SystemTap Language Reference

19th October 2006

Contents

1 SystemTap Overview

1.1 Why use SystemTap?

The goal of SystemTap is to provide infrastructure to simplify the gathering of information about the running Linux kernel so that it can be further analyzed. This can assist in identifying the underlying cause of a performance or functional problem. SystemTap is designed to eliminate the need for the developer to go through the tedious instrument, recompile, install, and reboot sequence required to collect data on the operation of the kernel. The recent addition of Kprobes to the Linux kernel provides the needed support but does not provide an easy to use infrastructure. SystemTap provides a simple command line interface and scripting language for writing kernel instrumentation.

SystemTap is under rapid development and is still evolving. Do not use it on production systems. Expect that things will change. However, SystemTap in its current state can still be a useful tool for developers.

As SystemTap evolves, we plan to add "tapsets" to aid in collecting specific types of data. We will also eventually support probing userpace applications. We are looking into integrating Systemtap with similar tools such as Frysk, Oprofile and LTT.

Current project members include Red Hat, IBM, Intel, and Hitachi.

1.2 Event/action language

SystemTap's language is strictly typed, declaration free, procedural, and inspired by dtrace and awk. It allows source code points or events in the kernel to be associated with handlers, which are subroutines that are executed synchronously. It is somewhat similar conceptually to "breakpoint command lists" in the gdb debugger.

There are two main outermost constructs: probes and functions. Within these, statements and expressions use C-like operator syntax and precedence.

1.3 Sample SystemTap scripts

1.4 The *stap* command

The *stap* program is the frontend to the Systemtap tool. It accepts probing instructions, translates those instructions into C code, compiles this C code, and loads the resulting kernel module into a running Linux kernel to perform the requested system trace/probe functions. You can supply the script in a named file, from standard input, or from the command line. The program runs until it is interrupted by the user, or if the script voluntarily invokes the *exit()* function, or by a sufficient number of soft errors.

Stap

- Translates the script

- Generates and compiles a kernel module
- Inserts the module – output to *stap*'s stdout
- CTRL-C unloads the module and terminates *stap*
- (This section should NOT try to explain all *stap* options.)

2 Types of SystemTap Scripts

2.1 Probe scripts

Analogous to programs; identify probe point and associated handlers.

2.2 Tapset scripts

Libraries of probe aliases and auxiliary functions.

/usr/share/systemtap/tapset contains tapset scripts

3 Components of a SystemTap Script

The main construct in the scripting language identifies probes. Probes associate abstract events with a statement block ("probe handler") that is to be executed when any of those events occur.

3.1 Probe definitions

The general syntax is as follows:

```
probe PROBEPOINT [, PROBEPOINT] { [STMT ...] }
```

Events are specified in a special syntax called "probe points". There are several varieties of probe points defined by the translator, and tapset scripts may define further ones using aliases. These are listed in the *stapprobes*(5) manual pages.

The probe handler is interpreted relative to the context of each event. For events associated with kernel code, this context may include variables defined in the source code at that spot. These "target variables" are presented to the script as variables whose names are prefixed with "`$`". They may be accessed only if the kernel's compiler preserved them despite optimization. This is the same constraint that a debugger user faces when working with optimized code. Some other events have very little context.

3.2 Probe aliases

```
probe <alias> = <probepoint> { <prologue_stmts> }
probe <alias> += <probepoint> { <epilogue_stmts> }
```

New probe points may be defined using "aliases". Probe point aliases look similar to probe definitions, but instead of activating a probe at the given point, it just defines a new probe point name as an alias to an existing one. There are two types of alias, the prologue style and the epilogue style which are identified by "=" and "+=" respectively.

Another probe that names the new probe point will create an actual probe, with the handler of the alias *prepended*.

This prepending behavior serves several purposes. It allows the alias definition to "preprocess" the context of the probe before passing control to the user-specified handler. This has several possible uses:

Skip probe unless given condition is met:

```
if ($flag1 != $flag2) next
```

Supply probe-describing values:

```
name = "foo"
```

Extract target variable to plain local variable:

```
var = $var
```

3.2.1 Prologue-style aliases (=)

The statement block that follows an alias definition is implicitly added as a prologue to any probe that refers to the alias. For example,

```
probe syscall.read = kernel.function("sys_read") {
   fildes = $fd
}
```

defines a new probe point *syscall.read*, which expands to *kernel.function("sys_read")*, with the given statement as a prologue.

