronous update is complete.
- **kwargs** – Optional: extra keyword arguments to pass to callback()

update_sync()
Synchronously update job statistics

flux.job.submit module

class flux.job.submit.**SubmitFuture**(*future_handle*, *prefixes=None*, *pimpl_t=None*)
Bases: *flux.future.Future*

get_id()

flux.job.submit.**submit**(*flux_handle*, *jobspec*, *urgency=<sphinx.ext.autodoc.importer._MockObject object>*, *waitable=False*, *debug=False*, *pre_signed=False*)
Submit a job to Flux

Ask Flux to run a job, blocking until a job ID is assigned.

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobspec** (Jobspec *or its string encoding*) – jobspec defining the job request

- **urgency** (*int*) – job urgency 0 (lowest) through 31 (highest) (default is 16). Priorities 0 through 15 are restricted to the instance owner.
- **waitable** (*bool*) – allow result to be fetched with job.wait() (default is False). Waitable=true is restricted to the instance owner.
- **debug** (*bool*) – enable job manager debugging events to job eventlog (default is False)
- **pre_signed** (*bool*) – jobspec argument is already signed (default is False)

Returns job ID

Return type int

flux.job.submit.**submit_async**(*flux_handle*, *jobspec*, *urgency=<sphinx.ext.autodoc.importer._MockObject object>*, *waitable=False*, *debug=False*, *pre_signed=False*, *novalidate=False*)

Ask Flux to run a job, without waiting for a response

Submit a job to Flux. This method returns immediately with a Flux Future, which can be used obtain the job ID later.

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobspec** (Jobspec *or its string encoding*) – jobspec defining the job request
- **urgency** (*int*) – job urgency 0 (lowest) through 31 (highest) (default is 16). Priorities 0 through 15 are restricted to the instance owner.
- **waitable** (*bool*) – allow result to be fetched with job.wait() (default is False). Waitable=True is restricted to the instance owner.
- **debug** (*bool*) – enable job manager debugging events to job eventlog (default is False)
- **pre_signed** (*bool*) – jobspec argument is already signed (default is False)
- **novalidate** (*bool*) – jobspec does not need to be validated. (default is False) novalidate=True is restricted to the instance owner.

Returns a Flux Future object for obtaining the assigned jobid

Return type *Future*

flux.job.wait module

class flux.job.wait.**JobResultFuture**(*future_handle*, *prefixes=None*, *pimpl_t=None*)

Bases: *flux.future.Future*

Future fulfilled with a job "result"

Supports methods to return the result as either a raw dict or *flux.job.info.JobInfo* object.

get_dict(**args*, ***kwargs*)

get_info(**args*, ***kwargs*)

class flux.job.wait.**JobWaitFuture**(*future_handle*, *prefixes=None*, *pimpl_t=None*)

Bases: *flux.future.Future*

get_status()

class flux.job.wait.**JobWaitResult**(*jobid*, *success*, *errstr*)

Bases: tuple

errstr
Alias for field number 2

jobid
Alias for field number 0

success
Alias for field number 1

flux.job.wait.**result**(*flux_handle*, *jobid*, *flags=0*)

Wait for a job to reach its terminal state and return job result

This function waits for job completion by watching the eventlog. Because this function must process the eventlog, it is a little more heavyweight than *flux.job.wait.wait()*. However, it may be used for non-waitable jobs, jobs that have already completed, and works multiple times on the same jobid.

This function will wait until the job result is available and returns a *flux.job.info.JobInfo* object filled with the available information.

Note: The JobInfo object returned from this method is only capable of computing a small subset of job information, including, but possibly not limited to:

- id
- t_submit, t_run, t_cleanup
- returncode
- waitstatus
- runtime
- result
- result_id

Parameters

- **flux_handle** (*flux.Flux*) – handle for Flux broker
- **jobid** (*flux.job.JobID*) – the jobid for which to fetch result

Returns A limited JobInfo object which can be used to fetch the final job result, returncode, etc.

Return type JobInfo

flux.job.wait.**result_async**(*flux_handle*, *jobid*, *flags=0*)

Wait for a job to reach its terminal state and return job result

This function waits for job completion by watching the eventlog. Because this function must process the eventlog, it is a little more heavyweight than *flux.job.wait.wait_async()*. However, it may be used for non-waitable jobs, jobs that have already completed, and works multiple times on the same jobid.

