SystemTap Language Reference

July 14, 2023

This document was derived from other documents contributed to the SystemTap project by employees of Red Hat, IBM and Intel.

Contents

1 SystemTap overview

1.1 About this guide

This guide is a comprehensive reference of 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 11

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 instrumentation for both kernel and user space. With SystemTap, developers, system administrators, and users can easily write scripts that gather and manipulate system data that is otherwise unavailable from 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

Following 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 at `http://sourceware.org/systemtap/wiki/HomePage`, or the SystemTap War Stories at `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)
        printf ("odds[%d] = %d", x, odds[x])

    foreach (x in evens-)
        printf ("evens[%d] = %d", x, evens[x])
}
```

This prints:

```
odds[0] = 1
odds[1] = 3
odds[3] = 7
odds[4] = 9
evens[4] = 8
evens[2] = 4
evens[1] = 2
evens[0] = 0
```

Note that all variable types are inferred, and that all locals and globals are initialized. Integers are set to 0 and strings are set to the empty string.

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", 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 at run time 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 SystemTap script runs until one of the following conditions occurs:

- The user interrupts the script with a CTRL-C.
- The script executes the exit() function.
- The script encounters a sufficient number of soft errors.
- The monitored command started with the stap program's `-c` option exits.

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 requires 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. The following list describes a selection of these macros:

MAXNESTING – The maximum number of recursive function call levels. The default is 10.

MAXSTRINGLEN – The maximum length of strings. The default is 256 bytes for 32 bit machines and 512 bytes for all other machines.

MAXTRYLOCK – The maximum number of iterations to wait for locks on global variables before declaring possible deadlock and skipping the probe. The default is 1000.

MAXACTION – The maximum number of statements to execute during any single probe hit. The default is 1000.

MAXMAPENTRIES – The maximum number of rows in an array if the array size is not specified explicitly when declared. The default is 2048.

MAXERRORS – The maximum number of soft errors before an exit is triggered. The default is 0.

MAXSKIPPED – The maximum number of skipped reentrant probes before an exit is triggered. The default is 100.

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

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

To list the probe-able functions in the kernel, use the listing option (`-l`). For example:

```
$ stap -l 'kernel.function("*")' | sort
```

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(3)`, `tapset::*(3stap)`, and `probe::*(3stap)` man pages. The STMT statement block is executed whenever ıany of the named PROBEPOINT events occurs.

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* (or "context 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. Multiple aliases may share the same underlying probe points. 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 uses a probe point alias 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. It is not useful to define new variables there (since no subsequent code will see them), but rather the code can take action based upon variables set by the prologue or by the user code. The following is an example:

```
# Defines a new probe point with the given statement as an
# epilogue.
#
probe syscall.read += kernel.function("sys_read") {
    if (traceme) println ("tracing me")
}
```

3.2.3 Probe alias usage

A probe alias is used the same way as any built-in probe type, by naming it:

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

3.2.4 Alias suffixes

It is possible to include a suffix with a probe alias invocation. If only the initial part of a probe point matches an alias, the remainder is treated as a suffix and attached to the underlying probe point(s) when the alias is expanded. For example:

```
/* Define an alias: */
probe sendrecv = tcp.sendmsg, tcp.recvmsg { ... }

/* Use the alias in its basic form: */
probe sendrecv { ... }

/* Use the alias with an additional suffix: */
probe sendrecv.return { ... }
```

Here, the second use of the probe alias is equivalent to writing `probe tcp.sendmsg.return, tcp.recvmsg.return`.

As another example, the probe points `tcp.sendmsg.return` and `tcp.recvmsg.return` are actually defined as aliases in the tapset `tcp.stp`. They expand to a probe point of the form `kernel.function("...").return`, so they can also be suffixed:

```
probe tcp.sendmsg.return.maxactive(10) {
    printf("returning from sending %d bytes\n", size)
}
```

Here, the probe point expands to `kernel.function("tcp_sendmsg").return.maxactive(10)`.

3.2.5 Alias suffixes and wildcards

When expanding wildcards, SystemTap generally avoids considering alias suffixes in the expansion. The exception is when a wildcard element is encountered that does not have any ordinary expansions. Consider the following example:

```
probe some_unrelated_probe = ... { ... }

probe myprobe = syscall.read { ... }

probe myprobe.test = some_unrelated_probe { ... }

probe myprobe.* { ... }

probe myprobe.ret* { ... }
```

Here, `return` would be a valid suffix for `myprobe`. The wildcard `myprobe.*` matches the ordinary alias `myprobe.test`, and hence the suffix expansion `myprobe.return` is not included. Conversely, `myprobe.ret*` does not match any ordinary aliases, so the suffix `myprobe.return` is included as an expansion.

3.3 Variables

Identifiers for variables and functions are alphanumeric sequences, 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 remain instantiated as long as the SystemTap session. There is one namespace for all global variables, regardless of the script file in which they are found. Because of possible concurrency limits, such as multiple probe handlers, 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. Global variables which are written but never read will be displayed automatically at session shutdown. 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>]
```

The scope of a global variable may be limited to a tapset or user script file using private keyword. The global keyword is optional when defining a private global variable. Following declaration marks var1 and var2 private globals.

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

3.3.1 Unused variables

The SystemTap translator removes unused variables. Global variable that are never written or read are discarded. Every local variables where the variable is only written but never read are also discarded. This optimization prunes unused variables defined in the probe aliases, but never used in the probe handler. If desired, this optimization can disabled with the `-u` option.

3.4 Auxiliary functions

General syntax:

```
function <name>[:<type>] ( <arg1>[:<type>], ... )[:<priority>] { <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 and Section 5.2. 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.

Functions may be marked private using the private keyword to limit their scope to the tapset or user script file they are defined in. An example definition of a private function follows:

```
private function three:long () { return 3 }
```

Functions terminating without reaching an explicit return statement will return an implicit 0 or "", determined by type inference.

Functions may be overloaded during both runtime and compile time.

Runtime overloading allows the executed function to be selected while the module is running based on runtime conditions and is achieved using the "next" statement in script functions and `STAP_NEXT` macro for embedded-C functions. For example,

```
function f() { if (condition) next; print("first function") }
function f() %{ STAP_NEXT; print("second function") %}
function f() { print("third function") }
```

During a functioncall f(), the execution will transfer to the third function if condition evaluates to true and print "third function". Note that the second function is unconditionally nexted.

Parameter overloading allows the function to be executed to be selected at compile time based on the number of arguments provided to the functioncall. For example,