3.2.2 Epilogue-style aliases (+=)

The statement block that follows an alias definition is implicitly added as an epilogue to any probe that refers to the alias. For example,

```
probe syscall.read += kernel.function("sys_read") {
   fildes = $fd
}
```

defines a new probe point with the given statement as an epilogue.

3.2.3 Probe alias usage

Another probe definition may use the alias like this:

```
probe syscall.read {
   printf("reading fd=%d0, fildes)
}
```

3.2.4 Unused alias variables

Variables initialized by `prologue_stmts` but not subsequently used by the handler are discarded? Ignored?

3.3 Variables

Identifiers for variables and functions are an alphanumeric sequence, and may include "_" and "$" characters. They may not start with a plain digit, as in C. Each variable is by default local to the probe or function statement block within which it is mentioned, and therefore its scope and lifetime is limited to a particular probe or function invocation. Scalar variables are implicitly typed as either string or integer. Associative arrays also have a string or integer value, and a a tuple of strings and/or integers serving as a key. The translator performs *type inference* on all identifiers, including array indexes and function parameters. Inconsistent type-related use of identifiers signals an error.

Variables may be declared global, so that they are shared amongst all probes and live as long as the entire SystemTap session. There is one namespace for all global variables, regardless of which script file they are found within. A global declaration may be written at the outermost level anywhere, not just within a block of code. The following declaration marks "var1" and "var2" as global. The translator will infer for each its value type, and if it is used as an array, its key types.

```
global var1[=<value>], var2[=<value>]
```

3.4 Auxiliary functions

```
function <name>[:<type>] ( <arg1>[:<type>], ... ) { <stmts> }
```

SystemTap scripts may define subroutines to factor out common work. Functions take any number of scalar (integer or string) arguments, and must return a single scalar (integer or string). An example function declaration looks like this:

```
function thisfn (arg1, arg2) {
   return arg1 + arg2
}
```

Note the general absence of type declarations, which are instead inferred by the translator. However, if desired, a function definition may include explicit type declarations for its return value and/or its arguments. This is especially helpful for embedded-C functions. In the following example, the type inference engine need only infer type type of arg2 (a string).

```
function thatfn:string (arg1:long, arg2) {
   return sprint(arg1) . arg2
}
```

Functions may call others or themselves recursively, up to a fixed nesting limit. This limit is defined by a macro in the translated C code and is in the neighborhood of 10.

3.5 Embedded C

When in *guru mode*, the translator accepts embedded code in the script. Such code is enclosed between `%{` and `%}` markers, and is transcribed verbatim, without analysis, in some sequence, into the generated C code. At the outermost level, this may be useful to add `#include` instructions, and any auxiliary definitions for use by other embedded code.

3.6 Embedded C functions

```
function <name>:<type> ( <arg1>:<type>, ... ) %{ <C_stmts> %}
```

Embedded code is permitted in a function body. In this case, the script language body is replaced entirely by a piece of C code enclosed again between `%{` and `%}` markers. This C code may do anything reasonable and safe. There are a number of undocumented but complex safety constraints on concurrency, resource consumption, and runtime limits, so this is an advanced technique.

The memory locations set aside for input and output values are made available to it using a macro named `THIS`. Here are some examples:

```
function add_one (val) %{
   THIS->__retvalue = THIS->val + 1;
%}
function add_one_str (val) %{
   strlcpy (THIS->__retvalue, THIS->val, MAXSTRINGLEN);
   strlcat (THIS->__retvalue, "one", MAXSTRINGLEN);
%}
```

The function argument and return value types have to be inferred by the translator from the call sites in order for this to work. The user should examine C code generated for ordinary script-language functions in order to write compatible embedded-C ones.

4 Probe Points

4.1 General syntax

4.1.1 Prefixes: kernel, timer, etc.

4.1.2 Suffixes: .entry, .return

4.1.3 Wild-carded file names, function names

4.2 Built-in probe point types

4.2.1 `kernel.function("func[@file]")`

4.2.2 `kernel.inline("func[@file]")`

4.2.3 `kernel.statement("func@file:linenumber")`

4.2.4 `kernel.statement(<hexaddr>)`

4.2.5 `module("modname") -- kernel module`

4.2.6 Timer probes

e.g., `timer.sec(5)`

4.2.7 Other units

s/sec, ms/msec, us/usec, ns/nsec, jiffies, hz

4.2.8 randomize clause

[What's the syntax for this?]

