SystemTap Language Reference

February 22, 2007

Contents

1 SystemTap overview **7**
- 1.1 About this guide . . . 7
- 1.2 Reasons to use SystemTap . . . 7
- 1.3 Event-action language . . . 7
- 1.4 Sample SystemTap scripts . . . 7
  - 1.4.1 Basic SystemTap syntax and control structures . . . 7
  - 1.4.2 Primes between 0 and 49 . . . 8
  - 1.4.3 Recursive functions . . . 9
- 1.5 The stap command . . . 9
- 1.6 Safety and security . . . 10

2 Types of SystemTap scripts **10**
- 2.1 Probe scripts . . . 10
- 2.2 Tapset scripts . . . 10

3 Components of a SystemTap script **11**
- 3.1 Probe definitions . . . 11
- 3.2 Probe aliases . . . 11
  - 3.2.1 Prologue-style aliases (=) . . . 12
  - 3.2.2 Epilogue-style aliases (+=) . . . 12
  - 3.2.3 Probe alias usage . . . 12
  - 3.2.4 Unused alias variables . . . 12
- 3.3 Variables . . . 12
- 3.4 Auxiliary functions . . . 13
- 3.5 Embedded C . . . 13
- 3.6 Embedded C functions . . . 14

4 Probe points **14**
- 4.1 General syntax . . . 14
  - 4.1.1 Prefixes: kernel, timer, and so on . . . 14
  - 4.1.2 Suffixes: .entry, .return . . . 15
  - 4.1.3 Wildcarded file names, function names . . . 15
  - 4.1.4 Optional probe points . . . 15
- 4.2 Built-in probe point types (DWARF probes) . . . 15
  - 4.2.1 kernel.function . . . 16
  - 4.2.2 kernel.inline . . . 16
  - 4.2.3 kernel.statement . . . 16
- 4.3 Marker probes . . . 17
- 4.4 Timer probes . . . 17
- 4.5 Return probes . . . 18
- 4.6 Special probe points . . . 18
  - 4.6.1 begin . . . 18
  - 4.6.2 end . . . 18
  - 4.6.3 Begin and end probe sequence . . . 18
  - 4.6.4 never . . . 18
- 4.7 Probes to monitor performance . . . 19
  - 4.7.1 $counter . . . 19

4.7.2 read_counter . . . 19

5 Language elements **19**
5.1 Identifiers . . . 19
5.2 Data types . . . 19
5.2.1 Numbers . . . 19
5.2.2 Literals . . . 19
5.2.3 Integers . . . 20
5.2.4 Strings . . . 20
5.2.5 Associative arrays . . . 20
5.2.6 Statistics . . . 20
5.3 Semicolons . . . 20
5.4 Comments . . . 20
5.5 Whitespace . . . 20
5.6 Expressions . . . 20
5.6.1 Binary string operators . . . 20
5.6.2 Numeric assignment operators . . . 20
5.6.3 String assignment operators . . . 21
5.6.4 Unary numeric operators . . . 21
5.6.5 Binary numeric or string comparison operators . . . 21
5.6.6 Ternary operator . . . 21
5.6.7 Grouping operator . . . 21
5.6.8 Function call . . . 21
5.6.9 $ptr->member . . . 21
5.6.10 <value> in <array_name> . . . 21
5.6.11 [<value>, ...] in <array_name> . . . 21
5.7 Literals passed in from the stap command line . . . 21
5.7.1 $1 ... $<NN> for integers . . . 21
5.7.2 @1 ... @<NN> for strings . . . 21
5.8 Conditional compilation . . . 22
5.8.1 Conditions . . . 22
5.8.2 Conditions based on kernel version: kernel_v, kernel_vr . . . 22
5.8.3 Conditions based on architecture: arch . . . 22
5.8.4 True and False Tokens . . . 22

6 Statement types **23**
6.0.5 break and continue . . . 23
6.0.6 - (delete) <name> . . . 23
6.0.7 delete . . . 23
6.0.8 do . . . 23
6.0.9 EXP (expression) . . . 23
6.0.10 for . . . 23
6.0.11 foreach . . . 24
6.0.12 Use of plus (+) or minus (-) suffix for ascending or descending order . . . 24
6.0.13 goto . . . 24
6.0.14 if . . . 24
6.0.15 next . . . 24
6.0.16 ; (null statement) . . . 24

6.0.17 return . . . 25
6.0.18 switch . . . 25
6.0.19 { } (statement block) . . . 25
6.0.20 while . . . 25

7 Associative arrays 25
7.1 Examples . . . 25
7.2 Types of values . . . 26
7.3 Number and types of indexes . . . 26
7.4 Array capacity . . . 26
7.5 Iteration, foreach . . . 26

8 Statistics (aggregates) 26
8.1 The aggregation (<<<) operator . . . 27
8.2 Extraction functions . . . 27
8.3 Integer extractors . . . 27
8.3.1 @count(s) . . . 27
8.3.2 @sum(s) . . . 27
8.3.3 @min(s) . . . 27
8.3.4 @max(s) . . . 27
8.3.5 @avg(s) . . . 27
8.4 Histogram extractors . . . 27
8.4.1 @hist_linear . . . 27
8.4.2 @hist_log . . . 28
8.4.3 Printing histograms . . . 28

9 Predefined functions 28
9.1 Logging functions . . . 28
9.1.1 error . . . 28
9.1.2 log . . . 28
9.1.3 print . . . 29
9.1.4 printf . . . 29
9.1.5 sprint . . . 30
9.1.6 sprintf . . . 30
9.1.7 system . . . 30
9.1.8 warn . . . 31
9.2 Task context at the probe point . . . 31
9.2.1 backtrace . . . 31
9.2.2 caller . . . 31
9.2.3 caller_addr . . . 31
9.2.4 cpu . . . 31
9.2.5 egid . . . 31
9.2.6 euid . . . 32
9.2.7 execname . . . 32
9.2.8 gid . . . 32
9.2.9 is_return . . . 32
9.2.10 pexecname . . . 32
9.2.11 pid . . . 32

9.2.12 ppid . . . 32
9.2.13 print_backtrace . . . 33
9.2.14 print_regs . . . 33
9.2.15 print_stack . . . 33
9.2.16 stack_size . . . 33
9.2.17 stack_unused . . . 33
9.2.18 stack_used . . . 33
9.2.19 stp_pid . . . 34
9.2.20 target . . . 34
9.2.21 task_cpu . . . 34
9.2.22 task_current . . . 34
9.2.23 task_egid . . . 34
9.2.24 task_execname . . . 34
9.2.25 task_euid . . . 34
9.2.26 task_gid . . . 35
9.2.27 task_nice . . . 35
9.2.28 task_parent . . . 35
9.2.29 task_pid . . . 35
9.2.30 task_prio . . . 35
9.2.31 task_state . . . 35
9.2.32 task_tid . . . 36
9.2.33 task_uid . . . 36
9.2.34 tid . . . 36
9.2.35 uid . . . 36
9.3 Accessing string data at a probe point . . . 36
9.3.1 kernel_string . . . 36
9.3.2 user_string . . . 36
9.3.3 user_string2 . . . 37
9.3.4 user_string_warn . . . 37
9.3.5 user_string_quoted . . . 37
9.4 Initializing queue statistics . . . 37
9.4.1 qs_wait . . . 37
9.4.2 qs_run . . . 37
9.4.3 qs_done . . . 38
9.5 Using queue statistics . . . 38
9.5.1 qsq_blocked . . . 38
9.5.2 qsq_print . . . 38
9.5.3 qsq_service_time . . . 38
9.5.4 qsq_start . . . 38
9.5.5 qsq_throughput . . . 38
9.5.6 qsq_utilization . . . 39
9.5.7 qsq_wait_queue_length . . . 39
9.5.8 qsq_wait_time . . . 39
9.5.9 A queue example . . . 39
9.6 Probe point identification . . . 40
9.6.1 pp . . . 40
9.6.2 probefunc . . . 40
9.6.3 probemod . . . 40