Once the eventlog terminal state is reached, the returned Future is fulfilled with a set of information gleaned from the processed events, including whether the job started running (in case it was canceled before starting), any exception state, and the final exit code and wait(2) status.

Parameters

- **flux_handle** (*flux.Flux*) – handle for Flux broker
- **jobid** (*flux.job.JobID*) – the jobid for which to fetch result

Returns A Future fulfilled with the job result.

Return type *JobResultFuture*

flux.job.wait.**wait**(*flux_handle*, *jobid=<sphinx.ext.autodoc.importer._MockObject object>*)
Wait for a job to complete

Submit a request to wait for job completion, blocking until a response is received, then return the job status.

Only jobs submitted with waitable=True can be waited for.

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobid** – the job ID to wait for (default is any waitable job)

Returns job status, a tuple of: Job ID (int), success (bool), and an error (string) if success=False

Return type tuple

flux.job.wait.**wait_async**(*flux_handle*, *jobid=<sphinx.ext.autodoc.importer._MockObject object>*)
Wait for a job to complete, asynchronously

Submit a request to wait for job completion. This method returns immediately with a Flux Future, which can be used to process the result later.

Only jobs submitted with waitable=True can be waited for.

Parameters

- **flux_handle** (Flux) – handle for Flux broker from flux.Flux()
- **jobid** – the job ID to wait for (default is any waitable job)

Returns a Flux Future object for obtaining the job result

Return type *Future*

flux.resource package

Submodules

flux.resource.ResourceSet module

class flux.resource.ResourceSet.**ResourceSet**(*arg=None*, *version=1*)
Bases: object

append(**args*)
Append a ResourceSet to another

copy()
Return a copy of a ResourceSet

count(*name*)
Return a count of resource objects within a ResourceSet

Parameters **name** – The name of the object to count, e.g. "core"

diff(**args*)

dumps()
Return a short-form, human-readable string of a ResourceSet object

encode()
Encode a ResourceSet object to its serialized string representation

intersect(**args*)

ncores

ngpus

nnodes

nodelist
Return a flux.hostlist.Hostlist containing the list of hosts in this ResourceSet

ranks
Return a flux.idset.IDset containing the set of ranks in this ResourceSet

remove_ranks(*ranks*)
Remove the rank or ranks specified from the ResourceSet

Parameters **ranks** – A flux.idset.IDset object, or number or string which can be converted into an IDset, containing the ranks to remove

rlist

state
An optional state associated with this ResourceSet (e.g. "up")

union(**args*)

flux.resource.ResourceSetImplementation module

class flux.resource.ResourceSetImplementation.**ResourceSetImplementation**
Bases: abc.ABC

This abstract class defines the interface that a ResourceSet implementation shall provide in order to work with the ResourceSet class

append(*rset*)
Append one resource set to another

copy()
Return a copy of the resource set

count(*name*)
Return the total number of resources of type 'name'

diff(*rset*)
Return the set difference of two resource sets

dumps()
Return a short-form string representation of a resource set

encode()
Return a JSON string representation of the resource set

intersect(*rset*)
Return the set intersection of two resource sets

nnodes()
Return the number of nodes in the resource set as an IDset

nodelist()
Return the list of nodes in the resource set as a Hostlist

ranks()
Return the set of ranks in the resource set as an IDset

remove_ranks(*ranks*)
Remove an IDset of ranks from a resource set

union(*rset*)
Return the union of two resource sets

flux.resource.Rlist module

class flux.resource.Rlist.**Rlist**(*rstring=None*, *handle=None*)
Bases: *flux.wrapper.WrapperPimpl*

class InnerWrapper(*rstring=None*, *handle=None*)
Bases: *flux.wrapper.Wrapper*

add_child(*rank*, *name*, *ids*)

add_rank(*rank*, *hostname=None*, *cores='0'*)

append(*arg*)

copy()

count(*name*)

diff(*arg*)

dumps()

encode()

intersect(*arg*)

nnodes()

nodelist()

ranks(*hosts=None*)

remap()

remove_ranks(*ranks*)

union(*arg*)

version = 1

Submodules

flux.constants module

Used by autodoc_mock_imports.