4.2.9 Return probes

`$return = return value`

4.3 Probe point aliases defined by tapsets

4.3.1 `kernel.syscall.open`

4.4 Special probe points

The probe points begin and end are defined by the translator to refer to the time of session startup and shutdown. All "begin" probe handlers are run, in some sequence, during the startup of the session. All global variables will have been initialized prior to this point. All "end" probes are run, in some sequence, during the normal shutdown of a session, such as in the aftermath of an exit() function call, or an interruption from the user. In the case of an error-triggered shutdown, "end" probes are not run. There are no target variables available in either context.

4.4.1 begin

4.4.2 end

5 Language Elements

5.1 Identifiers

Identifiers for variables and functions are an alphanumeric sequence, and may include "_" and "$" characters. They may not start with a plain digit, as in C.

Significance of "$": $varname = the value of variable varname in the context of the probed function

5.2 Data types

5.2.1 Variables' types are generally inferred from use.

5.2.2 Function arg types and return values can be explicitly specified.

5.2.3 integers, numeric literals

5.2.4 strings, string literals

5.2.5 associative arrays

(ref later section)

5.2.6 statistics

(ref later section)

5.3 Semicolons

Optional statement delimiters

5.4 Comments

Three forms of comments are supported:

```
# ... shell style, to the end of line
// ... C++ style, to the end of line
/* ... C style ... */
```

5.5 Whitespace

Whitespace is ignored.

5.6 Expressions

Systemtap supports a number of operators that have the same general syntax, semantics, and precedence as in C and awk. Arithmetic is performed as per typical C rules for signed integers. Division by zero or overflow is detected and results in an error.

5.6.1 binary string operators

`* / % + - >> << & ^ | && ||`
  `.` (string concatenation)

5.6.2 numeric assignment operators

`= *= /= %= += -= >>= <<= &= ^= |=`

5.6.3 string assignment operators

`= .=`

5.6.4 unary numeric operators

`+ - ! ~ ++ --`

5.6.5 binary numeric or string comparison operators

`< > <= >= == !=`

5.6.6 ternary operator

`cond ? exp1 : exp2`

5.6.7 grouping operator

`( exp )`

5.6.8 function call

`fn ([ arg1, arg2, ... ])`

5.6.9 `$ptr->member`

Where ptr is a kernel pointer available in the probed context.

5.6.10 `<value> in <array_name>`

Evaluates to true if array contains an element with the specified index.

5.6.11 [<value>, ...] in <array_name>

5.7 Literals passed in from the *stap* command line

Literals are either strings enclosed in double-quotes (passing through the usual C escape codes with backslashes), or integers (in decimal, hexadecimal, or octal, using the same notation as in C). All strings are limited in length to some reasonable value (a few hundred bytes). Integers are 64-bit signed quantities, although the parser also accepts (and wraps around) values above positive 2**63.

Script arguments given at the end of the command line may be expanded as literals. These may be used in all contexts where literals are accepted. Reference to an argument number beyond what was actually given is an error.

5.7.1 `$1 ... $<NN>` for integers

Use \$1 ... \$<NN> for casting as a numeric literal.

5.7.2 `@1 ... @<NN>` for strings

Use `@1 ... @<NN>` for casting as string literal.

5.8 Conditional compilation

5.8.1 Conditions

A simple conditional preprocessing stage is run as a part of parsing. The general form is similar to the `cond ? exp1 : exp2` ternary operator:

```
%( CONDITION %? TRUE-TOKENS %)
%( CONDITION %? TRUE-TOKENS %: FALSE-TOKENS %)
```

The `CONDITION` is a very limited expression whose format is determined by its first keyword.

```
%( <condition> %? <code> [ %: <code> ] %)
```

5.8.2 Conditions based on kernel version: `kernel_v`, `kernel_vr`

If the first part is the identifier `kernel_vr` or `kernel_v` to refer to the kernel version number, with (`"2.6.13-1.322FC3smp"`) or without (`"2.6.13"`) the release code suffix, then the second part is one of the six standard numeric comparison operators `<`, `<=`, `==`, `!=`, `>`, and `>=`, and the third part is a string literal that contains an RPM-style version-release value. The condition is deemed satisfied if the version of the target kernel (as optionally overridden by the `-r` option) compares to the given version string. The comparison is performed by the glibc function `strverscmp`.