9.7 Formatting functions . . . 40
9.7.1 ctime . . . 41
9.7.2 errno_str . . . 41
9.7.3 returnstr . . . 41
9.7.4 thread_indent . . . 41
9.7.5 thread_timestamp . . . 41
9.8 String functions . . . 41
9.8.1 isinstr . . . 42
9.8.2 strlen . . . 42
9.8.3 substr . . . 42
9.8.4 text_str . . . 42
9.8.5 text_strn . . . 42
9.9 Timestamps . . . 42
9.9.1 get_cycles . . . 42
9.9.2 gettimeofday_ms . . . 43
9.9.3 gettimeofday_ns . . . 43
9.9.4 gettimeofday_s . . . 43
9.9.5 gettimeofday_us . . . 43
9.10 Miscellaneous tapset functions . . . 43
9.10.1 addr_to_node . . . 43
9.10.2 exit . . . 43
9.10.3 system . . . 44

10 For further reference 44

1 SystemTap overview

1.1 About this guide

This guide is a comprehensive reference to SystemTap's language constructs and syntax. The contents borrow heavily from existing SystemTap documentation found in manual pages and the tutorial. The presentation of information here provides the reader with a single place to find language syntax and recommended usage. In order to successfully use this guide, you should be familiar with the general theory and operation of SystemTap. If you are new to SystemTap, you will find the tutorial to be an excellent place to start learning. For detailed information about tapsets, see the manual pages provided with the distribution. For information about the entire collection of SystemTap reference material, see Section 10 on page 44

1.2 Reasons to use SystemTap

SystemTap provides infrastructure to simplify the gathering of information about a running Linux kernel so that it may be further analyzed. This analysis assists in identifying the underlying cause of a performance or functional problem. SystemTap was designed to eliminate the need for a developer to go through the tedious instrument, recompile, install, and reboot sequence normally required to collect this kind of data. To do this, it provides a simple command-line interface and scripting language for writing kernel instrumentation. With SystemTap, developers, system administrators, and users can easily write scripts that gather and manipulate kernel data that is not otherwise available using standard Linux tools. Users of SystemTap will find it to be a significant improvement over older methods.

1.3 Event-action language

SystemTap's language is strictly typed, declaration free, procedural, and inspired by dtrace and awk. Source code points or events in the kernel are associated with handlers, which are subroutines that are executed synchronously. These probes are conceptually similar 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.

1.4 Sample SystemTap scripts

Here are some example scripts that illustrate the basic operation of SystemTap. For more examples, see the examples/small_demos/ directory in the source directory, the SystemTap wiki `http://sourceware.org/systemtap/wiki/HomePage`, or the War Stories `http://sourceware.org/systemtap/wiki/WarStories` page.

1.4.1 Basic SystemTap syntax and control structures

The following code examples demonstrate SystemTap syntax and control structures.

```
global odds, evens
probe begin {
  # "no" and "ne" are local integers
  for (i=0; i<10; i++) {
    if (i % 2) odds [no++] = i
    else evens [ne++] = i
  }
  delete odds[2]
```

```
  delete evens[3]
  exit ()
}
probe end {
  foreach (x+ in odds) {
    log("odds[" . sprint(x) . "] = " . sprint(odds[x]))
  }
  foreach (x in evens-) {
    log("evens[" . sprint(x) . "] = " . sprint(evens[x]))
  }
}
```

This prints:

```
odds[1] = 1
odds[3] = 5
odds[4] = 7
odds[5] = 9
evens[5] = 8
evens[4] = 6
evens[2] = 2
```

Note that the evens[1] key is missing, since its value was zero. Any array element with an "empty" value (zero or empty string) is considered not present in the array, as if it was deleted. Note that all variable types are inferred, and that all locals and globals are initialized.

1.4.2 Primes between 0 and 49

```
function isprime (x) {
  if (x < 2) return 0
  for (i=2; i<x; i++) {
    if (x % i == 0) return 0
    if (i * i > x) break
  }
  return 1
}
probe begin {
  for (i=0; i<50; i++)
    if (isprime (i)) printf("%d\n", i)
  exit()
}
```

This prints:

```
2
3
5
```

```
7
11
13
17
19
23
29
31
37
41
43
47
```

1.4.3 Recursive functions

```
function fibonacci(i) {
  if (i < 1) error ("bad number")
  if (i == 1) return 1
  if (i == 2) return 2
  return fibonacci (i-1) + fibonacci (i-2)
}
probe begin {
  printf("11th fibonacci number: %d\n", fibonacci (11))
  exit ()
}
```

This prints:

```
11th fibonacci number: 118
```

Any larger number input to the function may exceed the MAXACTION or MAXNESTING limits, which will be caught by the parser and result in an error. For more about limits see Section 1.6.

1.5 The stap command

The stap program is the front-end to the SystemTap tool. It accepts probing instructions written in its scripting language, 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 or 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, the script voluntarily invokes the exit() function, or by a sufficient number of soft errors.

The stap command does the following:

- 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

For a full list of options to the stap command, see the stap(1) manual page.

1.6 Safety and security

SystemTap is an administrative tool. It exposes kernel internal data structures and potentially private user information. It acquires root privileges to actually run the kernel objects it builds using the **sudo** command, applied to the **staprun** program. staprun is a part of the SystemTap package, dedicated to module loading and unloading and kernel-to-user data transfer. Since staprun does not perform any additional security checks on the kernel objects it is given, do not give elevated privileges via sudo to untrusted users.

The translator asserts certain safety constraints. It ensures that no handler routine can run for too long, allocate memory, perform unsafe operations, or unintentionally interfere with the kernel. Use of script global variables is locked to protect against manipulation by concurrent probe handlers. Use of *guru mode* constructs such as embedded C (see Section 3.5) can violate these constraints, leading to a kernel crash or data corruption.

The resource use limits are set by macros in the generated C code. These may be overridden with the -D flag. A selection of these includes the following:

MAXNESTING - Maximum number of recursive function call levels, default 10.

MAXSTRINGLEN - Maximum length of strings, default 128.

MAXTRYLOCK - Maximum number of iterations to wait for locks on global variables before declaring possible deadlock and skipping the probe, default 1000.

MAXACTION - Maximum number of statements to execute during any single probe hit, default 1000.

MAXMAPENTRIES - Maximum number of rows in any single global array, default 2048.

MAXERRORS - Maximum number of soft errors before an exit is triggered, default 0, which means that the first error will exit the script.

MAXSKIPPED - Maximum number of skipped reentrant probes before an exit is triggered, default 100.

MINSTACKSPACE - Minimum number of free kernel stack bytes required in order to run a probe handler, default 1024. This number should be large enough for the probe handler's own needs, plus a safety margin.

If something goes wrong with stap or staprun after a probe has started running, you may safely kill both user processes, and remove the active probe kernel module with the rmmod command. Any pending trace messages may be lost.

2 Types of SystemTap scripts

2.1 Probe scripts

Probe scripts are analogous to programs; these scripts identify probe points and associated handlers.

2.2 Tapset scripts

Tapset scripts are libraries of probe aliases and auxiliary functions.

The /usr/share/systemtap/tapset directory contains tapset scripts. While these scripts look like regular SystemTap scripts, they cannot be run directly.

