Red Hat Enterprise Linux 6

SystemTap Tapset Reference

For SystemTap in Red Hat Enterprise Linux 6

Last Updated: 2017-10-20

Red Hat Enterprise Linux 6 SystemTap Tapset Reference

For SystemTap in Red Hat Enterprise Linux 6

Robert Krátký
Red Hat Customer Content Services
rkratky@redhat.com

William Cohen
Red Hat Performance Tools

Don Domingo
Red Hat Customer Content Services

Red Hat, Inc.

Edited by

Jacquelynn East
Red Hat Customer Content Services

Legal Notice

Abstract

The Tapset Reference Guide describes the most common tapset definitions users can apply to SystemTap scripts. All included tapsets documented in this guide are current as of the latest upstream version of SystemTap.

Table of Contents

PREFACE

CHAPTER 1. INTRODUCTION

SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system. This assists diagnosis of a performance or functional problem. SystemTap eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.

SystemTap provides a simple command line interface and scripting language for writing instrumentation for a live, running kernel. This instrumentation uses probe points and functions provided in the *tapset* library.

Simply put, tapsets are scripts that encapsulate knowledge about a kernel subsystem into pre-written probes and functions that can be used by other scripts. Tapsets are analogous to libraries for C programs. They hide the underlying details of a kernel area while exposing the key information needed to manage and monitor that aspect of the kernel. They are typically developed by kernel subject-matter experts.

A tapset exposes the high-level data and state transitions of a subsystem. For the most part, good tapset developers assume that SystemTap users know little to nothing about the kernel subsystem's low-level details. As such, tapset developers write tapsets that help ordinary SystemTap users write meaningful and useful SystemTap scripts.

1.1. DOCUMENTATION GOALS

This guide aims to document SystemTap's most useful and common tapset entries; it also contains guidelines on proper tapset development and documentation. The tapset definitions contained in this guide are extracted automatically from properly-formatted comments in the code of each tapset file. As such, any revisions to the definitions in this guide should be applied directly to their respective tapset file.

CHAPTER 2. TAPSET DEVELOPMENT GUIDELINES

This chapter describes the upstream guidelines on proper tapset documentation. It also contains information on how to properly document your tapsets, to ensure that they are properly defined in this guide.

2.1. WRITING GOOD TAPSETS

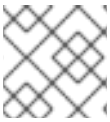
The first step to writing good tapsets is to create a simple model of your subject area. For example, a model of the process subsystem might include the following:

Key Data

- process ID
- parent process ID
- process group ID

State Transitions

- forked
- exec'd
- running
- stopped
- terminated



NOTE

Both lists are examples, and are not meant to represent a complete list.

Use your subsystem expertise to find probe points (function entries and exits) that expose the elements of the model, then define probe aliases for those points. Be aware that some state transitions can occur in more than one place. In those cases, an alias can place a probe in multiple locations.

For example, process execs can occur in either the **do_execve()** or the **compat_do_execve()** functions. The following alias inserts probes at the beginning of those functions:

```
probe kprocess.exec = kernel.function("do_execve"),
kernel.function("compat_do_execve")
{probe body}
```

Try to place probes on stable interfaces (i.e., functions that are unlikely to change at the interface level) whenever possible. This will make the tapset less likely to break due to kernel changes. Where kernel version or architecture dependencies are unavoidable, use preprocessor conditionals (see the **stap(1)** man page for details).

Fill in the probe bodies with the key data available at the probe points. Function entry probes can access the entry parameters specified to the function, while exit probes can access the entry parameters and the return value. Convert the data into meaningful forms where appropriate (e.g.,

bytes to kilobytes, state values to strings, etc).

You may need to use auxiliary functions to access or convert some of the data. Auxiliary functions often use embedded C to do things that cannot be done in the SystemTap language, like access structure fields in some contexts, follow linked lists, etc. You can use auxiliary functions defined in other tapsets or write your own.