5.8.3 Conditions based on architecture: arch

If the first part is the identifier `arch` to refer to the processor architecture, then the second part then the second part is one of the two string comparison operators `==` or `!=`, and the third part is a string literal for matching it. This comparison is simple string (in)equality.

5.8.4 True/False Tokens

The TRUE-TOKENS and FALSE-TOKENS are zero or more general parser tokens (possibly including nested preprocessor conditionals), and are pasted into the input stream if the condition is true or false. For example, the following code induces a parse error unless the target kernel version is newer than 2.6.5:

```
%( kernel_v <= "2.6.5" %? **ERROR** %) # invalid token sequence
```

The following code might adapt to hypothetical kernel version drift:

```
probe kernel.function (
   %( kernel_v <= "2.6.12" %? "__mm_do_fault" %:
      %( kernel_vr == "2.6.13-1.8273FC3smp" %? "do_page_fault" %:
         UNSUPPORTED %) %)
   ) { /* ... */ }
%( arch == "ia64" %?
   probe syscall.vliw = kernel.function("vliw_widget") {}
%)
```

6 Statement Types

Statements enable procedural control flow. They may occur within functions and probe handlers. The total number of statements executed in response to any single probe event is limited to some number defined by a macro in the translated C code, and is in the neighborhood of 1000.

6.0.5 break, continue

Exit or iterate the innermost nesting loop (within `while` or `for` or `foreach`) statement. Same syntax and semantics as in C.

6.0.6 - delete <name>

Ref "Associative Arrays" section

6.0.7 `delete`

```
delete ARRAY[INDEX1, INDEX2, ...]
```

Remove from `ARRAY` the element specified by the index tuple. The value will no longer be available, and subsequent iterations will not report the element. It is not an error to delete an element that does not exist.

```
delete ARRAY
```

Remove all elements from `ARRAY`.

```
delete SCALAR
```

Removes the value of `SCALAR`. Integers and strings are cleared to 0 and `""` respectively, while statistics are reset to the initial empty state.

6.0.8 `do`

```
do STMT while (EXP)
```

Same syntax and semantics as in C.

6.0.9 `EXP` (expression)

Execute the string- or integer-valued expression and throw away the value.

6.0.10 `for`

```
for (EXP1; EXP2; EXP3) STMT
```

Execute `EXP1` as initialization. While `EXP2` is non-zero, execute `STMT`, then the iteration expression `EXP3`. Same syntax and semantics as in C.

6.0.11 `foreach`

```
foreach (VAR in ARRAY) STMT
```

Loop over each element of the named global array, assigning current key to `VAR`. The array may not be modified within the statement. By adding a single `+` or `-` operator after the `VAR` or the `ARRAY` identifier, the iteration will proceed in a sorted order, by ascending or descending index or value.

```
foreach ([VAR1, VAR2, ...] in ARRAY) STMT
```

Same as above, used when the array is indexed with a tuple of keys. A sorting suffix may be used on at most one `VAR` or `ARRAY` identifier.

6.0.12 Use of + or - suffix for ascending vs. descending order

Ref "Associative Arrays" section, `foreach`

6.0.13 `if`

```
if (EXP) STMT1 [ else STMT2 ]
```

Compare integer-valued `EXP` to zero. Execute the first (non-zero) or second `STMT` (zero).

Same syntax and semantics as in C.

6.0.14 `next`

Return now from enclosing probe handler.

6.0.15 ; (null statement)

```
statement1
;
statement2
```

Null statement, do nothing. It is useful as an optional separator between statements to improve syntax error detection and to handle certain grammar ambiguities.

6.0.16 `return`

```
return EXP
```

Return `EXP` value from enclosing function. If the function's value is not taken anywhere, then a return statement is not needed, and the function will have a special "unknown" type with no return value.

6.0.17 { } (statement block with zero or more statements)

```
{ STMT1 STMT2 ... }
```

Execute each statement in sequence in this block. Note that separators or terminators are generally not necessary between statements. Same syntax and semantics as in C.

6.0.18 `while`

`while (EXP) STMT`

While integer-valued `EXP` evaluates to non-zero, execute `STMT`. Same syntax and semantics as in C.