3 Components of a SystemTap script

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

The following example shows how to trace entry and exit from a function using two probes.

```
probe kernel.function("sys_mkdir") { log ("enter") }
probe kernel.function("sys_mkdir").return { log ("exit") }
```

To list the probe-able functions in the kernel, use the last-pass option to the translator. The output needs to be filtered because each inlined function instance is listed separately. The following statement is an example.

```
$ stap -p2 -e 'probe kernel.function("*") {}' | sort | uniq
```

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 others using aliases. The provided probe points 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 location. These *target variables* are presented to the script as variables whose names are prefixed with a dollar sign ($). They may be accessed only if the compiler used to compile the kernel preserved them, despite optimization. This is the same constraint imposed by a debugger when working with optimized code. Other events may have very little context.

3.2 Probe aliases

The general syntax is as follows.

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

New probe points may be defined using *aliases*. A probe point alias looks similar to probe definitions, but instead of activating a probe at the given point, it defines a new probe point name as an alias to an existing one. New probe aliases may refer to one or more existing probe aliases. The following is an example.

```
probe socket.sendmsg = kernel.function ("sock_sendmsg") { ... }
probe socket.do_write = kernel.function ("do_sock_write") { ... }
probe socket.send = socket.sendmsg, socket.do_write {}
```

There are two types of aliases, the prologue style and the epilogue style which are identified by the equal sign (=) and "+=" respectively.

A probe that names the new probe point will create an actual probe, with the handler of the alias *pre-pended*.

This pre-pending behavior serves several purposes. It allows the alias definition to pre-process the context of the probe before passing control to the handler specified by the user. This has several possible uses, demonstrated as follows.

```
# Skip probe unless given condition is met:
if ($flag1 != $flag2) next
# Supply values describing probes:
name = "foo"
# Extract the target variable to a plain local variable:
var = $var
```

3.2.1 Prologue-style aliases (=)

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

```
# Defines a new probe point syscall.read, which expands to
# kernel.function("sys_read"), with the given statement as
# a prologue.
#
probe syscall.read = kernel.function("sys_read") {
   fildes = $fd
}
```

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. The following is an example:

```
# Defines a new probe point with the given statement as an
# epilogue.
#
probe syscall.read += kernel.function("sys_read") {
   fildes = $fd
}
```

3.2.3 Probe alias usage

Another probe definition may use a previously defined alias. The following is an example.

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

3.2.4 Unused alias variables

An unused alias variable is a variable defined in a probe alias, usually as one of a group of `var = $var` assignments, which is not actually used by the script probe that instantiates the alias. These variables are discarded.

3.3 Variables

Identifiers for variables and functions are an alphanumeric sequence, and may include the underscore (_) and the dollar sign ($) characters. They may not start with a plain digit. Each variable is by default local to the probe or function statement block where 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 tuple of strings or integers serves as a key. Arrays must be declared as global. Local arrays are not allowed.

The translator performs *type inference* on all identifiers, including array indexes and function parameters. Inconsistent type-related use of identifiers results in an error.

Variables may be declared global. Global variables are shared among all probes and live as long as the SystemTap session. There is one namespace for all global variables, regardless of in which script file they are found. Because of possible concurrency limits, such as due to multiple probe handlers running on different CPUs, each global variable used by a probe is automatically read- or write-locked while the handler is running. A global declaration may be written at the outermost level anywhere in a script file, not just within a block of code. The following declaration marks `var1` and `var2` as global. The translator will infer a value type for each, and if the variable is used as an array, its key types.

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

3.4 Auxiliary functions

General syntax:

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

SystemTap scripts may define subroutines to factor out common work. Functions may take any number of scalar arguments, and must return a single scalar value. Scalars in this context are integers or strings. For more information on scalars, see Section 3.3 on the preceding page and Section 5.2 on page 19. The following is an example function declaration.

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

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

```
function thatfn:string(arg1:long, arg2) {
        return sprintf("%d%s",arg1, arg2)
}
```

Functions may call others or themselves recursively, up to a fixed nesting limit. See Section 1.6 on page 10.

3.5 Embedded C

SystemTap supports a *guru mode* where script safety features such as code and data memory reference protection are removed. For complete details about these protections, see safety.tex. When in guru mode, the translator accepts embedded code enclosed between “%{” and “%}” markers in the script file. Embedded code is transcribed verbatim, without analysis, in sequence, into generated C code. At the outermost level of a script, guru mode may be useful to add #include instructions, or any auxiliary definitions for use by other embedded code.

3.6 Embedded C functions

General syntax:

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

Embedded code is permitted in a function body. In that case, the script language body is replaced entirely by a piece of C code enclosed between %{ and %} markers. The enclosed code may do anything reasonable and safe as allowed by the parser. There are a number of undocumented but complex safety constraints on concurrency, resource consumption, and runtime limits, so this is an advanced technique. Refer to the SystemTap source code for details.

The memory locations reserved for input and output values are provided to the function using a macro named `THIS`. The following are 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 must be inferred by the translator from the call sites in order for this to work. You should examine C code generated for ordinary script language functions to write compatible embedded-C.

4 Probe points

4.1 General syntax

The general probe point syntax is a dotted-symbol sequence. This divides the event namespace into parts, analogous to the style of the Domain Name System. Each component identifier is parameterized by a string or number literal, with a syntax analogous to a function call.

The following are all syntactically valid probe points.

```
kernel.function("foo").return
syscall(22)
user.inode("/bin/vi").statement(0x2222)
end
```

Probes may be broadly classified into *synchronous* and *asynchronous*. A synchronous event is occurs when any processor executes an instruction matched by the specification. This gives these probes a reference point (instruction address) from which more contextual data may be available. Other families of probe points refer to asynchronous events such as timers, where no fixed reference point is related. Each probe point specification may match multiple locations, such as by using wildcards or aliases, and all are probed. A probe declaration may contain several specifications separated by commas, which are all probed.

4.1.1 Prefixes: kernel, timer, and so on

Prefixes specify the probe target, such as kernel, module, timer, and so on.

4.1.2 Suffixes: .entry, .return

Suffixes specify the code path to probe, such as the return value of a probed function.

4.1.3 Wildcarded file names, function names

A component may include an asterisk (*) character, which expands to other matching probe points. An example follows.

```
kernel.syscall.*
```

4.1.4 Optional probe points

A probe point may be followed by a question mark (?) character, to indicate that it is optional, and that no error should result if it fails to expand. This effect passes down through all levels of alias or wildcard expansion.

The following is the general syntax.

```
kernel.function("no_such_function") ?
```

4.2 Built-in probe point types (DWARF probes)

This family of probe points uses symbolic debugging information for the target kernel, module, or program, as may be found in executables that have not been stripped, or the separate **debuginfo** packages. They allow placement of probes logically into the execution path of the target program by specifying a set of points in the source or object code. When a matching statement executes on any processor, the probe handler is run in that context.

Points in a kernel are identified by module, source file, line number, function name, C label name, or some combination of these.

Here is a list of probe point families currently supported.

- The .function variant places a probe near the beginning of the named function, so that parameters are available as context variables.
- The .return variant places a probe at the moment of return from the named function, so the return value is available as the $return context variable.
- The .inline variant is similar to .function but probes inline functions. Inline functions do not have an identifiable return point, so .return is not supported on .inline probes.
- The .statement variant places a probe at the exact spot, exposing those local variables that are visible there.

In the following probe descriptions, MPATTERN stands for a string literal that identies the loaded kernel module of interest. It may include asterisk (*), "[]", and question mark (?) wildcards. PATTERN stands for a string literal that identifies a point in the program. It is composed of three parts:

1. The first part is the name of a function, as would appear in the nm program's output. This part may use the asterisk and question mark wildcard operators to match multiple names.
2. The second part is optional, and begins with the ampersand (@) character. It is followed by a source file name wildcard pattern, such as mm/slab*.

3. The third part is optional if the file name part was given. It identifies the line number in the source file, preceded by a colon.

Alternately, specify PATTERN as a numeric constant to indicate a relative module address or an absolute kernel address.