In the following example, **`copy_process()`** returns a pointer to the **`task_struct`** for the new process. Note that the process ID of the new process is retrieved by calling **`task_pid()`** and passing it the **`task_struct`** pointer. In this case, the auxiliary function is an embedded C function defined in **`task.stp`**.

```
probe kprocess.create = kernel.function("copy_process").return
{
   task = $return
   new_pid = task_pid(task)
}
```

It is not advisable to write probes for every function. Most SystemTap users will not need or understand them. Keep your tapsets simple and high-level.

2.2. ELEMENTS OF A TAPSET

The following sections describe the most important aspects of writing a tapset. Most of the content herein is suitable for developers who wish to contribute to SystemTap's upstream library of tapsets.

2.2.1. Tapset Files

Tapset files are stored in ***`src/`*****`tapset/`** of the SystemTap GIT directory. Most tapset files are kept at that level. If you have code that only works with a specific architecture or kernel version, you may choose to put your tapset in the appropriate subdirectory.

Installed tapsets are located in **`/usr/share/systemtap/tapset/`** or **`/usr/local/share/systemtap/tapset`**.

Personal tapsets can be stored anywhere. However, to ensure that SystemTap can use them, use **`-I`** ***`tapset_directory`*** to specify their location when invoking **`stap`**.

2.2.2. Namespace

Probe alias names should take the form ***`tapset_name.probe_name`***. For example, the probe for sending a signal could be named **`signal.send`**.

Global symbol names (probes, functions, and variables) should be unique accross all tapsets. This helps avoid namespace collisions in scripts that use multiple tapsets. To ensure this, use tapset-specific prefixes in your global symbols.

Internal symbol names should be prefixed with an underscore (_).

2.2.3. Comments and Documentation

All probes and functions should include comment blocks that describe their purpose, the data they provide, and the context in which they run (e.g. interrupt, process, etc). Use comments in areas where your intent may not be clear from reading the code.

Note that specially-formatted comments are automatically extracted from most tapsets and included in this guide. This helps ensure that tapset contributors can write their tapset *and* document it in the same place. The specified format for documenting tapsets is as follows:

```
/**
 * probe tapset.name - Short summary of what the tapset does.
 * @argument: Explanation of argument.
 * @argument2: Explanation of argument2. Probes can have multiple
arguments.
 *
 * Context:
 * A brief explanation of the tapset context.
 * Note that the context should only be 1 paragraph short.
 *
 * Text that will appear under "Description."
 *
 * A new paragraph that will also appear under the heading "Description".
 *
 * Header:
 * A paragraph that will appear under the heading "Header".
 **/
```

For example:

```
/**
 * probe vm.write_shared_copy- Page copy for shared page write.
 * @address: The address of the shared write.
 * @zero: Boolean indicating whether it is a zero page
 *         (can do a clear instead of a copy).
 *
 * Context:
 *  The process attempting the write.
 *
 *  Fires when a write to a shared page requires a page copy.  This is
 *  always preceded by a vm.shared_write.
 **/
```

To override the automatically-generated **Synopsis** content, use:

```
 * Synopsis:
 * New Synopsis string
 *
```

For example:

```
/**
 * probe signal.handle - Fires when the signal handler is invoked
 * @sig: The signal number that invoked the signal handler
 *
 * Synopsis:
 * <programlisting>static int handle_signal(unsigned long sig, siginfo_t
*info, struct k_sigaction *ka,
 * sigset_t *oldset, struct pt_regs * regs)</programlisting>
 */
```

It is recommended that you use the **<programlisting>** tag in this instance, since overriding the **Synopsis** content of an entry does not automatically form the necessary tags.

For the purposes of improving the DocBook XML output of your comments, you can also use the following XML tags in your comments:

- **command**
- **emphasis**
- **programlisting**
- **remark** (tagged strings will appear in Publican beta builds of the document)

CHAPTER 3. CONTEXT FUNCTIONS

The context functions provide additional information about where an event occurred. These functions can provide information such as a backtrace to where the event occurred and the current register values for the processor.