flux.debugged module

flux.debugged.**get_mpir_being_debugged**()

flux.debugged.**set_mpir_being_debugged**(*value*)

flux.future module

class flux.future.**Future**(*future_handle*, *prefixes=None*, *pimpl_t=None*)
Bases: *flux.wrapper.WrapperPimpl*

A wrapper for interfaces that create and consume flux futures

class InnerWrapper(*handle=None*, *match=<sphinx.ext.autodoc.importer._MockObject object>*, *filter_match=True*, *prefixes=None*, *destructor=<bound method ? of <flux.core.inner.Core object>>*)
Bases: *flux.wrapper.Wrapper*

check_wrap(*fun*, *name*)

error_string()

get(**args*, ***kwargs*)

get_flux()

get_reactor()

incref()

is_ready()

reset()

then(*callback*, **args*, *timeout=-1.0*, ***kwargs*)

wait_for(**args*, ***kwargs*)

class flux.future.**WaitAllFuture**(*children=None*)
Bases: *flux.future.Future*

Create a composite future which waits for all children to be fulfilled

push(*child*, *name=None*)

flux.future.**continuation_callback**(*c_future*, *opaque_handle*)

flux.hostlist module

class flux.hostlist.**Hostlist**(*arg=''*, *handle=None*)
Bases: *flux.wrapper.WrapperPimpl*

A Flux hostlist object

The Hostlist class wraps libflux-hostlist to implement a list of hosts which can be converted to and from the RFC 29 hostlist encoding.

class InnerWrapper(*handle=None*)
Bases: *flux.wrapper.Wrapper*

append(**args*)
Append one or more arguments to a Hostlist

Args may be either a Hostlist or any valid argument to Hostlist()

copy()
Copy a Hostlist object

count()
Return the number of hosts in Hostlist

delete(*hosts*)
Delete host or hosts from Hostlist

param: hosts: A Hostlist or string in RFC 29 hostlist encoding

encode()
Encode a Hostlist to an RFC 29 hostlist string

expand()
Convert a Hostlist to a Python list

sort()
Sort a Hostlist

uniq()
Sort and remove duplicate hostnames from Hostlist

class flux.hostlist.**HostlistIterator**(*hostlist*)
Bases: object

flux.hostlist.**decode**(*arg*)
Decode a string or iterable of strings in RFC 29 hostlist format to a Hostlist object

flux.idset module

class flux.idset.**IDset**(*arg=''*, *flags=<sphinx.ext.autodoc.importer._MockObject object>*, *handle=None*)
Bases: *flux.wrapper.WrapperPimpl*

A Flux idset object

The IDset class wraps libflux-idset, and encapsulates a set of unordered non-negative integers. See idset_create(3).

A Python IDset object may be created from a valid RFC22 idset string, e.g. "0", "0-3", "0,5,7", or any Python iterable type as long as the iterable contains only non-negative integers. For example:

```
>>> ids = IDset("0-3")
>>> ids2 = IDset([0, 1, 2, 3])
>>> ids3 = IDset({0, 1, 2, 3})
```

class InnerWrapper(*arg=''*, *handle=None*)
Bases: *flux.wrapper.Wrapper*

add(*arg*)
Add all ids or values in arg to IDset :param: arg: IDset, string, or iterable of integers to add

static arg_to_set(*arg*, *method*)

clear(*start*, *end=None*)
Clear an id or range of ids in an IDset :param: start: The first id to clear :param: end: (optional) The last id in a range to clear

Returns a copy of self so that this will work: >>> print(IDset("0-9").clear(0,3))

copy()

count()
Return the number of integers in an IDset

encode(*flags=None*)
Encode an IDset to a string. :param: flags: (optional) flags to influence encoding

equal(*idset*)

expand()
Expand an IDset into a list of integers

first()
Return the first id set in an IDset

intersect(**args*)
Return the set intersection of the target IDset and all args

All args will be converted to IDsets if possible, i.e. any IDset, valid idset string, or iterable composed of integers will work.