Some of the source-level variables, such as function parameters, locals, or globals visible in the compilation unit, are visible to probe handlers. Refer to these variables by prefixing their name with a question mark within the scripts. In addition, a special syntax allows limited traversal of structures, pointers, and arrays.

`$var` refers to an in-scope variable var. If it is a type similar to an integer, it will be cast to a 64-bit integer for script use. Pointers similar to a string (char *) are copied to SystemTap string values bu the kernel_string or user_string functions.

`$var->field` traverses a structure's field. The indirection operator may be repeated to follow additional levels of pointers.

`$var[N]` indexes into an array. The index is given with a literal number.

4.2.1 kernel.function

The .function variant places a probe near the beginning of the named function, so that parameters are available as context variables.

General syntax:

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

The following are examples.

```
# Refers to all kernel functions with "init" or "exit"
# in the name:
kernel.function("*init*"), kernel.function("*exit*")
# Refers to any functions within the "kernel/sched.c"
# file that span line 240:
kernel.function("*@kernel/sched.c:240")
```

4.2.2 kernel.inline

The .inline variant is similar to .function but probes inline functions. Inline functions do not have an identifiable return point, so .return is not supported on .inline probes. The following is the general syntax.

```
kernel.inline(PATTERN)
```

Example:

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

4.2.3 kernel.statement

The .statement variant places a probe at the exact spot, exposing those local variables that are visible there. The following is the general syntax.

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

Example:

```
# Refers to the statement of line 2917 within the
# "kernel/sched.c":
kernel.statement("*@kernel/sched.c:2917")
```

4.3 Marker probes

This family of probe points connects to static probe markers inserted into the kernel or a module. These markers are special macro calls in the kernel that make probing faster and more reliable than with DWARF-based probes. DWARF debugging information is not required to use probe markers.

Marker probe points begin with a kernel or module("name") prefix, the same as DWARF probes. This prefix identifies the source of the symbol table used for finding markers. The suffix names the marker itself: mark("name"). The marker name string, which may contain wildcard characters, is matched against the names given to the marker macros when the kernel or module was compiled.

The handler associated with a marker probe reads any optional parameters specified at the macro call site named $arg1 through $argNN, where NN is the number of parameters supplied by the macro. Number and string parameters are passed in a type-safe manner.

4.4 Timer probes

You can use intervals defined by the standard kernel jiffies timer to trigger probe handlers asynchronously. A *jiffy* is a kernel-defined unit of time typically between 1 and 60 msec. Two probe point variants are supported by the translator:

```
timer.jiffies(N)
timer.jiffies(N).randomize(M)
```

The probe handler runs every N jiffies. If the `randomize` component is given, a linearly distributed random value in the range [-M ... +M] is added to N every time the handler executes. N is restricted to a reasonable range (1 to approximately 1,000,000), and M is restricted to be less than N. There are no target variables provided in either context. Probes can be run concurrently multiple processors.

Intervals may be specified in units of time. There are two probe point variants similar to the jiffies timer:

```
timer.ms(N)
timer.ms(N).randomize(M)
```

Here, N and M are specified in milliseconds, but the full options for units are seconds (sec), milliseconds (msec), microseconds (μsec), nanoseconds (nsec), and hertz (Hz). Randomization is not supported for hertz timers.

The resolution of the timers depends on the target kernel. For kernels prior to 2.6.17, timers are limited to jiffies resolution, so intervals are rounded up to the nearest jiffies interval. After 2.6.17, the implementation uses hrtimers for tighter precision, though the resulting resolution will be dependent upon architecture. In either case, if the randomize component is given, then the random value will be added to the interval before any rounding occurs.

Profiling timers are available to provide probes that execute on all CPUs at each system tick. This probe takes no parameters, as follows.

```
timer.profile
```

Full context information of the interrupted process is available, making this probe suitable for implementing a time-based sampling profiler.

The following is an example of timer usage.

```
# Refers to a periodic interrupt, every 1000 jiffies:
timer.jiffies(1000)
```

```
# Fires every 5 seconds:
timer.sec(5)
# refers to a periodic interrupt, every 1000 +/- 200 jiffies:
timer.jiffies(1000).randomize(200)
```

4.5 Return probes

The .return variant places a probe at the moment of return from the named function, so that the return value is available as the $return context variable. Inline functions do not have an identifiable return point, so .return is not supported on .inline probes.

4.6 Special probe points

The probe points begin and end are defined by the translator to refer to the time of session startup and shutdown. There are no target variables available in either context.

4.6.1 begin

The `begin` probe is the start of the SystemTap session. All begin probe handlers are run during the startup of the session. All global variables will have been initialized prior to this point.

4.6.2 end

The `end` probe is the end of the SystemTap session. All end probes are run 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 shutdown triggered by error, `end` probes are not run.

4.6.3 Begin and end probe sequence

Begin and end probes are specified with an optional sequence number that controls the order in which they are run. If no sequence number is provided, the sequence number defaults to zero and probes are run in the order that they occur in the script file. Sequence numbers may be either positive or negative, and are especially useful for tapset writers who want to do initialization in a begin probe. The following are examples.

```
# In a tapset file:
probe begin(-1000) { ... }
# In a user script:
probe begin { ... }
```

The user script `begin` probe defaults to sequence number zero, so the tapset `begin` probe will run first.

4.6.4 never

The never probe point is defined by the translator to mean *never*. Its statements are analyzed for symbol and type correctness, but its probe handler is never run. This probe point may be useful in conjunction with optional probes. See Section 4.1.4.

4.7 Probes to monitor performance

The perfmon family of probe points is used to access the performance monitoring hardware available in modern processors. These probes points require perfmon2 support in the kernel to access the hardware.

Performance monitor hardware points have a `perfmon` prefix. The suffix names the event being counted, for example `counter("event")`. The event names are specific to the processor implementation, except for generic cycle and instructions events, which are available on all processors. The probe `perfmon.counter(“event”)` starts a counter on the processor which counts the number of events that occur on that processor. For more details on the performance monitoring events available on a specific processor, see the help text returned by typing the perfmon2 command `pfmon -l`.

4.7.1 $counter

$counter is a handle used in the body of a probe for operations involving the counter associated with the probe.

4.7.2 read_counter

read_counter is a function passed to the handle for a perfmon probe. It returns the current count for the event.

5 Language elements

5.1 Identifiers

Identifiers are used to name variables and functions. They are an alphanumeric sequence that may include the underscore and dollar sign characters. They have the same syntax as C identifiers, except that the dollar sign is also a legal character. Identifiers that begin with a dollar sign are interpreted as references to variables in the target software, rather than to SystemTap script variables. Identifiers may not start with a plain digit.

5.2 Data types

The SystemTap language includes a small number of data types, but no type declarations. A variable's type is inferred from its use. To support this inference, the translator enforces consistent typing of function arguments and return values, array indexes and values. There are no implicit type conversions between strings and numbers. Inconsistent type-related use of identifiers signals an error.

5.2.1 Numbers

Numbers are 64-bit signed integers. The parser will also accept (and wrap around) values above positive 2^{63}.

5.2.2 Literals

Literals are either strings or integers. Literals can be expressed in decimal, octal, or hexadecimal, using C notation. Type suffixes (e.g., L or U) are not used.

5.2.3 Integers

Integers are decimal, hexadecimal, or octal, and use the same notation as in C. Integers are 64-bit signed quantities, although the parser also accepts (and wraps around) values above positive 2^{63}.

5.2.4 Strings

Strings are enclosed in double quotes ("string"), and pass through standard C escape codes with backslashes. Strings are limited in length to MAXSTRINGLEN. For more information about this and other limits, see Section 1.6 on page 10.