7 Associative Arrays

7.1 Examples

7.2 Types of values

7.3 Number and types of indexes

7.4 Array capacity

Adjust with "stap -DMAXMAPENTRIES=<n>"

7.5 Iteration

7.5.1 foreach

8 Statistics

(AKA Aggregates)

8.1 <<< operator

8.2 Integer extractors

8.2.1 @count(s)

8.2.2 @sum(s)

8.2.3 @min(s)

8.2.4 @max(s)

8.2.5 @avg(s)

8.3 Histogram extractors

8.3.1 @hist_linear

8.3.2 @hist_log

9 Predefined Functions

Note that these functions are implemented in tapsets (with support in the runtime, in some cases) rather than being built into the translator, so "predefined" is a relative term.

9.1 Logging functions

9.1.1 error

```
error:unknown (msg:string)
```

An error has occurred. Log the given string to the error stream. Append an implicit end-of-line. *stpd* prepends the string "ERROR:". Block any further execution of statements in this probe. If the number of errors so far exceeds the MAXERRORS parameter, also trigger an *exit*().

9.1.2 `log`

```
log:unknown (msg:string)
```

Log the given string to the common trace buffer. Append an implicit end-of-line.

9.1.3 `print`

```
print:unknown ()
```

Print the given integer, string, or statistics value to the common trace buffer.

9.1.4 `printf`

```
printf:unknown (fmt:string, )
```

Like the C printf, except valid types are limited to string ("%s") and integer ("%d").

9.1.5 `sprint`

Operates like print, but returns the formatted string instead of logging it.

9.1.6 `sprintf`

Operates like printf, but like sprint, returns the formatted string instead of logging it.

9.1.7 `system`

```
system (cmd:string)
```

Runs a command on the system. The command will run in the background when the current probe completes.

9.1.8 `warn`

```
warn:unknown (msg:string)
```

Log the given string to the warning stream. Append an implicit end-of-line. stpd prepends the string "WARNING:".

9.2 Task context at probepoint

9.2.1 backtrace

```
backtrace:string ()
```

Return a string of hex addresses that are a backtrace of the stack. It may be truncated due to maximum string length.

9.2.2 cpu

```
cpu:long ()
```

Return the current cpu number.

9.2.3 egid

```
egid:long ()
```

Return the effective gid of the current process.

9.2.4 euid

```
euid:long ()
```

Return the effective uid of the current process.

9.2.5 execname

```
execname:string ()
```

Return the name of the current process.

9.2.6 gid

```
gid:long ()
```

Return the gid of the current process.

9.2.7 is_return

```
is_return:long ()
```

Return 1 if the probe point is a return probe. DEPRECATED.

9.2.8 pexecname

```
pexecname:string()
```

Return the name of the parent process.

9.2.9 `pid`

```
pid:long ()
```

Return the id of the current process.

9.2.10 `ppid`

```
ppid:long ()
```

Return the id of the parent process.

9.2.11 `print_backtrace`

```
print_backtrace:unknown ()
```

Equivalent to `print_stack(backtrace())`, except that deeper stack nesting may be supported. Return nothing.

9.2.12 `print_regs`

```
print_regs:unknown ()
```

Print a register dump.

9.2.13 `print_stack`

```
print_stack:unknown (bt:string)
```

Perform a symbolic lookup of the addresses in the given string, which is assumed to be the result of a prior call to `backtrace()`. Print one line per address, including the address, the name of the function containing the address, and an estimate of its position within that function. Return nothing.

9.2.14 `target`

```
target:long ()
```

Return the pid of the target process.

9.2.15 `tid`

```
tid:long ()
```

Return the id of the current thread.

9.2.16 `uid`

```
uid:long ()
```

Return the uid of the current process.
(See context.stp, task.stp)

9.3 Accessing string data at probepoint

9.3.1 `kernel_string`

    kernel_string:string (addr:long)

Copy a string from kernel space at given address. The validation of this address is only partial at present.

9.3.2 `user_string`

    user_string:string (addr:long)

Copy a string from user space at given address. The validation of this address is only partial at present.

9.3.3 etc. (See conversions.stp)

9.4 Initializing Queue Statistics

The `queue_stats` tapset provides functions that, given notifications of elementary queuing events (wait, run, done), tracks averages such as queue length, service and wait times, and utilization. The following three functions should be called from appropriate probes, in sequence:

9.4.1 `qs_wait`

    qs_wait:unknown (qname:string)

Record that a new request was enqueued for the given queue name.

9.4.2 `qs_run`

    qs_run:unknown (qname:string)

Record that a previously enqueued request was removed from the given wait queue and is now being serviced.