```
function g() { print("first function") }
function g(x) { print("second function") }
g() -> "first function"
g(1) -> "second function"
```

Note that runtime overloading does not occur in the above example, as exactly one function will be resolved for the functioncall. The use of a next statement inside a function while no more overloads remain will trigger a runtime exception Runtime overloading will only occur if the functions have the same arity, functions with the same name but different number of parameters are completely unrelated.

Execution order is determined by a priority value which may be specified. If no explicit priority is specified, user script functions are given a higher priority than library functions. User script functions and library functions are assigned a default priority value of 0 and 1 respectively. Functions with the same priority are executed in declaration order. For example,

```
function f():3 { if (condition) next; print("first function") }
function f():1 { if (condition) next; print("second function") }
function f():2 { print("third function") }
```

3.5 Embedded C

SystemTap supports a *guru mode* where script safety features such as code and data memory reference protection are removed. Guru mode is set by passing the **-g** option to the stap command. When in guru mode, the translator accepts C code enclosed between "%{" and "%}" markers in the top level of the script file. The embedded C code is transcribed verbatim, without analysis, in sequence, into the top level of the generated C code. Thus, guru mode may be useful for adding #include instructions at the top level of the generated module, or providing auxiliary definitions for use by other embedded code.

When in guru mode, embedded C code blocks are also allowed as the body of a SystemTap function (as described in Section 3.6), and in place of any SystemTap expression. In the latter case, the code block must contain a valid expression according to C syntax.

Here is an example of the various permitted methods of embedded C code inclusion:

```
%{
#include <linux/in.h>
#include <linux/ip.h>
%} /* <-- top level */

/* Reads the char value stored at a given address: */
function __read_char:long(addr:long) %{ /* pure */
         STAP_RETURN(kderef(sizeof(char), STAP_ARG_addr));
         CATCH_DEREF_FAULT ();
%} /* <-- function body */

/* Determines whether an IP packet is TCP, based on the iphdr: */
function is_tcp_packet:long(iphdr) {
         protocol = @cast(iphdr, "iphdr")->protocol
         return (protocol == %{ IPPROTO_TCP %}) /* <-- expression */
}
```

3.6 Embedded C functions

General syntax:

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

Embedded C 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 C parser.

There are a number of undocumented but complex safety constraints on concurrency, resource consumption and runtime limits that are applied to code written in the SystemTap language. These constraints are not

applied to embedded C code, so use embedded C code with extreme caution. Be especially careful when dereferencing pointers. Use the kread() macro to dereference any pointers that could potentially be invalid or dangerous. If you are unsure, err on the side of caution and use kread(). The kread() macro is one of the safety mechanisms used in code generated by embedded C. It protects against pointer accesses that could crash the system.

For example, to access the pointer chain `name = skb->dev->name` in embedded C, use the following code.

```
struct net_device *dev;
char *name;
dev = kread(&(skb->dev));
name = kread(&(dev->name));
```

The memory locations reserved for input and output values are provided to a function using macros named `STAP_ARG_foo` (for arguments named `foo`) and `STAP_RETVALUE`. Errors may be signalled with `STAP_ERROR`. Output may be written with `STAP_PRINTF`. The function may return early with `STAP_RETURN`. Here are some examples:

```
function integer_ops:long (val) %{
  STAP_PRINTF("%d\n", STAP_ARG_val);
  STAP_RETVALUE = STAP_ARG_val + 1;
  if (STAP_RETVALUE == 4)
      STAP_ERROR("wrong guess: %d", (int) STAP_RETVALUE);
  if (STAP_RETVALUE == 3)
      STAP_RETURN(0);
  STAP_RETVALUE ++;
%}
function string_ops:string (val) %{
  strlcpy (STAP_RETVALUE, STAP_ARG_val, MAXSTRINGLEN);
  strlcat (STAP_RETVALUE, "one", MAXSTRINGLEN);
  if (strcmp (STAP_RETVALUE, "three-two-one"))
      STAP_RETURN("parameter should be three-two-");
%}
function no_ops () %{
    STAP_RETURN(); /* function inferred with no return value */
%}
```

The function argument and return value types should be stated if the translator cannot infer them from usage. The translator does not analyze the embedded C code within the function.

You should examine C code generated for ordinary script language functions to write compatible embedded-C. Usually, all SystemTap functions and probes run with interrupts disabled, thus you cannot call functions that might sleep within the embedded C.

3.7 Embedded C pragma comments

Embedded C blocks may contain various markers to assert optimization and safety properties.

- `/* pure */` means that the C code has no side effects and may be elided entirely if its value is not used by script code.
- `/* stable */` means that the C code always has the same value (in any given probe handler invocation), so repeated calls may be automatically replaced by memoized values. Such functions must take no parameters, and also be `/* pure */`.
- `/* unprivileged */` means that the C code is so safe that even unprivileged users are permitted to use it. (This is useful, in particular, to define an embedded-C function inside a tapset that may be used by unprivileged code.)
- `/* myproc-unprivileged */` means that the C code is so safe that even unprivileged users are permitted to use it, provided that the target of the current probe is within the user's own process.
- `/* guru */` means that the C code is so unsafe that a systemtap user must specify `-g` (guru mode) to use this, even if the C code is being exported from a tapset.
- `/* unmangled */`, used in an embedded-C function, means that the legacy (pre-1.8) argument access syntax should be made available inside the function. Hence, in addition to `STAP_ARG_foo` and `STAP_RETVALUE` one can use `THIS->foo` and `THIS->__retvalue` respectively inside the function. This is useful for quickly migrating code written for SystemTap version 1.7 and earlier.
- `/* unmodified-fnargs */` in an embedded-C function, means that the function arguments are not modified inside the function body.
- `/* string */` in embedded-C expressions only, means that the expression has `const char *` type and should be treated as a string value, instead of the default long numeric.

3.8 Accessing script level global variables

Script level global variables may be accessed in embedded-C functions and blocks. To read or write the global variable **var**, the `/* pragma:read:var */` or `/* pragma:write:var */` marker must be first placed in the embedded-C function or block. This provides the macros `STAP_GLOBAL_GET_*` and `STAP_GLOBAL_SET_*` macros to allow reading and writing, respectively. For example:

```
global var
global var2[100]
function increment() %{
    /* pragma:read:var */ /* pragma:write:var */
    /* pragma:read:var2 */ /* pragma:write:var2 */
    STAP_GLOBAL_SET_var(STAP_GLOBAL_GET_var()+1); //var++
    STAP_GLOBAL_SET_var2(1, 1, STAP_GLOBAL_GET_var2(1, 1)+1); //var2[1,1]++
%}
```

Variables may be read and set in both embedded-C functions and expressions. Strings returned from embedded-C code are decayed to pointers. Variables must also be assigned at script level to allow for type inference. Map assignment does not return the value written, so chaining does not work.

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")
kernel.function("foo").return
module{"ext3"}.function("ext3_*")
kernel.function("no_such_function") ?
syscall.*
end
timer.ms(5000)
```

Probes may be broadly classified into *synchronous* or *asynchronous*. A synchronous event 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

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

4.1.2 Suffixes

Suffixes further qualify the point to probe, such as **.return** for the exit point of a probed function. The absence of a suffix implies the function entry point.