5.2.5 Associative arrays

See Section 7 on page 25

5.2.6 Statistics

See Section 8 on page 26

5.3 Semicolons

The semicolon is the null statement: do nothing. It is optional, and useful as a separator between statements to improve detection of syntax errors and to reduce ambiguities in grammar.

5.4 Comments

Three forms of comments are supported, as follows.

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

5.5 Whitespace

As in C, spaces, tabs, returns, newlines, and comments are treated as whitespace. Whitespace is ignored by the parser.

5.6 Expressions

SystemTap supports a number of operators that use the same general syntax, semantics, and precedence as in C and awk. Arithmetic is performed per C rules for signed integers. The parser detects division by zero or an overflow and generates an error. The following subsections list these operators.

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

General syntax:

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

5.6.9 $ptr->member

`ptr` is a kernel pointer available in a probed context.

5.6.10 <value> in <array_name>

This expression evaluates to true if the array contains an element with the specified index.

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

The number of index values must match the number of indexes previously specified.

5.7 Literals passed in from the stap command line

Literals are either strings enclosed in double quotes (" ") or integers. For information about integers, see Section 5.2.3 on the preceding page. For information about strings, see Section 5.2.4 on the facing page.

Script arguments at the end of a command line are expanded as literals. You can use these in all contexts where literals are accepted. A reference to a nonexistent argument number 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 a string literal.

5.8 Conditional compilation

5.8.1 Conditions

One of the steps of parsing is a simple conditional preprocessing stage. The general form of this is similar to the ternary operator (Section 5.6.6 on the previous page)

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

The CONDITION is a limited expression whose format is determined by its first keyword. The following is the general syntax.

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

5.8.2 Conditions based on kernel version: kernel_v, kernel_vr

If the first part of a conditional expression is the identifier `kernel_v` or `kernel_vr`, the second part must be one of six standard numeric comparison operators ”<”, ”<=”, ”==”, ”!=”, ”>”, or ”>=”, and the third part must be a string literal that contains an RPM-style version-release value. The condition returns true if the version of the target kernel (as optionally overridden by the **-r** option) matches the given version string. The comparison is performed by the glibc function strverscmp.

`kernel_v` refers to the kernel version number only, such as “2.6.13".

`kernel_vr` refers to the kernel version number including the release code suffix, such as “2.6.13-1.322FC3smp”.

5.8.3 Conditions based on architecture: arch

If the first part of the conditional expression is the identifier `arch` which refers to the processor architecture, then the second part is a string comparison operator ”==” or ”!=”, and the third part is a string literal for matching it. This comparison is a simple string equality or inequality.

5.8.4 True and False Tokens

TRUE-TOKENS and FALSE-TOKENS are zero or more general parser tokens, possibly including nested preprocessor conditionals, that 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 adapts 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 within functions and probe handlers. The total number of statements executed in response to any single probe event is limited to MAXACTION, which defaults to 1000. See Section 1.6 on page 10.

6.0.5 break and continue

Use `break` or `continue` to exit or iterate the innermost nesting loop statement, such as within a `while` or `for` or `foreach` statement. The syntax and semantics are the same as those used in C.

6.0.6 - (delete) <name>

See Section 7 on page 25

6.0.7 delete

`delete` removes an element.

The following statement removes 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[INDEX1, INDEX2, ...]
```

The following syntax removes all elements from ARRAY:

```
delete ARRAY
```

The following statement removes the value of SCALAR. Integers and strings are cleared to zero and null (`""`) respectively, while statistics are reset to their initial empty state:

```
delete SCALAR
```

6.0.8 do

The `do` statement has the same syntax and semantics as in C.

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

6.0.9 EXP (expression)

An `expression` executes a string- or integer-valued expression and discards the value.

6.0.10 for

The `for` statement is similar to the `for` statement in C.

The `for` expression executes EXP1 as initialization. While EXP2 is non-zero, it executes STMT, then the iteration expression EXP3. The syntax and semantics are the same as used in C.

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

6.0.11 foreach

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

The foreach statement loops over each element of a named global array, assigning the current key to VAR. The array must not be modified within the statement. If you add a single plus (+) or minus (-) operator after the VAR or the ARRAY identifier, the iteration order will be sorted by the ascending or descending index or value.

The following statement behaves the same, except it is used when an array is indexed with a tuple of keys. Use a sorting suffix on at most one VAR or ARRAY identifier.

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

The following statement is the same as the first example, except that the `limit` keyword limits the number of loop iterations to EXP times. EXP is evaluated once at the beginning of the loop.

```
foreach (VAR in ARRAY [ limit EXP ]) STMT
```

6.0.12 Use of plus (+) or minus (-) suffix for ascending or descending order

See Section 6.0.11 and Section 7 on the facing page.

6.0.13 goto

This command is not supported in SystemTap.

6.0.14 if

General syntax:

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

The `if` statement compares an integer-valued EXP to zero. It executes the first STMT if non-zero ,or the second STMT if zero.

The `if` command has the same syntax and semantics as used in C.

6.0.15 next

The `next` statement returns immediately from the enclosing probe handler.

6.0.16 ; (null statement)

General syntax:

```
statement1
;
statement2
```

The semicolon represents the null statement, or do nothing. It is useful as an optional separator between statements to improve syntax error detection and to handle certain grammar ambiguities.

6.0.17 return

General syntax:

```
return EXP
```

The `return` statement returns the EXP value from the enclosing function. If the value of the function is not reserved, then a return statement is not needed, and the function will have a special *unknown* type with no return value.

6.0.18 switch

The `switch` statement is not supported by SystemTap.

6.0.19 { } (statement block)

This is the statement block with zero or more statements enclosed within brackets. The following is the general syntax:

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

The statement block executes each statement in sequence in the block. Separators or terminators are generally not necessary between statements. The statement block uses the same syntax and semantics as in C.

6.0.20 while

General syntax:

```
while (EXP) STMT
```

The `while` statement uses the same syntax and semantics as in C. In the statement above, while the integer-valued EXP evaluates to non-zero, the parser will execute STMT.

7 Associative arrays

Associative arrays are implemented as hash tables with a maximum size set at startup. Associative arrays are too large to be created dynamically for individual probe handler runs, so they must be declared as global. The basic operations for arrays are setting and looking up elements. These operations are expressed in awk syntax: the array name followed by an opening bracket ([), a comma-separated list of index expressions, and a closing bracket (]). Each index expression may be a string or a number, as long as it is consistently typed throughout the script.

7.1 Examples

```
# Increment the named array slot:
foo [4,"hello"] ++
# Update a statistic:
processusage [uid(),execname()] ++
# Set a timestamp reference point:
```

```
times [tid()] = get_cycles()
# Compute a timestamp delta:
delta = get_cycles() - times [tid()]
```

7.2 Types of values

Unset array elements may be fetched, and return a null value (zero or an empty string) as appropriate. Assigning a null value to an array element deletes the element. SystemTap provides "syntactic sugar" for these operations, in the form of explicit membership testing and deletion. In the following example, the first and last statements show that you cannot test for membership of an element with a value of zero or null.

```
if ([4,"hello"] in foo) {}  # membership test
delete times[tid()]         # deletion of a single element
delete times                # deletion of all elements
if (foo[4,"hello"] == 0) {} # equivalent to the first statement above
```

7.3 Number and types of indexes

TBD

7.4 Array capacity

You can adjust array capacity by setting MAXMAPENTRIES, which defaults to 2048. See Section 1.6 on page 10.

7.5 Iteration, foreach

Like awk, SystemTap's foreach creates a loop that iterates over key tuples of an array, not only values. The iteration may be sorted by any single key or a value by adding an extra plus symbol (+) or minus symbol (-) to the code. The following are examples.