NAME
function::print_regs — Print a register dump.

SYNOPSIS

```
function print_regs()
```

ARGUMENTS
None

GENERAL SYNTAX
print_regs

DESCRIPTION
This function prints a register dump.

NAME
function::execname — Returns the execname of a target process (or group of processes).

SYNOPSIS

```
function execname:string()
```

ARGUMENTS
None

GENERAL SYNTAX
execname:**string**

DESCRIPTION
Returns the execname of a target process (or group of processes).

NAME
function::pid — Returns the ID of a target process.

SYNOPSIS

```
function pid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
pid:**long**

DESCRIPTION
This function returns the ID of a targer process.

NAME
function::tid — Returns the thread ID of a target process.

SYNOPSIS

```
function tid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
tid:**long**

DESCRIPTION
This function returns the thread ID of the target process.

NAME
function::ppid — Returns the process ID of a target process's parent process.

SYNOPSIS

```
function ppid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
ppid:**long**

DESCRIPTION
This function return the process ID of the target proccess's parent process.

NAME
function::pgrp — Returns the process group ID of the current process.

SYNOPSIS

```
function pgrp:long()
```

ARGUMENTS
None

GENERAL SYNTAX
pgrp:**long**

DESCRIPTION
This function returns the process group ID of the current process.

NAME
function::sid — Returns the session ID of the current process.

SYNOPSIS

```
function sid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
sid:**long**

DESCRIPTION
The session ID of a process is the process group ID of the session leader. Session ID is stored in the signal_struct since Kernel 2.6.0.

NAME
function::pexecname — Returns the execname of a target process's parent process.

SYNOPSIS

```
function pexecname:string()
```

ARGUMENTS
None

GENERAL SYNTAX
pexecname:**string**

DESCRIPTION
This function returns the execname of a target process's parent procces.

NAME
function::gid — Returns the group ID of a target process.

SYNOPSIS

```
function gid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
gid:**long**

DESCRIPTION
This function returns the group ID of a target process.

NAME
function::egid — Returns the effective gid of a target process.

SYNOPSIS

```
function egid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
egid:**long**

DESCRIPTION
This function returns the effective gid of a target process

NAME
function::uid — Returns the user ID of a target process.

SYNOPSIS

```
function uid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
uid:**long**

DESCRIPTION
This function returns the user ID of the target process.

NAME
function::euid — Return the effective uid of a target process.

SYNOPSIS

```
function euid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
euid:**long**

DESCRIPTION
Returns the effective user ID of the target process.

NAME
function::is_myproc — Determines if the current probe point has occurred in the user's own process.

SYNOPSIS

```
function is_myproc:long()
```

ARGUMENTS
None

GENERAL SYNTAX
is_myproc:**long**

DESCRIPTION
This function returns 1 if the current probe point has occurred in the user's own process.

NAME
function::cpu — Returns the current cpu number.

SYNOPSIS

```
function cpu:long()
```

ARGUMENTS

None

GENERAL SYNTAX
cpu:**long**

DESCRIPTION
This function returns the current cpu number.

NAME
function::pp — Returns the active probe point.

SYNOPSIS

```
function pp:string()
```

ARGUMENTS
None

GENERAL SYNTAX
pp:**string**

DESCRIPTION
This function returns the fully-resolved probe point associated with a currently running probe handler, including alias and wild-card expansion effects. Context: The current probe point.

NAME
function::registers_valid — Determines validity of **register** and **u_register** in current context.