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.*
kernel.function("sys_*)
```

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") ?
```

4.1.5 Brace expansion

Brace expansion is a mechanism which allows a list of probe points to be generated. It is very similar to shell expansion. A component may be surrounded by a pair of curly braces to indicate that the comma-separated sequence of one or more subcomponents will each constitute a new probe point. The braces may be arbitrarily nested. The ordering of expanded results is based on product order.

The question mark (?), exclamation mark (!) indicators and probe point conditions may not be placed in any expansions that are before the last component.

The following is an example of brace expansion.

```
syscall.{write,read}
# Expands to
syscall.write, syscall.read

{kernel,module("nfs")}.function("nfs*")!
# Expands to
kernel.function("nfs*")!, module("nfs").function("nfs*")!
```

4.2 Built-in probe point types (DWARF probes)

This family of probe points uses symbolic debugging information for the target kernel or module, as may be found in executables that have not been stripped, or in the separate **debuginfo** packages. They allow logical placement of probes into the execution path of the target 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 or some combination of these.

Here is a list of probe point specifications currently supported:

```
kernel.function(PATTERN)
kernel.function(PATTERN).call
kernel.function(PATTERN).return
kernel.function(PATTERN).return.maxactive(VALUE)
kernel.function(PATTERN).inline
kernel.function(PATTERN).label(LPATTERN)
module(MPATTERN).function(PATTERN)
module(MPATTERN).function(PATTERN).call
module(MPATTERN).function(PATTERN).return.maxactive(VALUE)
module(MPATTERN).function(PATTERN).inline
kernel.statement(PATTERN)
kernel.statement(ADDRESS).absolute
```

```
module(MPATTERN).statement(PATTERN)
```

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 entry parameters are also available, though the function may have changed their values. Return probes may be further qualified with **.maxactive**, which specifies how many instances of the specified function can be probed simultaneously. You can leave off **.maxactive** in most cases, as the default (**KRETACTIVE**) should be sufficient. However, if you notice an excessive number of skipped probes, try setting **.maxactive** to incrementally higher values to see if the number of skipped probes decreases.

The **.inline** modifier for **.function** filters the results to include only instances of inlined functions. The **.call** modifier selects the opposite subset. The **.exported** modifier filters the results to include only exported 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 above probe descriptions, MPATTERN stands for a string literal that identifies the loaded kernel module of interest and LPATTERN stands for a source program label. Both MPATTERN and LPATTERN may include asterisk (*), square brackets "[]", 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 the path to the source file containing the function, which may include a wildcard pattern, such as mm/slab*. In most cases, the path should be relative to the top of the linux source directory, although an absolute path may be necessary for some kernels. If a relative pathname doesn't work, try absolute.
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 ":" or "+". The line number is assumed to be an absolute line number if preceded by a ":", or relative to the entry of the function if preceded by a "+". All the lines in the function can be matched with ":*". A range of lines x through y can be matched with ":x-y".

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 dollar sign within the scripts. In addition, a special syntax allows limited traversal of structures, pointers, arrays, taking the address of a variable or pretty printing a whole structure.

`$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 by the `kernel_string()` or `user_string()` functions.

`@var("varname")` is an alternative syntax for `$varname`. It can also be used to access global variables in a particular compile unit (CU). `@var("varname@src/file.c")` refers to the global (either file local or

external) variable varname defined when the file src/file.c was compiled. The CU in which the variable is resolved is the first CU in the module of the probe point which matches the given file name at the end and has the shortest file name path (e.g. given `@var("foo@bar/baz.c")` and CUs with file name paths `src/sub/module/bar/baz.c` and `src/bar/baz.c` the second CU will be chosen to resolve `foo`).

The notation `@var("varname", "/path/to/exe-or-so)` is also supported to explicitly specify an executable or library file path in which the global or top-level static variable resides.

`$var->field` or `@var("var@file.c")->field` traverses a structure's field. The indirection operator may be repeated to follow additional levels of pointers.

`$var[N]` or `@var("var@file.c")[N]` indexes into an array. The index is given with a literal number.

`&$var` or `&@var("var@file.c")` provides the address of a variable as a long. It can also be used in combination with field access or array indexing to provide the address of a particular field or an element in an array with `&var->field`, `&@var("var@file.c")[N]` or a combination of those accessors.

Using a single `$` or a double `$$` suffix provides a swallow or deep string representation of the variable data type. Using a single `$`, as in `$var$`, will provide a string that only includes the values of all basic type values of fields of the variable structure type but not any nested complex type values (which will be represented with {...}). Using a double `$$`, as in `@var("var")$$` will provide a string that also includes all values of nested data types.

`$$vars` expands to a character string that is equivalent to `sprintf("parm1=%x ... parmN=%x var1=%x ... varN=%x", $parm1, ..., $parmN, $var1, ..., $varN)`

`$$locals` expands to a character string that is equivalent to `sprintf("var1=%x ... varN=%x", $var1, ..., $varN)`

`$$parms` expands to a character string that is equivalent to `sprintf("parm1=%x ... parmN=%x", $parm1, ..., $parmN)`

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

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/time.c"
# file that span line 240:
kernel.function("*@kernel/time.c:240")
```

```
# Refers to all functions in the ext3 module:
module("ext3").function("*")
```

4.2.2 kernel.statement, module().statement

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

General syntax:

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

Example:

```
# Refers to the statement at line 296 within the
# kernel/time.c file:
kernel.statement("*@kernel/time.c:296")
# Refers to the statement at line bio_init+3 within the fs/bio.c file:
kernel.statement("bio_init@fs/bio.c+3")
```

4.3 Function 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. The entry parameters are also accessible in the context of the return probe, though their values may have been changed by the function. Inline functions do not have an identifiable return point, so `.return` is not supported on `.inline` probes.

4.4 DWARF-less probing

In the absence of debugging information, you can still use the *kprobe* family of probes to examine the entry and exit points of kernel and module functions. You cannot look up the arguments or local variables of a function using these probes. However, you can access the parameters by following this procedure:

When you're stopped at the entry to a function, you can refer to the function's arguments by number. For example, when probing the function declared:

```
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t
count)
```

You can obtain the values of `fd`, `buf`, and `count`, respectively, as `uint_arg(1)`, `pointer_arg(2)`, and `ulong_arg(3)`. In this case, your probe code must first call `asmlinkage()`, because on some architectures the asmlinkage attribute affects how the function's arguments are passed.

When you're in a return probe, `$return` isn't supported without DWARF, but you can call `returnval()` to get the value of the register in which the function value is typically returned, or call `returnstr()` to get a string version of that value.

And at any code probepoint, you can call `register("regname")` to get the value of the specified CPU register when the probe point was hit. `u_register("regname")` is like `register("regname")`, but interprets the value as an unsigned integer.

SystemTap supports the following constructs:

```
kprobe.function(FUNCTION)
kprobe.function(FUNCTION).return
kprobe.module(NAME).function(FUNCTION)
kprobe.module(NAME).function(FUNCTION).return
kprobe.statement(ADDRESS).absolute
```

Use **.function** probes for kernel functions and **.module** probes for probing functions of a specified module. If you do not know the absolute address of a kernel or module function, use **.statement** probes. Do not use wildcards in *FUNCTION* and *MODULE* names. Wildcards cause the probe to not register. Also, statement probes are available only in guru mode.

4.5 Userspace probing

Support for userspace probing is supported on kernels that are configured to include the utrace or uprobes extensions.

4.5.1 Begin/end variants

Constructs:

```
process.begin
process("PATH").begin
process(PID).begin

process.thread.begin
process("PATH").thread.begin
process(PID).thread.begin

process.end
process("PATH").end
process(PID).end

process.thread.end
process("PATH").thread.end
process(PID).thread.end
```

The `.begin` variant is called when a new process described by `PID` or `PATH` is created. If no `PID` or `PATH` argument is specified (for example `process.begin`), the probe flags any new process being spawned.

The `.thread.begin` variant is called when a new thread described by `PID` or `PATH` is created.

The `.end` variant is called when a process described by `PID` or `PATH` dies.

The `.thread.end` variant is called when a thread described by `PID` or `PATH` dies.

4.5.2 Syscall variants

Constructs:

```
process.syscall
process("PATH").syscall
process(PID).syscall

process.syscall.return
process("PATH").syscall.return
process(PID).syscall.return
```

The `.syscall` variant is called when a thread described by `PID` or `PATH` makes a system call. The system call number is available in the `$syscall` context variable. The first six arguments of the system call are available in the `$argN` parameter, for example `$arg1`, `$arg2`, and so on.

The `.syscall.return` variant is called when a thread described by `PID` or `PATH` returns from a system call. The system call number is available in the `$syscall` context variable. The return value of the system call is available in the `$return` context variable.

4.5.3 Function/statement variants

Constructs:

```
process("PATH").function("NAME")
process("PATH").statement("*@FILE.c:123")
process("PATH").function("*").return
process("PATH").function("myfun").label("foo")
```

Full symbolic source-level probes in userspace programs and shared libraries are supported. These are exactly analogous to the symbolic DWARF-based kernel or module probes described previously and expose similar contextual `$-variables`. See Section 4.2 for more information

Here is an example of prototype symbolic userspace probing support:

```
# stap -e 'probe process("ls").function("*").call {
           log (probefunc()." ".$$parms)
           }' \
       -c 'ls -l'
```

To run, this script requires debugging information for the named program and utrace support in the kernel. If you see a "pass 4a-time" build failure, check that your kernel supports utrace.

4.5.4 Absolute variant

A non-symbolic probe point such as `process(PID).statement(ADDRESS).absolute` is analogous to `kernel.statement(ADDRESS).absolute` in that both use raw, unverified virtual addresses and provide no `$variables`. The target `PID` parameter must identify a running process and `ADDRESS` must identify a valid instruction address. All threads of the listed process will be probed. This is a guru mode probe.

4.5.5 Process probe paths

For all process probes, `PATH` names refer to executables that are searched the same way that shells do: the explicit path specified if the path name begins with a slash (/) character sequence; otherwise `$PATH` is searched. For example, the following probe syntax:

```
probe process("ls").syscall {}
probe process("./a.out").syscall {}
```

works the same as:

```
probe process("/bin/ls").syscall {}
probe process("/my/directory/a.out").syscall {}
```

If a process probe is specified without a `PID` or `PATH` parameter, all user threads are probed. However, if systemtap is invoked in target process mode, process probes are restricted to the process hierarchy associated with the target process. If stap is running in `--unprivileged` mode, only processes owned by the current user are selected.

4.5.6 Target process mode

Target process mode (invoked with `stap -c CMD` or `-x PID`) implicitly restricts all `process.*` probes to the given child process. It does not affect `kernel.*` or other probe types. The `CMD` string is normally run directly, rather than from a "`/bin/sh -c`" sub-shell, since utrace and uprobe probes receive a fairly "clean" event stream. If meta-characters such as redirection operators are present in `CMD`, "`/bin/sh -c CMD`" is still used, and utrace and uprobe probes will receive events from the shell. For example:

```
% stap -e 'probe process.syscall, process.end {
           printf("%s %d %s\n", execname(), pid(), pp())}' \
       -c ls
```

Here is the output from this command:

```
ls 2323 process.syscall
ls 2323 process.syscall
ls 2323 process.end
```

If `PATH` names a shared library, all processes that map that shared library can be probed. If dwarf debugging information is installed, try using a command with this syntax:

```
probe process("/lib64/libc-2.8.so").function("....") { ... }
```

This command probes all threads that call into that library. Typing "`stap -c CMD`" or "`stap -x PID`" restricts this to the target command and descendants only. You can use `$$vars` and others. You can provide the location of debug information to the stap command with the `-d DIRECTORY` option. To qualify a probe point to a location in a library required by a particular process try using a command with this syntax:

```
probe process("...").library("...").function("....") { ... }
```

The library name may use wildcards.

The first syntax in the following will probe the functions in the program linkage table of a particular process. The second syntax will also add the program linkage tables of libraries required by that process. .plt("...") can be specified to match particular plt entries.