last()
Return the greatest id set in an IDset

next(*i*)
Return the next id set in an IDset after value i

set(*start*, *end=None*)
Set an id or range of ids in an IDset :param: start: The first id to set :param: end: (optional) The last id in a range to set

Returns a copy of self so that this will work: >>> print(IDset().set(0,3))

set_flags(*flags*)
Set default flags for IDset encoding: valid flags are IDSET_FLAG_RANGE and IDSET_FLAG_BRACKETS

subtract(*arg*)
subtract all ids or values in arg from IDset :param: arg: IDset, string, or iterable of integers to subtract

test(*i*)
Test if an id is set in an IDset :param: i: the id to test

union(**args*)
Return the union of the current IDset and all args

All args will be converted to IDsets if possible, i.e. any IDset, valid idset string, or iterable composed of integers will work.

class flux.idset.**IDsetIterator**(*idset*)
Bases: object

flux.idset.**decode**(*string*)
Decode an idset string and return IDset object

flux.kvs module

class flux.kvs.**KVSDir**(*flux_handle=None*, *path='.'*, *handle=None*)

Bases: *flux.wrapper.WrapperPimpl*, collections.abc.MutableMapping

class InnerWrapper(*flux_handle=None*, *path='.'*, *handle=None*)

Bases: *flux.wrapper.Wrapper*

class KVSDirIterator(*kvsdir*)

Bases: collections.abc.Iterator

next()

commit(*flags=0*) → int

directories()

exists(*name*)

files()

fill(*contents: Mapping[str, Any]*)

Populate this directory with keys specified by contents

Parameters **contents** – A dict of keys and values to be created in the directory or None, sub-directories can be created by using dir.file syntax, sub-dicts will be stored as json values in a single key

key_at(*key*)

list_all()

mkdir(*key: str*, *contents: Mapping[str, Any] = None*)

Create a new sub-directory, optionally pre-populated with the contents of files as would be done with fill(contents)

Parameters

- **key** – Key of the directory to be created
- **contents** – A dict of keys and values to be created in the directory or None, sub-directories can be created by using *dir.file* syntax, sub-dicts will be stored as json values in a single key

class flux.kvs.**KVSWrapper**(*ffi*, *lib*, *handle=None*, *match=None*, *filter_match=True*, *prefixes=()*, *destructor=None*)

Bases: *flux.wrapper.Wrapper*

flux_kvsitr_next(**args*, ***kwargs*)

flux.kvs.**commit**(*flux_handle*, *flags: int = 0*) → int

flux.kvs.**dropcache**(*flux_handle*)

flux.kvs.**exists**(*flux_handle*, *key*)

flux.kvs.**get**(*flux_handle*, *key*)

flux.kvs.**get_dir**(*flux_handle*, *key='.'*)

flux.kvs.**get_key_direct**(*flux_handle*, *key*)

flux.kvs.**inner_walk**(*kvsdir*, *curr_dir*, *topdown=False*)

flux.kvs.**isdir**(*flux_handle*, *key*)

flux.kvs.**join**(**args*)

flux.kvs.**put**(*flux_handle*, *key*, *value*)

flux.kvs.**put_mkdir**(*flux_handle*, *key*)

flux.kvs.**put_symlink**(*flux_handle*, *key*, *target*)

flux.kvs.**put_unlink**(*flux_handle*, *key*)

flux.kvs.**walk**(*directory*, *topdown=False*, *flux_handle=None*)
Walk a directory in the style of os.walk()

flux.memoized_property module

flux.memoized_property.**memoized_property**(*fget*)
Return a property attribute for new-style classes that only calls its getter on the first access. The result is stored and on subsequent accesses is returned, preventing the need to call the getter any more.