SYNOPSIS

```
function registers_valid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
registers_valid:**long**

DESCRIPTION
This function returns 1 if **register** and **u_register** can be used in the current context, or 0 otherwise. For example, **registers_valid** returns 0 when called from a begin or end probe.

NAME
function::user_mode — Determines if probe point occurs in user-mode.

SYNOPSIS

```
function user_mode:long()
```

ARGUMENTS
None

GENERAL SYNTAX
user_mode:**long**

Return 1 if the probe point occurred in user-mode.

NAME
function::is_return — Whether the current probe context is a return probe.

SYNOPSIS

```
function is_return:long()
```

ARGUMENTS
None

GENERAL SYNTAX
is_return:**long**

DESCRIPTION
Returns 1 if the current probe context is a return probe, returns 0 otherwise.

NAME
function::target — Return the process ID of the target process.

SYNOPSIS

```
function target:long()
```

ARGUMENTS
None

GENERAL SYNTAX
target:**long**

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

NAME
function::module_name — The module name of the current script.

SYNOPSIS

```
function module_name:string()
```

ARGUMENTS
None

GENERAL SYNTAX
module_name:**string**

DESCRIPTION
This function returns the name of the stap module. Either generated randomly (stap_[0-9a-f]+_[0-9a-f]+) or set by stap -m <module_name>.

NAME
function::stp_pid — The process id of the stapio process.

SYNOPSIS

```
function stp_pid:long()
```

ARGUMENTS
None

GENERAL SYNTAX
stp_pid:**long**

DESCRIPTION
This function returns the process id of the stapio process that launched this script. There could be other SystemTap scripts and stapio processes running on the system.

NAME
function::stack_size — Return the size of the kernel stack.

SYNOPSIS

```
function stack_size:long()
```

ARGUMENTS
None

GENERAL SYNTAX
stack_size:**long**

DESCRIPTION
This function returns the size of the kernel stack.

NAME
function::stack_used — Returns the amount of kernel stack used.

SYNOPSIS

```
function stack_used:long()
```

ARGUMENTS
None

GENERAL SYNTAX
stack_used:**long**

DESCRIPTION
This function determines how many bytes are currently used in the kernel stack.

NAME
function::stack_unused — Returns the amount of kernel stack currently available.

SYNOPSIS

```
function stack_unused:long()
```

ARGUMENTS
None

GENERAL SYNTAX
stack_unused:**long**

DESCRIPTION
This function determines how many bytes are currently available in the kernel stack.

NAME
function::uaddr — User space address of current running task. EXPERIMENTAL.

SYNOPSIS

```
function uaddr:long()
```

ARGUMENTS

None

GENERAL SYNTAX
uaddr:**long**

DESCRIPTION
Returns the address in userspace that the current task was at when the probe occurred. When the current running task isn't a user space thread, or the address cannot be found, zero is returned. Can be used to see where the current task is combined with **usymname** or **symdata**. Often the task will be in the VDSO where it entered the kernel. FIXME - need VDSO tracking support #10080.

NAME
function::cmdline_args — Fetch command line arguments from current process

SYNOPSIS

```
function cmdline_args:string(n:long,m:long,delim:string)
```

ARGUMENTS

n

First argument to get (zero is the command itself)

m

Last argument to get (or minus one for all arguments after n)

delim

String to use to delimit arguments when more than one.

GENERAL SYNTAX
cmdline_args:string(n:long, m:long, delim:string)

DESCRIPTION
Returns arguments from the current process starting with argument number n, up to argument m. If there are less than n arguments, or the arguments cannot be retrieved from the current process, the empty string is returned. If m is smaller than n then all arguments starting from argument n are returned. Argument zero is traditionally the command itself.

NAME
function::cmdline_arg — Fetch a command line argument.

SYNOPSIS

```
function cmdline_arg:string(n:long)
```

ARGUMENTS

n

Argument to get (zero is the command itself)

GENERAL SYNTAX

cmdline_arg:string(n:long)

DESCRIPTION

Returns argument the requested argument from the current process or the empty string when there are not that many arguments or there is a problem retrieving the argument. Argument zero is traditionally the command itself.

NAME

function::cmdline_str — Fetch all command line arguments from current process

SYNOPSIS

```
function cmdline_str:string()
```

ARGUMENTS

None

GENERAL SYNTAX

cmdline_str:**string**

DESCRIPTION

Returns all arguments from the current process delimited by spaces. Returns the empty string when the arguments cannot be retrieved.

NAME

function::env_var — Fetch environment variable from current process

SYNOPSIS