```
probe process("...").plt { ... }
probe process("...").plt process("...").library("...").plt { ... }
```

4.5.7 Instruction probes

Constructs:

```
process("PATH").insn
process(PID).insn

process("PATH").insn.block
process(PID).insn.block
```

The `process().insn` and `process().insn.block` probes inspect the process after each instruction or block of instructions is executed. These probes are not implemented on all architectures. If they are not implemented on your system, you will receive an error message when the script starts.

The `.insn` probe is called for every single-stepped instruction of the process described by `PID` or `PATH`.

The `.insn.block` probe is called for every block-stepped instruction of the process described by `PID` or `PATH`.

To count the total number of instructions that a process executes, type a command similar to:

```
$ stap -e 'global steps; probe process("/bin/ls").insn {steps++}
           probe end {printf("Total instructions: %d\n", steps);}' \
       -c /bin/ls
```

Using this feature will significantly slow process execution.

4.5.8 Static userspace probing

You can probe symbolic static instrumentation compiled into programs and shared libraries with the following syntax:

```
process("PATH").mark("LABEL")
```

The `.mark` variant is called from a static probe defined in the application by `STAP_PROBE1(handle,LABEL,arg1)`. `STAP_PROBE1` is defined in the sdt.h file. The parameters are:

Parameter	Definition
`handle`	the application handle
`LABEL`	corresponds to the `.mark` argument
`arg1`	the argument

Use `STAP_PROBE1` for probes with one argument. Use `STAP_PROBE2` for probes with 2 arguments, and so on. The arguments of the probe are available in the context variables `$arg1`, `$arg2`, and so on.

As an alternative to the `STAP_PROBE` macros, you can use the dtrace script to create custom macros. The sdt.h file also provides dtrace compatible markers through `DTRACE_PROBE` and an associated python `dtrace` script. You can use these in builds based on dtrace that need dtrace -h or -G functionality.

4.6 Java probes

Support for probing Java methods is available using Byteman as a backend. Byteman is an instrumentation tool from the JBoss project which systemtap can use to monitor invocations for a specific method or line in a Java program.

Systemtap does so by generating a Byteman script listing the probes to instrument and then invoking the Byteman `bminstall` utility. A custom option `-D OPTION` (see the Byteman documentation for more details) can be passed to bminstall by invoking systemtap with option `-J OPTION`. The systemtap option `-j` is also provided as a shorthand for `-J org.jboss.byteman.compile.to.bytecode`.

This Java instrumentation support is currently a prototype feature with major limitations: Java probes attach only to one Java process at a time; other Java processes beyond the first one to be observed are ignored. Moreover, Java probing currently does not work across users; the stap script must run (with appropriate permissions) under the same user as the Java process being probed. (Thus a stap script under root currently cannot probe Java methods in a non-root-user Java process.)

There are four probe point variants supported by the translator:

```
java("PNAME").class("CLASSNAME").method("PATTERN")
java("PNAME").class("CLASSNAME").method("PATTERN").return
java(PID).class("CLASSNAME").method("PATTERN")
java(PID).class("CLASSNAME").method("PATTERN").return
```

The first two probe points refer to Java processes by the name of the Java process. The PATTERN parameter specifies the signature of the Java method to probe. The signature must consist of the exact name of the method, followed by a bracketed list of the types of the arguments, for instance `myMethod(int,double,Foo)`. Wildcards are not supported.

The probe can be set to trigger at a specific line within the method by appending a line number with colon, just as in other types of probes: `myMethod(int,double,Foo):245`.

The CLASSNAME parameter identifies the Java class the method belongs to, either with or without the package qualification. By default, the probe only triggers on descendants of the class that do not override the method definition of the original class. However, CLASSNAME can take an optional caret prefix, as in `class("^org.my.MyClass")`, which specifies that the probe should also trigger on all descendants of MyClass that override the original method. For instance, every method with signature foo(int) in program org.my.MyApp can be probed at once using

```
java("org.my.MyApp").class("^java.lang.Object").method("foo(int)")
```

The last two probe points work analogously, but refer to Java processes by PID. (PIDs for already running processes can be obtained using the `jps` utility.)

Context variables defined within java probes include `$provider` (which identifies the class providing the definition of the triggered method) and `$name` (which gives the signature of the method). Arguments to the method can be accessed using context variables `$arg1$` through `$arg10`, for up to the first 10 arguments of a method.

4.7 PROCFS probes

These probe points allow procfs pseudo-files in `/proc/systemtap/`*`MODNAME`* to be created, read and written. Specify the name of the systemtap module as *`MODNAME`*. There are four probe point variants supported by the translator:

```
procfs("PATH").read
procfs("PATH").write
procfs.read
procfs.write
```

`PATH` is the file name to be created, relative to `/proc/systemtap/MODNAME`. If no `PATH` is specified (as in the last two variants in the previous list), `PATH` defaults to "command".

When a user reads `/proc/systemtap/MODNAME/PATH`, the corresponding procfs read probe is triggered. Assign the string data to be read to a variable named `$value`, as follows:

```
procfs("PATH").read { $value = "100\n" }
```

When a user writes into `/proc/systemtap/MODNAME/PATH`, the corresponding procfs write probe is triggered. The data the user wrote is available in the string variable named `$value`, as follows:

```
procfs("PATH").write { printf("User wrote: %s", $value) }
```

4.8 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` prefix which identifies the source of the symbol table used for finding markers. The suffix names the marker itself: `mark.("MARK")`. The marker name string, which can contain wildcard characters, is matched against the names given to the marker macros when the kernel or module is compiled. Optionally, you can specify `format("FORMAT")`. Specifying the marker format string allows differentiation between two markers with the same name but different marker format strings.

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.

The marker format string associated with a marker is available in `$format`. The marker name string is available in `$name`.

Here are the marker probe constructs:

```
kernel.mark("MARK")
kernel.mark("MARK").format("FORMAT")
```

For more information about marker probes, see `http://sourceware.org/systemtap/wiki/UsingMarkers`.

4.9 Tracepoints

This family of probe points hooks to static probing tracepoints inserted into the kernel or kernel modules. As with marker probes, these tracepoints are special macro calls inserted by kernel developers to make probing faster and more reliable than with DWARF-based probes. DWARF debugging information is not required to probe tracepoints. Tracepoints have more strongly-typed parameters than marker probes.

Tracepoint probes begin with `kernel`. The next part names the tracepoint itself: `trace("name")`. The tracepoint `name` string, which can contain wildcard characters, is matched against the names defined by the kernel developers in the tracepoint header files.

The handler associated with a tracepoint-based probe can read the optional parameters specified at the macro call site. These parameters are named according to the declaration by the tracepoint author. For example, the tracepoint probe `kernel.trace("sched_switch")` provides the parameters `$rq`, `$prev`, and `$next`. If the parameter is a complex type such as a struct pointer, then a script can access fields with the same syntax as DWARF `$target` variables. Tracepoint parameters cannot be modified; however, in guru mode a script can modify fields of parameters.

The name of the tracepoint is available in `$$name`, and a string of `name=value` pairs for all parameters of the tracepoint is available in `$$vars` or `$$parms`.

4.10 Syscall probes

The `syscall.*` aliases define several hundred probes. They use the following syntax:

```
syscall.NAME
syscall.NAME.return
```

Generally, two probes are defined for each normal system call as listed in the syscalls(2) manual page: one for entry and one for return. System calls that never return do not have a corresponding `.return` probe.

Each probe alias defines a variety of variables. Look at the tapset source code to find the most reliable source of variable definitions. Generally, each variable listed in the standard manual page is available as a script-level variable. For example, `syscall.open` exposes file name, flags, and mode. In addition, a standard suite of variables is available at most aliases, as follows:

- `argstr`: A pretty-printed form of the entire argument list, without parentheses.
- `name`: The name of the system call.
- `retstr`: For return probes, a pretty-printed form of the system call result.

Not all probe aliases obey all of these general guidelines. Please report exceptions that you encounter as a bug.

4.11 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 on 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 (s or sec), milliseconds (ms or msec), microseconds (us or usec), nanoseconds (ns or 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 greater 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.tick
```

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

It is recommended to use the tapset probe `timer.profile` rather than `timer.profile.tick`. This probe point behaves identically to `timer.profile.tick` when the underlying functionality is available, and falls back to using `perf.sw.cpu_clock` on some recent kernels which lack the corresponding profile timer facility.

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)
```

4.12 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.

4.12.1 begin

The `begin` probe is the start of the SystemTap session. All `begin` probe handlers are run during the startup of the session.

4.12.2 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 a SystemTap `exit` function call, or an interruption from the user. In the case of an shutdown triggered by error, `end` probes are not run.

4.12.3 error

The *error* probe point is similar to the end probe, except the probe handler runs when the session ends if an error occurred. In this case, an `end` probe is skipped, but each `error` probe is still attempted. You can use an `error` probe to clean up or perform a final action on script termination.

Here is a simple example:

```
probe error { println ("Oops, errors occurred. Here's a report anyway.")
              foreach (coin in mint) { println (coin) } }
```

4.12.4 begin, end, and error probe sequence

`begin`, `end`, and `error` probes can be 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.

4.12.5 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.

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 indices and values. There are no implicit type conversions between strings and numbers. Inconsistent type-related use of an identifier signals an error.

5.2.1 Literals

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

5.2.2 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} but below 2^{64}.

5.2.3 Strings

Strings are enclosed in quotation marks ("string"), and pass through standard C escape codes with backslashes. A string literal may be split into several pieces, which are glued together, as follows.

```
str1 = "foo" "bar"
  /* --> becomes "foobar" */

str2 = "a good way to do a multi-line\n"
       "string literal"
  /* --> becomes "a good way to do a multi-line\nstring literal" */
```

```
str3 = "also a good way to " @1 " splice command line args"
  /* --> becomes "also a good way to foo splice command line args",
     assuming @1 is given as foo on the command line */
```

Observe that script arguments can also be glued into a string literal.

Strings are limited in length to MAXSTRINGLEN. For more information about this and other limits, see Section 1.6.

5.2.4 Associative arrays

See Section 7

5.2.5 Statistics

See Section 8

5.3 Semicolons

The semicolon is the null statement, or do nothing statement. 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. If the parser detects division by zero or an overflow, it generates an error. The following subsections list these operators.

5.6.1 Binary numeric operators

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

5.6.2 Binary string operators

. (string concatenation)

5.6.3 Numeric assignment operators

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

5.6.4 String assignment operators

```
= .=
```

5.6.5 Unary numeric operators

```
+ - !  ~ ++ --
```

5.6.6 Numeric & string comparison, regular expression matching operators

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

The `=~` and `!~` operators perform regular expression matching. The second operand must be a string literal containing a syntactically valid regular expression. The `=~` operator returns `1` on a successful match and `0` on a failed match. The `!~` operator returns `1` on a failed match. The regular expression syntax supports most of the features of POSIX Extended Regular Expressions, except for subexpression reuse (`\1`) functionality. After a successful match, the matched substring and subexpressions can be extracted using the `matched` tapset function. The `ngroups` tapset function returns the number of subexpressions in the last successfully matched regular expression.

5.6.7 Ternary operator

```
cond ?  exp1 :  exp2
```

5.6.8 Grouping operator

```
( exp )
```

5.6.9 Function call

General syntax:

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

5.6.10 $ptr->member

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

5.6.11 Pointer typecasting

Typecasting is supported using the `@cast()` operator. A script can define a pointer type for a *long* value, then access type members using the same syntax as with `$target` variables. After a pointer is saved into a script integer variable, the translator loses the necessary type information to access members from that pointer. The `@cast()` operator tells the translator how to read a pointer.

The following statement interprets `p` as a pointer to a struct or union named `type_name` and dereferences the `member` value:

```
@cast(p, "type_name"[, "module"])->member
```

The optional `module` parameter tells the translator where to look for information about that type. You can specify multiple modules as a list with colon (`:`) separators. If you do not specify the module parameter, the translator defaults to either the probe module for dwarf probes or to *kernel* for functions and all other probe types.

The following statement retrieves the parent PID from a kernel task_struct:

```
@cast(pointer, "task_struct", "kernel")->parent->tgid
```

The translator can create its own module with type information from a header surrounded by angle brackets (`< >`) if normal debugging information is not available. For kernel headers, prefix it with `kernel` to use the appropriate build system. All other headers are built with default GCC parameters into a user module. The following statements are examples.

```
@cast(tv, "timeval", "<sys/time.h>")->tv_sec
@cast(task, "task_struct", "kernel<linux/sched.h>")->tgid
```

In guru mode, the translator allows scripts to assign new values to members of typecasted pointers.

Typecasting is also useful in the case of `void*` members whose type might be determinable at run time.