Example

```
>>> class C(object):
...     load_name_count = 0
...     @memoized_property
...     def name(self):
...         "name's docstring"
...         self.load_name_count += 1
...         return "the name"
>>> c = C()
>>> c.load_name_count
0
>>> c.name
"the name"
>>> c.load_name_count
1
>>> c.name
"the name"
>>> c.load_name_count
1
```

flux.message module

class flux.message.**Message**(*type_id=<sphinx.ext.autodoc.importer._MockObject object>*, *handle=None*, *destruct=False*)
Bases: *flux.wrapper.WrapperPimpl*

Flux message wrapper class.

class InnerWrapper(*type_id=<sphinx.ext.autodoc.importer._MockObject object>*, *handle=None*, *destruct=False*)
Bases: *flux.wrapper.Wrapper*

classmethod from_event_encode(*topic*, *payload=None*)

payload

payload_str

topic

type

type_str

class flux.message.**MessageWatcher**(*flux_handle*, *type_mask*, *callback*, *topic_glob='*'*, *match_tag=<sphinx.ext.autodoc.importer._MockObject object>*, *args=None*)

Bases: flux.core.watchers.Watcher

destroy()

start()

stop()

flux.message.**msg_typestr**(*msg_type*)

flux.progress module

class flux.progress.**Bottombar**(*formatter=None*, ***kwargs*)

Bases: object

Maintain a status line at bottom of terminal using vt100 escape codes

The Bottombar class implements a very simple status line which stays positioned at the last line of vt100 capable terminals through the use of vt100 escape codes.

This class will only work propertly on vt100 compatible terminals, which includes xterm, rxvt, and gnome-terminal and their derivatives on Linux, as well as iTerm and Terminal on OSX, and reportedly the new Windows Terminal on Windows.

Use of Bottombar requires that a `formatter` function be provided. The `formatter` will be called on each update as:

```
formatter(bbar, width)
```

Where `bbar` is the bottombar object being formatted and `width` is the current terminal width at the time of the update. The default `formatter` will simply print all extra

As a convenience, the Bottombar constructor collects all extra keyword arguments and presents them as attributes on the Bottombar object for later access from within and outside the provided `formatter`, e.g:

```
def formatter(bb, width):
    text = f"iteration={bb.i}"
    return text + time.ctime().rjust(width - len(text))

bb = Bottombar(formatter, i=0).start()
for i in range(0, 128):
    bb.update(i=i)
    time.sleep(.05)
bb.stop()
```

will print a statusbar with an iteration count left justified, and the current time right justified.

elapsed

The elapsed time since `bb.start()` in floating point seconds. As a convenience, `bb.elapsed` may be converted to a `datetime.timedelta` object via the `dt` attribute, e.g. `bb.elapsed.dt`.

Type float

Parameters

- **formatter** (*function*) – Function which returns the status string
- **kwargs** – all extra keyword arguments are collected in the Bottombar instance and made available as attributes for convenience

redraw()
Redraw bar without update

start()
Start drawing a Bottombar

stop()
Reset terminal and write final bottombar state with newline

update(***kwargs*)
Update keyword args and redraw a bottombar

class flux.progress.**ElapsedTime**
Bases: float

An ElapsedTime object is a floating point elapsed time in seconds that comes with a convenient "dt" proerty that returns a datetime.timedelta object

dt

class flux.progress.**ProgressBar**(*total=100*, *style='vertbars'*, *before=''*, *after=' {percent:5.1f}%'*, *autostop=False*, ***kwargs*)
Bases: *flux.progress.Bottombar*

Simple progress bar that stays on last line of terminal

The ProgressBar class uses the features of Bottombar to create a progress bar, plus optional other text, which stays on the last line of a terminal. A vt100 compatible terminal is required.

Parameters

- **total** (*int*) – The total expected number of items/units for which the progressbar is monitoring progress, default=100.
- **style** (*str, optional*) – A string progress bar style from the list "line", "bar", "dots", "steps", "vertbars".
- **before** (*str, optional*) – A string to place before the progress bar.
- **after** (*str, optional*) – A string to place after the progress bar. default=" {percent:5.1f}%"
- **autostop** (*bool, optional*) – If True, ProgressBar instance will be automatically stopped when count == total. Otherwise, terminal reset will be deferred to an atexit handler.
- **kwargs** (*optional*) – Extra keyword args are saved and passed as args when formatting the `before` and `after` strings.