```
function env_var:string(name:string)
```

ARGUMENTS

name

Name of the environment variable to fetch

GENERAL SYNTAX

evn_var:string(name:string)

DESCRIPTION

Returns the contents of the specified environment value for the current process. If the variable isn't set an empty string is returned.

NAME

function::print_stack — Print out kernel stack from string.

SYNOPSIS

```
function print_stack(stk:string)
```

ARGUMENTS

stk

String with list of hexadecimal addresses.

GENERAL SYNTAX

print_stack(stk:string)

DESCRIPTION

This function performs a symbolic lookup of the addresses in the given string, which is assumed to be the result of a prior call to **backtrace**.

Print one line per address, including the address, the name of the function containing the address, and an estimate of its position within that function. Return nothing.

NAME

function::sprint_stack — Return stack for kernel addresses from string. EXPERIMENTAL!

SYNOPSIS

```
function sprint_stack:string(stk:string)
```

ARGUMENTS

stk

String with list of hexadecimal (kernel) addresses.

DESCRIPTION

Perform a symbolic lookup of the addresses in the given string, which is assumed to be the result of a prior call to **backtrace**.

Returns a simple backtrace from the given hex string. One line per address. Includes the symbol name (or hex address if symbol couldn't be resolved) and module name (if found). Includes the offset from the start of the function if found, otherwise the offset will be added to the module (if found, between

brackets). Returns the backtrace as string (each line terminated by a newline character). Note that the returned stack will be truncated to MAXSTRINGLEN, to print fuller and richer stacks use print_stack.

NAME
function::probefunc — Return the probe point's function name, if known.

SYNOPSIS

```
function probefunc:string()
```

ARGUMENTS
None

GENERAL SYNTAX
probefunc:**string**

DESCRIPTION
This function returns the name of the function being probed. It will do this based on the probe point string as returned by **pp**.

PLEASE NOTE
this function is deprecated, please use **symname** and/or **usymname**. This function might return a function name based on the current address if the probe point context couldn't be parsed.

NAME
function::probemod — Return the probe point's kernel module name.

SYNOPSIS

```
function probemod:string()
```

ARGUMENTS
None

GENERAL SYNTAX
probemod:**string**

DESCRIPTION
This funciton returns the name of the kernel module containing the probe point, if known.

NAME
function::modname — Return the kernel module name loaded at the address.

SYNOPSIS

```
function modname:string(addr:long)
```

ARGUMENTS

addr

The address.

DESCRIPTION

Returns the module name associated with the given address if known. If not known it will return the string "<unknown>". If the address was not in a kernel module, but in the kernel itself, then the string "kernel" will be returned.

NAME

function::symname — Return the kernel symbol associated with the given address.

SYNOPSIS

```
function symname:string(addr:long)
```

ARGUMENTS

addr

The address to translate.

GENERAL SYNTAX

symname:string(addr:long)

DESCRIPTION

Returns the (function) symbol name associated with the given address if known. If not known it will return the hex string representation of addr.

NAME

function::symdata — Return the kernel symbol and module offset for the address.

SYNOPSIS

```
function symdata:string(addr:long)
```

ARGUMENTS

addr

The address to translate.

GENERAL SYNTAX
symdata:string(addr:long)

DESCRIPTION
Returns the (function) symbol name associated with the given address if known, the offset from the start and size of the symbol, plus module name (between brackets). If symbol is unknown, but module is known, the offset inside the module, plus the size of the module is added. If any element is not known it will be omitted and if the symbol name is unknown it will return the hex string for the given address.

NAME
function::usymname — Return the symbol of an address in the current task. EXPERIMENTAL!

SYNOPSIS

```
function usymname:string(addr:long)
```

ARGUMENTS

addr

The address to translate.

DESCRIPTION
Returns the (function) symbol name associated with the given address if known. If not known it will return the hex string representation of addr.

NAME
function::usymdata — Return the symbol and module offset of an address. EXPERIMENTAL!

SYNOPSIS