```
probe foo {
   if ($var->type == 1) {
      value = @cast($var->data, "type1")->bar
   } else {
      value = @cast($var->data, "type2")->baz
   }
   print(value)
}
```

5.6.12 <value> in <array_name>

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

5.6.13 [<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 5.2.2. For information about strings, see Section 5.2.3.

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 literal pasting

Use `$1 ... $<NN>` for pasting the entire argument string into the input stream, which will be further lexically tokenized.

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

Use `@1 ... @<NN>` for casting an entire argument as a string literal.

5.7.3 Examples

For example, if the following script named example.stp

```
probe begin { printf("%d, %s\n", $1, @2) }
```

is invoked as follows

```
# stap example.stp '5+5' mystring
```

then 5+5 is substituted for $1 and "mystring" for @2. The output will be

```
10, mystring
```

5.8 Conditional compilation

5.8.1 Conditions

One of the steps of parsing is a simple preprocessing stage. The preprocessor supports conditionals with a general form similar to the ternary operator (Section 5.6.7).

```
%( 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 available target variables

The predicate @defined() is available for testing whether a particular $variable/expression is resolvable at translation time. The following is an example of its use:

```
probe foo { if (@defined($bar)) log ("$bar is available here") }
```

5.8.3 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".

5.8.4 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. The currently supported architecture strings are i386, i686, x86_64, ia64, s390, and powerpc.

5.8.5 Conditions based on privilege level: systemtap_privilege

If the first part of the conditional expression is the identifier `systemtap_privilege` which refers to the privilege level the systemtap script is being compiled with, 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. The possible privilege strings to consider are `"stapusr"` for unprivileged scripts, and `"stapsys"` or `"stapdev"` for privileged scripts. (In general, to test for a privileged script it is best to use `!= "stapusr"`.)

This condition can be used to write scripts that can be run in both privileged and unprivileged modes, with additional functionality made available in the privileged case.

5.8.6 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") {}
%)
```

The following code adapts to the presence of a kernel CONFIG option.

```
%( CONFIG_UTRACE == "y" %?
    probe process.syscall {}
%)
```

5.9 Preprocessor macros

This feature lets scripts eliminate some types of repetition.

5.9.1 Local macros

The preprocessor also supports a simple macro facility.

Macros taking zero or more arguments are defined using the following construct:

```
@define NAME %( BODY %)
@define NAME(PARAM_1, PARAM_2, ...) %( BODY %)
```

Macro arguments are referred to in the body by prefixing the argument name with an `@` symbol. Likewise, once defined, macros are invoked by prefixing the macro name with an `@` symbol:

```
@define foo %( x %)
@define add(a,b) %( ((@a)+(@b)) %)

   @foo = @add(2,2)
```

Macro expansion is currently performed in a separate pass before conditional compilation. Therefore, both TRUE- and FALSE-tokens in conditional expressions will be macroexpanded regardless of how the condition is evaluated. This can sometimes lead to errors:

```
// The following results in a conflict:
%( CONFIG_UTRACE == "y" %?
    @define foo %( process.syscall %)
%:
    @define foo %( **ERROR** %)
```

```
%)

// The following works properly as expected:
@define foo %(
  %( CONFIG_UTRACE == "y" %? process.syscall %: **ERROR** %)
%)
```

The first example is incorrect because both `@define`s are evaluated in a pass prior to the conditional being evaluated.

5.9.2 Library macros

Normally, a macro definition is local to the file it occurs in. Thus, defining a macro in a tapset does not make it available to the user of the tapset.

Publically available library macros can be defined by including `.stpm` files on the tapset search path. These files may only contain `@define` constructs, which become visible across all tapsets and user scripts.

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.

6.1 break and continue

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

6.2 try/catch

Use `try/catch` to handle most kinds of run-time errors within the script instead of aborting the probe handler in progress. The semantics are similar to C++ in that try/catch blocks may be nested. The error string may be captured by optionally naming a variable which is to receive it.

```
try {
   /* do something */
   /* trigger error like kread(0), or divide by zero, or error("foo") */
} catch (msg) { /* omit (msg) entirely if not interested */
   /* println("caught error ", msg) */
   /* handle error */
}
/* execution continues */
```

6.3 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
```

6.4 EXP (expression)

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

6.5 for

General syntax:

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

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.

6.6 foreach

General syntax:

```
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 as the first example, 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
```

You can combine the first and second syntax to capture both the full tuple and the keys at the same time as follows.

```
foreach (VAR = [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
```

6.7 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.

6.8 next

The `next` statement returns immediately from the enclosing probe handler. When used in functions, the execution will be immediately transferred to the next overloaded function.

6.9 ; (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.

6.10 return

General syntax:

```
return EXP
```

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

6.11 { } (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.

6.12 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 up to nine index 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

Array elements may be set to a number, a string, or an aggregate. The type must be consistent throughout the use of the array. The first assignment to the array defines the type of the elements. Unset array elements may be fetched and return a null value (zero or empty string) as appropriate, but they are not seen by a membership test.

7.3 Array capacity

Array sizes can be specified explicitly or allowed to default to the maximum size as defined by MAXMAPENTRIES. See Section 1.6 for details on changing MAXMAPENTRIES.

You can explicitly specify the size of an array as follows:

```
global ARRAY[<size>]
```

If you do not specify the size parameter, then the array is created to hold MAXMAPENTRIES number of elements.

7.4 Array wrapping

Arrays may be wrapped using the percentage symbol (%) causing previously entered elements to be overwritten if more elements are inserted than the array can hold. This works for both regular and statistics typed arrays.

You can mark arrays for wrapping as follows:

```
global ARRAY1%[<size>], ARRAY2%
```

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 or limited to only a few elements with the limit keyword. 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) { ... }

# Print the first 10 tuples and values in the array in decreasing sequence
```

```
foreach (v = [i,j] in foo- limit 10)
    printf("foo[%d,%s] = %d\n", i, j, v)
```

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.

For a full description of `foreach` see subsection 6.6.

7.6 Deletion

The `delete` statement can either remove a single element by index from an array or clear an entire array at once. See subsection 6.3 for details and examples.

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 associative array. For information about wrapping associative arrays with statistics elements, see section 7.4

8.1 The aggregation (<<<) operator

The aggregation operator is "<<<", and its effect is 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 as a sample 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 side of the aggregation operation.

8.3 Integer extractors

The following functions provide methods to extract information about aggregate.

8.3.1 @count(s)

This statement returns the number of samples accumulated in aggregate s.

8.3.2 @sum(s)

This statement returns the total sum of all samples in aggregate s.

8.3.3 @min(s)

This statement returns the minimum of all samples in aggregate s.

8.3.4 @max(s)

This statement returns the maximum of all samples in aggregate s.

8.3.5 @avg(s)

This statement returns the average value of all samples in aggregate s.

8.4 Histogram extractors

The following functions provide methods to extract histogram information. Printing a histogram with the print family of functions renders a histogram object as a tabular "ASCII art" bar chart.

8.4.1 @hist_linear

The statement `@hist_linear(v,L,H,W)` represents a linear histogram of aggregate `v`, where L and H represent the lower and upper end of a range of values and W represents the width (or size) of each bucket within the range. The low and high values can be negative, but the overall difference (high minus low) must be positive. The width parameter must also be positive.

In the output, a range of consecutive empty buckets may be replaced with a tilde (~) character. This can be controlled on the command line with -DHIST_ELISION=<num>, where <num> specifies how many empty buckets at the top and bottom of the range to print. The default is 2. A <num> of 0 removes all empty buckets. A negative <num> disables removal.

For example, if you specify -DHIST_ELISION=3 and the histogram has 10 consecutive empty buckets, the first 3 and last 3 empty buckets will be printed and the middle 4 empty buckets will be represented by a tilde (~).

The following is an example.