```
# Simple loop in arbitrary sequence:
foreach ([a,b] in foo) { fuss_with(foo[a,b]) }
# Loop in increasing sequence of value:
foreach ([a,b] in foo+) { }
# Loop in decreasing sequence of first key:
foreach ([a-,b] in foo) { }
```

The `break` and `continue` statements also work inside foreach loops. Since arrays can be large but probe handlers must execute quickly, you should write scripts that exit iteration early, if possible. For simplicity, SystemTap forbids any modification of an array during iteration with a foreach.

8 Statistics (aggregates)

Aggregate instances are used to collect statistics on numerical values, when it is important to accumulate new data quickly and in large volume. These instances operate without exclusive locks, and store only aggregated stream statistics. Aggregates make sense only for global variables. They are stored individually or as elements of an array.

8.1 The aggregation ($<<<$) operator

The aggregation operator is "$<<<$", and its effect are similar to an assignment or a C++ output streaming operation. The left operand specifies a scalar or array-index *l-value*, which must be declared global. The right operand is a numeric expression. The meaning is intuitive: add the given number to the set of numbers to compute their statistics. The specific list of statistics to gather is given separately by the extraction functions. The following is an example.

```
a <<< delta_timestamp
writes[execname()] <<< count
```

8.2 Extraction functions

For each instance of a distinct extraction function operating on a given identifier, the translator computes a set of statistics. With each execution of an extraction function, the aggregation is computed for that moment across all processors. The first argument of each function is the same style of l-value as used on the left hand side of the accumulate operation.

8.3 Integer extractors

The following functions give methods to extract information about integer values.

8.3.1 @count(s)

This statement returns the integer number of all values accumulated into s.

8.3.2 @sum(s)

This statement returns the integer total of all values accumulated into s.

8.3.3 @min(s)

This statement returns the integer minimum of all values accumulated into s.

8.3.4 @max(s)

This statement returns the integer maximum of all values accumulated into s.

8.3.5 @avg(s)

This statement returns the integer average of all values accumulated into s.

8.4 Histogram extractors

The following functions give methods to extract histogram information.

8.4.1 @hist_linear

The statement `@hist_linear(v,L,H,W)` represents a linear histogram `v` whose low/high/width parameters are given by the following three literal numbers: L, H, W.

8.4.2 @hist_log

The statement `@hist_log(v,N)` represents a base-2 logarithmic histogram with the given number of buckets. N may be omitted, and defaults to 64.

8.4.3 Printing histograms

Printing a histogram with the print family of functions renders a histogram object as a tabular bar chart. The following are examples.

```
probe foo {
   x <<< $value
}
probe end {
   printf ("avg %d = sum %d / count %d\n",
   @avg(x), @sum(x), @count(x))
   print (@hist_log(x))
}
```

9 Predefined functions

Unlike built-in functions, predefined functions are implemented in tapsets.

9.1 Logging functions

The following sections describe the logging functions you may use.

9.1.1 error

General syntax:

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

This function logs the given string to the error stream. It appends an implicit end-of-line. It blocks any further execution of statements in this probe. If the number of errors exceeds the MAXERRORS parameter, it triggers an `exit`.

9.1.2 log

General syntax:

```
log:unknown (msg:string)
log (const char *fmt, )
```

This function logs data. `log` sends the message immediately to staprun and to the bulk transport (relayfs) if it is being used. If the last character given is not a newline, then one is added.

This function is not as efficient as printf and should only be used for urgent messages. The functions dbug or warn are recommended.

9.1.3 print

General syntax:

```
print:unknown ()
```

This function prints a single value of any type.

9.1.4 printf

General syntax:

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

The printf function takes a formatting string as an argument, and a number of values of corresponding types, and prints them all. The printf formatting directives are similar to those of C, except that they are fully checked for type by the translator.

The %1n, %2n, %4n, %n functions instruct printf to write a binary length at the location that is the total length of the string written by printf. The field width is the number of bytes to write. The default is 2.

The binary write function are %1b, %2b, %4b, %8b, and %b. The field width specifies the number of bytes to write. The default width is 4 (32-bits). The following is an example.

```
$ stap -e 'probe begin{for(i=97;i<123;i++) \
  printf("%3d: %1b%1b%1b\n",i,i,i-32,i-64); \
  exit()}'
```

This prints:

```
 97: aA!
 98: bB"
 99: cC#
100: dD$
101: eE%
102: fF&
103: gG'
104: hH(
105: iI)
106: jJ*
107: kK+
108: lL,
109: mM-
110: nN.
111: oO/
112: pP0
113: qQ1
114: rR2
115: sS3
116: tT4
117: uU5
```

```
118: vV6
119: wW7
120: xX8
121: yY9
122: zZ:
```

Another example:

```
$ stap -e 'probe begin{printf("%1n%b%b", 0xc0dedbad, \
  0x12345678);exit()}' | hexdump -C
```

This prints:

```
00000000  08 ad db de c0 78 56 34  12                       |.....xV4.|
00000009
```

Another example:

```
$ stap -e 'probe begin{printf("%1b%1b%1blo \
  %1b%1brld\n", 72,101,108,87,111);exit()}'
```

This prints:

```
Hello World
```

9.1.5 sprint

General syntax:

```
sprint:unknown ()
```

This function operates like `print`, but returns the string rather than printing it.

9.1.6 sprintf

General syntax:

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

This function operates like `printf`, but returns the formatted string rather than printing it.

9.1.7 system

General syntax:

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

The system function runs a command on the system. The specified command runs in the background once the current probe completes.

9.1.8 warn

General syntax:

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

This function sends a warning message immediately to staprun. It is also sent over the bulk transport (relayfs) if it is being used. If the last character is not a newline, then one is added.

9.2 Task context at the probe point

The following functions provide ways to access the task context at a probe point.

9.2.1 backtrace

General syntax:

```
backtrace:string ()
```

This function returns a string of hex addresses that are a backtrace of the stack. The output is truncated to MAXSTRINGLEN.

9.2.2 caller

General syntax:

```
caller:string()
```

This function returns the address and name of the calling function. It works only for return probes.

9.2.3 caller_addr

General syntax:

```
caller_addr:long ()
```

This function returns the address of the calling function. It works only for return probes.

9.2.4 cpu

General syntax:

```
cpu:long ()
```

This function returns the current cpu number.

9.2.5 egid

General syntax:

```
egid:long ()
```

This function returns the effective GID of the current process.

9.2.6 euid

General syntax:

```
euid:long ()
```

This function returns the effective UID of the current process.

9.2.7 execname

General syntax:

```
execname:string ()
```

This function returns the name of the current process.

9.2.8 gid

General syntax:

```
gid:long ()
```

This function returns the GID of the current process.

9.2.9 is_return

General syntax:

```
is_return:long ()
```

This function returns 1 if the probe point is a return probe, else it returns zero. DEPRECATED.

9.2.10 pexecname

General syntax:

```
pexecname:string()
```

This function returns the name of the parent process.

9.2.11 pid

General syntax:

```
pid:long ()
```

This function returns the process ID of the current process.

9.2.12 ppid

General syntax:

```
ppid:long ()
```

This function returns the process ID of the parent process.

9.2.13 print_backtrace

General syntax:

```
print_backtrace:unknown ()
```

This function is equivalent to `print_stack(backtrace())`, except that deeper stack nesting is supported. The function does not return a value.

9.2.14 print_regs

General syntax:

```
print_regs:unknown ()
```

This function prints a register dump.

9.2.15 print_stack

General syntax:

```
print_stack:unknown (stk:string)
```

This function performs a symbolic lookup of the addresses in the given string, which is assumed to be the result of a prior call to `backtrace()`. It prints one line per address. Each printed line includes the address, the name of the function containing the address, and an estimate of its position within that function. The function does not return a value.

9.2.16 stack_size

General syntax:

```
stack_size:long ()
```

This function returns the size of the stack.

9.2.17 stack_unused

General syntax:

```
stack_unused:long ()
```

This function returns how many bytes are currently unused in the stack.

9.2.18 stack_used

General syntax:

```
stack_used:long ()
```

This function returns how many bytes are currently used in the stack.

9.2.19 stp_pid

```
stp_pid:long ()
```

This function returns the process ID of the stpd daemon.

9.2.20 target

General syntax:

```
target:long ()
```

This function returns the process ID of the target process. This is useful in conjunction with the -x PID or -c CMD command-line options to stap, for example to create scripts that filter on a specific process.

9.2.21 task_cpu

General syntax:

```
task_cpu:long (task:long)
```

This function returns the scheduled cpu for the given task.

9.2.22 task_current

General syntax:

```
task_current:long ()
```

This function returns the task_struct representing the current process.

9.2.23 task_egid

General syntax:

```
task_egid:long (task:long)
```

This function returns the effective GID of the given task.

9.2.24 task_execname

General syntax:

```
task_execname:string (task:long)
```

This function returns the name of the given task.

9.2.25 task_euid

General syntax:

```
task_euid:long (task:long)
```

This function returns the effective UID of the given task.

9.2.26 task_gid

General syntax:

```
task_gid:long (task:long)
```

This function returns the GID of the given task.

9.2.27 task_nice

General syntax:

```
task_nice:long (task:long)
```

This function returns the nice value of the given task.

9.2.28 task_parent

General syntax:

```
task_parent:long (task:long)
```

This function returns the parent task_struct of the given task.

9.2.29 task_pid

General syntax:

```
task_pid:long (task:long)
```

This function returns the process ID of the given task.

9.2.30 task_prio

General syntax:

```
task_prio:long (task:long)
```

This function returns the priority value of the given task.

9.2.31 task_state

General syntax:

```
task_state:long (task:long)
```

This function returns the state of the given task. Possible states are:

```
TASK_RUNNING           0
TASK_INTERRUPTIBLE     1
TASK_UNINTERRUPTIBLE   2
TASK_STOPPED           4
TASK_TRACED            8
EXIT_ZOMBIE           16
EXIT_DEAD             32
```

9.2.32 task_tid

General syntax:

```
task_tid:long (task:long)
```

This function returns the thread ID of the given task.

9.2.33 task_uid

General syntax:

```
task_uid:long (task:long)
```

This function returns the UID of the given task.

9.2.34 tid

General syntax:

```
tid:long ()
```

This function returns the ID of the current thread.

9.2.35 uid

General syntax:

```
uid:long ()
```

This function returns the UID of the current process.

9.3 Accessing string data at a probe point

The following functions provide methods to access string data at a probe point.

9.3.1 kernel_string

General syntax:

```
kernel_string:string (addr:long)
```

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

9.3.2 user_string

General syntax:

```
user_string:string (addr:long)
```

This function copies a string from user space at a given address. The validation of this address is only partial. In rare cases when userspace data is not accessible, this function returns `<unknown>`.

9.3.3 user_string2

General syntax:

```
user_string2:string (addr:long, err_msg:string)
```

This function is similar to `user_string`, (Section 9.3.2 on the facing page) but allows passing an error message as an argument to be returned if userspace data is not available.

9.3.4 user_string_warn

General syntax:

```
user_string_warn:string (addr:long)
```

This function copies a string from userspace at given address. It prints a verbose error message on failure.

9.3.5 user_string_quoted

General syntax:

```
user_string_quoted:string (addr:long)
```

This function copies a string from userspace at given address. Any ASCII characters that are not printable are replaced by the corresponding escape sequence in the returned string.

9.4 Initializing queue statistics

The queue_stats tapset provides functions that, when given notification of queuing events like wait, run, or done, track averages such as queue length, service and wait times, and utilization. Call the following three functions from appropriate probes, in sequence.

9.4.1 qs_wait

General syntax:

```
qs_wait:unknown (qname:string)
```

This function records that a new request was enqueued for the given queue name.

9.4.2 qs_run

General syntax:

```
qs_run:unknown (qname:string)
```

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

9.4.3 qs_done

General syntax:

```
qs_done:unknown (qname:string)
```

This function records that a request originally from the given queue has completed being serviced.

9.5 Using queue statistics

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

9.5.1 qsq_blocked

General syntax:

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

This function returns the fraction of elapsed time during which the wait queue was used.

9.5.2 qsq_print

General syntax:

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

This function prints a line containing statistics for the given queue.

9.5.3 qsq_service_time

General syntax:

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

This function returns the average time required to service a request.

9.5.4 qsq_start

General syntax:

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

This function resets the statistics counters for the given queue, and restarts tracking from the moment the function was called. This command is used to create a queue.

9.5.5 qsq_throughput

General syntax:

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

This function returns the average rate of requests per scale units of time.

9.5.6 qsq_utilization

General syntax:

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

This function returns the elapsed time during which a resource was utilized.

9.5.7 qsq_wait_queue_length

General syntax:

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

This function returns the average length of a wait queue.

9.5.8 qsq_wait_time

General syntax:

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

This function returns the average time elapsed during a request was enqueued to completed.

9.5.9 A queue example

The following is an example from src/testsuite/systemtap.samples/queue_demo.stp:

```
probe begin {
  qsq_start ("block-read")
  qsq_start ("block-write")
}
probe timer.ms(3500), end {
  qsq_print ("block-read")
  qsq_start ("block-read")
  qsq_print ("block-write")
  qsq_start ("block-write")
}
probe timer.ms(10000) { exit () }
# synthesize queue work/service using three randomized "threads" for each queue.
global tc
function qs_doit (thread, name) {
  n = tc[thread] = (tc[thread]+1) % 3 # per-thread state counter
  if (n==1) qs_wait (name)
  else if (n==2) qs_run (name)
  else if (n==0) qs_done (name)
}
probe timer.ms(100).randomize(100) { qs_doit (0, "block-read") }
probe timer.ms(100).randomize(100) { qs_doit (1, "block-read") }
probe timer.ms(100).randomize(100) { qs_doit (2, "block-read") }
probe timer.ms(100).randomize(100) { qs_doit (3, "block-write") }
```

```
probe timer.ms(100).randomize(100) { qs_doit (4, "block-write") }
probe timer.ms(100).randomize(100) { qs_doit (5, "block-write") }
```

This prints:

```
block-read: 9 ops/s, 1.090 qlen, 215749 await, 96382 svctm, 69% wait, 64% util
block-write: 9 ops/s, 0.992 qlen, 208485 await, 103150 svctm, 69% wait, 61% util
block-read: 9 ops/s, 0.968 qlen, 197411 await, 97762 svctm, 63% wait, 63% util
block-write: 8 ops/s, 0.930 qlen, 202414 await, 93870 svctm, 60% wait, 56% util
block-read: 8 ops/s, 0.774 qlen, 192957 await, 99995 svctm, 58% wait, 62% util
block-write: 9 ops/s, 0.861 qlen, 193857 await, 101573 svctm, 56% wait, 64% util
```

9.6 Probe point identification

The following functions help you identify probe points.

9.6.1 pp

General syntax:

```
pp:string ()
```

This function returns the probe point associated with a currently running probe handler, including alias and wild-card expansion effects.

9.6.2 probefunc

General syntax:

```
probefunc:string ()
```

This function returns the probe point function name.

9.6.3 probemod

General syntax:

```
probemod:string ()
```

This function returns the probe point module name.

9.7 Formatting functions

The following functions help you format output.

9.7.1 ctime

General syntax:

```
ctime:string(epochsecs:long)
```

This function accepts an argument of seconds since the epoch as returned by `gettimeofday_s()`. It returns a date string in UTC of the form:

```
"Wed Jun 30 21:49:008 1993"
```

9.7.2 errno_str

General syntax:

```
errno_str:string (err:long)
```

This function returns the symbolic string associated with the given error code, such as ENOENT for the number 2, or E#3333 for an out-of-range value like 3333.

9.7.3 returnstr

General syntax:

```
returnstr:string (returnp:long)
```

This function is used by the syscall tapset, and returns a string. Set returnp equal to 1 for decimal, or 2 for hex.

9.7.4 thread_indent

General syntax:

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

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

9.7.5 thread_timestamp

General syntax:

```
thread_timestamp:long ()
```

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

9.8 String functions

The following are string function you can use.

9.8.1 isinstr

General syntax:

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

This function returns 1 if string s1 contains string s2, otherwise zero.

9.8.2 strlen

General syntax:

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

This function returns the number of characters in str.

9.8.3 substr

General syntax:

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

This function returns the substring of `str` starting from character `start` and ending at character `stop`.

9.8.4 text_str

General syntax:

```
text_str:string(input:string)
```

This function accepts a string argument. Any ASCII characters in the string that are not printable are replaced by a corresponding escape sequence in the returned string.

9.8.5 text_strn

General syntax:

```
text_strn:string(input:string, len:long, quoted:long)
```

This function accepts a string of length `len`. Any ASCII characters that are not printable are replaced by a corresponding escape sequence in the returned string. If `quoted` is not null, the function adds a backslash character to the output.

9.9 Timestamps

The following functions provide methods to extract time data.

9.9.1 get_cycles

General syntax:

```
get_cycles:long ()
```

This function returns the processor cycle counter value if available, else it returns zero.

9.9.2 gettimeofday_ms

General syntax:

```
gettimeofday_ms:long ()
```

This function returns the number of milliseconds since the UNIX epoch.

9.9.3 gettimeofday_ns

General syntax:

```
gettimeofday_ns:long ()
```

This function returns the number of nanoseconds since the UNIX epoch.

9.9.4 gettimeofday_s

General syntax:

```
gettimeofday_s:long ()
```

This function returns the number of seconds since the UNIX epoch.

9.9.5 gettimeofday_us

General syntax:

```
gettimeofday_us:long ()
```

This function returns the number of microseconds since the UNIX epoch.

9.10 Miscellaneous tapset functions

The following are miscellaneous functions.

9.10.1 addr_to_node

General syntax:

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

This function accepts an address, and returns the node that the given address belongs to in a NUMA system.

9.10.2 exit

General syntax:

```
exit:unknown ()
```

This function enqueues a request to shut down the SystemTap session. It does not unwind the current probe handler, nor block new probe handlers. The stap daemon will respond to the request and initiate an ordered shutdown.

9.10.3 system

General syntax:

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

This function runs a command on the system. The command will run in the background when the current probe completes.

10 For further reference

For more information, see:

The SystemTap tutorial `http://sourceware.org/systemtap/tutorial/`

The SystemTap wiki `http://sourceware.org/systemtap/wiki`

The SystemTap documentation page `http://sourceware.org/systemtap/documentation.html`

From an unpacked source tarball, or CVS directory, the tapsets located in `src/tapset`, and the test scripts located in `src/testsuite/systemtap.samples`.

The man pages for tapsets. For a list, run the command "`apropos stapprobes`". The following is an example of the output returned by that command:

```
stapprobes           (5)  - systemtap probe points
stapprobes.iosched [stapprobes] (5)  - systemtap IO scheduler probe points
stapprobes.netdev [stapprobes] (5)  - systemtap network device probe points
stapprobes.nfs [stapprobes] (5)  - systemtap NFS client side probe points
stapprobes.nfsd [stapprobes] (5)  - systemtap NFS server side probe points
stapprobes.pagefault [stapprobes] (5)  - systemtap pagefault probe points
stapprobes.process [stapprobes] (5)  - systemtap process probe points
stapprobes.rpc [stapprobes] (5)  - systemtap SunRPC probe points
stapprobes.scsi [stapprobes] (5)  - systemtap scsi probe points
stapprobes.signal [stapprobes] (5)  - systemtap signal probe points
stapprobes.socket [stapprobes] (5)  - systemtap socket probe points
stapprobes.tcp [stapprobes] (5)  - systemtap tcp probe points
stapprobes.udp [stapprobes] (5)  - systemtap udp probe points
```

Index

+=, 12
-, 23
;, 20, 24
< < <, 27
=, 12
?, 15, 21
$, 21
, 25

addr_to_node, 43
aggregates, 26
arch, 22
associative arrays, 25
asynchronous, 14
auxiliary functions, 13
avg, 27

backtrace, 31
begin, 18
binary, 20
break, 23
built-in probes, 15

caller, 31
caller_addr, 31
comments, 20
comparison, 21
conditions, 22
constraints, 10
continue, 23
count, 27
cpu, 31
ctime, 41

data types, 19
delete, 23
do, 23
dwarf probes, 15

egid, 31
embedded C, 13
end, 18
epilogue-style aliases, 12
errno_str, 41
error, 28
euid, 32

example scripts, 7
execname, 32
exit, 43
expression, 23
expressions, 20

fn, 21
for, 23
foreach, 24, 26
formatting, 40

get_cycles, 42
gettimeofday_ s, 43
gettimeofday_ms, 43
gettimeofday_ns, 43
gettimeofday_us, 43
gid, 32
global, 13
goto, 24
grouping, 21
guru mode, 13

hist_linear, 27
hist_log, 28
histograms, 27

identifiers, 19
if, 24
index, 21
inference, 19
integers, 20
is_return, 32
isinstr, 42

jiffies, 17

kernel version, 22
kernel.function, 16
kernel.inline, 16
kernel.statement, 16
kernel_string, 36
kernel_v, 22
kernel_vr, 22

language, 7
lextraction, 27

limits, 10
literals, 19, 21
local arrays, 13
log, 28

max, 27
MAX* limit variables, 10
milliseconds, 17
min, 27

never, 18
next, 24
null statement, 24
numbers, 19
numeric, 20

pexecname, 32
pid, 32
pointer, 21
pp, 40
ppid, 32
prefixes, 14
print, 29
print_backtrace, 33
print_regs, 33
print_stack, 33
printf, 29
probe aliases, 11
probe points, 14
probe syntax, 14
probefunc, 40
prologue-style aliases, 12

qs_done, 38
qs_run, 37
qs_wait, 37
qsq wait_queue_length, 39
qsq_blocked, 38
qsq_print, 38
qsq_service_time, 38
qsq_start, 38
qsq_throughput, 38
qsq_utilization, 39
qsq_wait_time, 39
queue statistics, 37

randomize, 17
recursion, 9
return, 25
return probes, 18
returnstr, 41

sequence, 18
sprint, 30
sprintf, 30
stack_size, 33
stack_unused, 33
stack_used, 33
stap, 9
statement block, 25
stp_pid, 34
string, 41
strings, 20
strlen, 42
substr, 42
suffixes, 15
sum, 27
switch, 25
synchronous, 14
system, 30, 44

target, 34
target variables, 11
task_cpu, 34
task_current, 34
task_egid, 34
task_euid, 34
task_execname, 34
task_gid, 35
task_nice, 35
task_parent, 35
task_pid, 35
task_prio, 35
task_state, 35
task_tid, 36
task_uid, 36
text_str, 42
text_strn, 42
THIS, 14
thread_indent, 41
thread_timestamp, 41
tid, 36
timer probes, 17
timestamps, 42
tokens, 22

uid, 36
unary, 21

unused variables, 12
user_string, 36
user_string2, 37
user_string_quoted, 37
user_string_warn, 37

variables, 12

warn, 31
while, 25
whitespace, 20
wildcards, 15