9.4.3 `qs_done`

    qs_done:unknown (qname:string)

Record that a request originally from the given queue has completed being serviced.

9.5 Using Queue Statistics

Functions with the prefix `qsq_` are for querying the statistics averaged since the first queue operation (or when `qsq_start` was called). Since statistics are often fractional, a scale parameter is multiplies the result to a more useful scale. For some fractions, a scale of 100 will usefully return percentage numbers.

9.5.1 qsq_start

```
qsq_start:unknown (qname:string)
```

Reset the statistics counters for the given queue, and start tracking anew from this moment.

9.5.2 qsq_print

```
qsq_print:unknown (qname:string)
```

Print a line containing a selection of the given queue's statistics.

9.5.3 qsq_utilization

```
qsq_utilization:long (qname:string, scale:long)
```

Return the fraction of elapsed time when the resource was utilized.

9.5.4 qsq_blocked

```
qsq_blocked:long (qname:string, scale:long)
```

Return the fraction of elapsed time when the wait queue was used.

9.5.5 qsq_wait_queue_length

```
qsq_wait_queue_length:long (qname:string, scale:long)
```

Return the average length of the wait queue.

9.5.6 qsq_service_time

```
qsq_service_time:long (qname:string, scale:long)
```

Return the average time required to service a request.

9.5.7 qsq_wait_time

```
qsq_wait_time:long (qname:string, scale:long)
```

Return the average time a request took from being enqueued to completed.

9.5.8 qsq_throughput

```
qsq_throughput:long (qname:string, scale:long)
```

Return the average rate of requests per scale units of time.

9.6 Probepoint ID

9.6.1 pp

```
pp:string ()
```

Return the probe point associated with the currently running probe handler, including alias and wildcard expansion effects.

9.6.2 probefunc

```
probefunc:string ()
```

Return the probe point's function name, if known.

9.7 Formatting Functions

9.7.1 ctime

```
ctime:string (seconds:long)
```

Return a simple textual rendering (e.g., `"Wed Jun 30 21:49:008 1993"`) of the given number of seconds since the epoch, as perhaps returned by `gettimeofday_s()`.

9.7.2 errno_str

```
errno_str:string (e:long)
```

Return the symbolic string associated with the given error code, like `"ENOENT"` for the number 2, or `"E#3333"` for an out-of-range value like 3333.

9.7.3 thread_indent

```
thread_indent:string (delta:long)
```

Return a string with an appropriate indentation for this thread. Call it with a small positive or matching negative delta. If this is the outermost, initial level of indentation, reset the relative timestamp base to zero.

9.7.4 thread_timestamp

```
thread_timestamp:long ()
```

Return an absolute timestamp value for use by the indentation function. The default function uses `gettimeofday_us`

9.7.5 `returnstr`

9.8 String Functions

9.8.1 `isinstr`

```
isinstr:long (s1:string, s2:string)
```

Return 1 if string s1 contains string s2, returns 0 otherwise.

9.8.2 `strlen`

```
strlen:long (str:string)
```

Return the number of characters in str.

9.8.3 `substr`

```
substr:string (str:string,start:long, stop:long)
```

Return the substring of str starting from character start and ending at character stop.

9.8.4 etc.

(See string.stp)

9.9 Timestamps

9.9.1 `get_cycles`

```
get_cycles:long ()
```

Return the processor cycle counter value, or 0 if unavailable.

9.9.2 `gettimeofday_ms`

```
gettimeofday_ms:long ()
```

Return the number of milliseconds since the UNIX epoch.

9.9.3 `gettimeofday_s`

```
gettimeofday_s:long ()
```

Return the number of seconds since the UNIX epoch.

9.9.4 `gettimeofday_us`

```
gettimeofday_us:long ()
```

Return the number of microseconds since the UNIX epoch.
(See timestamp.stp)

9.10 Other tapset functions

9.10.1 addr_to_node

```
addr_to_node:long (addr:long)
```

Return which node the given address belongs to in a NUMA system.

9.10.2 exit

```
exit:unknown ()
```

Enqueue a request to shut down the systemtap session. This does not unwind the current probe handler, nor block new probe handlers. stpd will shortly respond to the request and initiate an orderly shutdown.

(See also logging.stp)

9.10.3 system

```
system (cmd:string)
```

Runs a command on the system. The command will run in the background when the current probe completes.

10 For Further Reference

Refs to *stap*(1) man page, tutorial, tapsets, Wiki

Index