The `before` and `after` strings are formatted on each update to the progressbar and passed all extra keyword args, plus the current `total`, `count`, `percent`, and `elapsed` time e.g.:

```
before_str = before.format(
                total=total,
                count=count,
                percent=percent,
                elapsed=elapsed,
                **kwargs
             )
```

which means that these strings are most useful when they are format strings, e.g.:

```
ProgressBar(before="Running {total} jobs, {percent}% complete: ")
```

bar_style = {'bar': '—', 'dots': '', 'line': '—', 'steps': ' ', 'vertbars': '

update(*advance=1*, ***kwargs*)

Update the state of a ProgressBar

Update the state of a ProgressBar and redraw if the progress bar is currently running. If `count == total` and autostop is set, the progress bar will be automatically stopped.

When the progress bar is stopped, the terminal will be reset and the final state of the bar will be left on the last line of output.

Parameters

- **advance** (*int, optional*) – Advance progress by `advance` amount.
- **kwargs** – Update stored keyword arguments

flux.rpc module

class flux.rpc.**RPC**(*flux_handle*, *topic*, *payload=None*, *nodeid=<sphinx.ext.autodoc.importer._MockObject object>*, *flags=0*)

Bases: *flux.future.Future*

An RPC state object

class RPCInnerWrapper(*handle=None*, *match=<sphinx.ext.autodoc.importer._MockObject object>*, *filter_match=True*, *prefixes=None*, *destructor=<bound method Wrapper.__getattr__.<locals>.wrap_class of <flux.core.inner.Core object>>*)

Bases: *flux.wrapper.Wrapper*

check_wrap(*fun*, *name*)

get()

get_str(**args*, ***kwargs*)

flux.security module

class flux.security.**SecurityContext**(*config_pattern=None*, *flags=0*)

Bases: *flux.wrapper.WrapperPimpl*

A Flux Security Context object

class InnerWrapper(*flags=0*)

Bases: *flux.wrapper.Wrapper*

check_wrap(*fun*, *name*)

sign_unwrap(*signed_payload*, *flags=0*)

sign_wrap(*payload*, *mech_type=<sphinx.ext.autodoc.importer._MockObject object>*, *flags=0*)

sign_wrap_as(*userid*, *payload*, *mech_type=<sphinx.ext.autodoc.importer._MockObject object>*, *flags=0*)

class flux.security.**SecurityFunctionWrapper**(*fun*, *name*, *function_type*, *ffi*, *add_handle=False*)
Bases: *flux.wrapper.FunctionWrapper*

flux.util module

flux.util.**check_future_error**(*func*)

flux.util.**encode_payload**(*payload*)

flux.util.**encode_topic**(*topic*)

class flux.util.**CLIMain**(*logger=None*)
Bases: object

flux.util.**parse_fsd**(*fsd_string*)

flux.wrapper module

Flux interface wrapper generator. This could, in principle, be used for other projects as well, but it encodes a number of assumptions about the error propagation and handling that flux uses.

class flux.wrapper.**ErrorPrinter**(*name*, *prefixes*)
Bases: object

class flux.wrapper.**FunctionWrapper**(*fun*, *name*, *function_type*, *ffi*, *add_handle=False*)
Bases: object

build_argument_translation_list(*fun_type*)

set_error_check(*fun*)

exception flux.wrapper.**InvalidArguments**(*name*, *signature*, *arguments*)
Bases: ValueError

exception flux.wrapper.**MissingFunctionError**(*name*, *c_name*, *name_list*, *arguments*)
Bases: Exception

class flux.wrapper.**Wrapper**(*ffi*, *lib*, *handle=None*, *match=None*, *filter_match=True*, *prefixes=()*, *destructor=None*)
Bases: *flux.wrapper.WrapperBase*

Forms a wrapper around an interface that dynamically searches for undefined names, and can detect and pass a handle argument of specified type when it is found in the signature of an un-specified target function.

check_handle(*name*, *fun_type*)

check_wrap(*fun*, *name*)

handle

class flux.wrapper.**WrapperBase**
Bases: object

handle

class flux.wrapper.**WrapperPimpl**
Bases: *flux.wrapper.WrapperBase*

handle

exception flux.wrapper.**WrongNumArguments**(*name*, *signature*, *ftype*, *arguments*, *htype*)
Bases: ValueError

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

D

E

F

G

H

I

J

K

L

M

N

T

U

V

W