```
global reads
probe netdev.receive {
    reads <<< length
```

```
}
probe end {
    print(@hist_linear(reads, 0, 10240, 200))
}
```

This generates the following output.

```
value |-------------------------------------------------- count
    0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1650
  200 |                                                      8
  400 |                                                      0
  600 |                                                      0
      ~
 1000 |                                                      0
 1200 |                                                      0
 1400 |                                                      1
 1600 |                                                      0
 1800 |                                                      0
```

This shows that 1650 network reads were of a size between 0 and 199 bytes, 8 reads were between 200 and 399 bytes, and 1 read was between 1200 and 1399 bytes. The tilde (~) character indicates the bucket for 800 to 999 bytes was removed because it was empty. Empty buckets for 2000 bytes and larger were also removed because they were empty.

8.4.2 @hist_log

The statement `@hist_log(v)` represents a base-2 logarithmic histogram. Empty buckets are replaced with a tilde (~) character in the same way as `@hist_linear()` (see above).

The following is an example.

```
global reads
probe netdev.receive {
    reads <<< length
}
probe end {
    print(@hist_log(reads))
}
```

This generates the following output.

```
value |-------------------------------------------------- count
    8 |                                                     0
   16 |                                                     0
   32 |                                                   254
   64 |                                                     3
  128 |                                                     2
  256 |                                                     2
  512 |                                                     4
 1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 16689
 2048 |                                                     0
 4096 |                                                     0
```

8.5 Deletion

The `delete` statement (subsection 6.3) applied to an aggregate variable will reset it to the initial empty state.

9 Formatted output

9.1 print

General syntax:

```
print ()
```

This function prints a single value of any type.

9.2 printf

General syntax:

```
printf (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 format must be a literal string constant. The printf formatting directives are similar to those of C, except that they are fully checked for type by the translator.

The formatting string can contain tags that are defined as follows:

```
%[flags][width][.precision][length]specifier
```

Where `specifier` is required and defines the type and the interpretation of the value of the corresponding argument. The following table shows the details of the specifier parameter:

Table 1: printf specifier values

Specifier	**Output**	**Example**
d or i	Signed decimal	392
o	Unsigned octal	610
s	String	sample
u	Unsigned decimal	7235
x	Unsigned hexadecimal (lowercase letters)	7fa
X	Unsigned hexadecimal (uppercase letters)	7FA
p	Pointer address	0x0000000000bc614e
b	Writes a binary value as text using the computer's native byte order. The field width specifies the number of bytes to write. Valid specifications are %b, %1b, %2b, %4b and %8b. The default width is 8 (64-bits).	See below
%	A % followed by another % character will write % to stdout.	%

The tag can also contain `flags`, `width`, `.precision` and `modifiers` sub-specifiers, which are optional and follow these specifications:

Table 2: printf flag values

Flags	**Description**
- (minus sign)	Left-justify within the given field width. Right justification is the default (see `width` sub-specifier).
+ (plus sign)	Precede the result with a plus or minus sign even for positive numbers. By default, only negative numbers are preceded with a minus sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with `o`, `x` or `X` specifiers the value is preceded with `0`, `0x` or `0X` respectively for non-zero values.
0	Left-pads the number with zeroes instead of spaces, where padding is specified (see `width` sub-specifier).

Table 3: printf width values

Width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

Table 4: printf precision values

Precision	Description
.number	For integer specifiers (`d, i, o, u, x, X`): `precision` specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. When no `precision` is specified, the default is 1. If the period is specified without an explicit value for `precision`, 0 is assumed.

Binary Write Examples

The following is an example of using the binary write functions:

```
probe begin {
    for (i = 97; i < 110; 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-
```

Another example:

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

This prints:

```
00000000  ad db de c0 00 00 00 00  78 56 34 12 00 00 00 00  |........xV4.....|
00000010
```

Another example:

```
probe begin{
    printf("%1b%1b%1blo %1b%1brld\n", 72,101,108,87,111)
    exit()
}
```

This prints:

```
Hello World
```

9.3 printd

General syntax:

```
printd (delimiter:string, ...)
```

This function takes a string delimiter and two or more values of any type, then prints the values with the delimiter interposed. The delimiter must be a literal string constant.

For example:

```
printd("/", "one", "two", "three", 4, 5, 6)
```

prints:

```
one/two/three/4/5/6
```

9.4 printdln

General syntax:

```
printdln (delimiter:string, ...)
```

This function operates like `printd`, but also appends a newline.

9.5 println

General syntax:

```
println ()
```

This function prints a single value like `print`, but also appends a newline.

9.6 sprint

General syntax:

```
sprint:string ()
```

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

9.7 sprintf

General syntax:

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

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

10 Tapset-defined functions

Unlike built-in functions, tapset-defined functions are implemented in tapset scripts. These are individually documented in the in `tapset::*(3stap)`, `function::*(3stap)`, and `probe::*(3stap)` man pages, and implemented under `/usr/share/systemtap/tapset`.

11 For Further Reference

For more information, see:

- The SystemTap tutorial at `http://sourceware.org/systemtap/tutorial/`
- The SystemTap wiki at `http://sourceware.org/systemtap/wiki`
- The SystemTap documentation page at `http://sourceware.org/systemtap/documentation.html`
- From an unpacked source tarball or GIT directory, the examples in in the src/examples directory, the tapsets in the src/tapset directory, and the test scripts in the src/testsuite directory.
- The man pages for tapsets. For a list, run the command "`man -k tapset::`".
- The man pages for individual probe points. For a list, run the command "`man -k probe::`".
- The man pages for individual systemtap functions. For a list, run the command "`man -k function::`".