```
function usymdata:string(addr:long)
```

ARGUMENTS

addr

The address to translate.

DESCRIPTION
Returns the (function) symbol name associated with the given address in the current task if known, the offset from the start and the size of the symbol, plus the module name (between brackets). If symbol is unknown, but module is known, the offset inside the module, plus the size of the module is added. If any element is not known it will be omitted and if the symbol name is unknown it will return the hex string for the given address.

NAME

function::print_ustack — Print out stack for the current task from string. EXPERIMENTAL!

SYNOPSIS

```
function print_ustack(stk:string)
```

ARGUMENTS

stk

String with list of hexadecimal addresses for the current task.

DESCRIPTION

Perform a symbolic lookup of the addresses in the given string, which is assumed to be the result of a prior call to **ubacktrace** for the current task.

Print one line per address, including the address, the name of the function containing the address, and an estimate of its position within that function. Return nothing.

NAME

function::sprint_ustack — Return stack for the current task from string. EXPERIMENTAL!

SYNOPSIS

```
function sprint_ustack:string(stk:string)
```

ARGUMENTS

stk

String with list of hexadecimal addresses for the current task.

DESCRIPTION

Perform a symbolic lookup of the addresses in the given string, which is assumed to be the result of a prior call to **ubacktrace** for the current task.

Returns a simple backtrace from the given hex string. One line per address. Includes the symbol name (or hex address if symbol couldn't be resolved) and module name (if found). Includes the offset from the start of the function if found, otherwise the offset will be added to the module (if found, between brackets). Returns the backtrace as string (each line terminated by a newline character). Note that the returned stack will be truncated to MAXSTRINGLEN, to print fuller and richer stacks use print_ustack.

NAME

function::print_backtrace — Print stack back trace

SYNOPSIS

```
function print_backtrace()
```

ARGUMENTS
None

GENERAL SYNTAX
print_backtrace

DESCRIPTION
This function isEquivalent to print_stack(**backtrace**), except that deeper stack nesting may be supported. The function does not return a value.

NAME
function::sprint_backtrace — Return stack back trace as string. EXPERIMENTAL!

SYNOPSIS

```
function sprint_backtrace:string()
```

ARGUMENTS
None

DESCRIPTION
Returns a simple (kernel) backtrace. One line per address. Includes the symbol name (or hex address if symbol couldn't be resolved) and module name (if found). Includes the offset from the start of the function if found, otherwise the offset will be added to the module (if found, between brackets). Returns the backtrace as string (each line terminated by a newline character). Note that the returned stack will be truncated to MAXSTRINGLEN, to print fuller and richer stacks use **print_backtrace**. Equivalent to sprint_stack(**backtrace**), but more efficient (no need to translate between hex strings and final backtrace string).

NAME
function::backtrace — Hex backtrace of current stack

SYNOPSIS

```
function backtrace:string()
```

ARGUMENTS
None

GENERAL SYNTAX
backtrace:**string**

DESCRIPTION

This function returns a string of hex addresses that are a backtrace of the stack. Output may be truncated as as per maximum string length (MAXSTRINGLEN).

NAME

function::task_backtrace — Hex backtrace of an arbitrary task

SYNOPSIS

```
function task_backtrace:string(task:long)
```

ARGUMENTS

task

pointer to task_struct

GENERAL SYNTAX

task_backtrace:string(task:long)

DESCRIPTION

This function returns a string of hex addresses that are a backtrace of the stack of a particular task Output may be truncated as per maximum string length.

NAME

function::caller — Return name and address of calling function

SYNOPSIS

```
function caller:string()
```

ARGUMENTS

None

GENERAL SYNTAX

caller:**string**

DESCRIPTION

This function returns the address and name of the calling function. This is equivalent to calling: sprintf("**s** 0x**x**", symname(**caller_addr**, **caller_addr**)) Works only for return probes at this time.

NAME

function::caller_addr — Return caller address

SYNOPSIS

-

```
function caller_addr:long()
```

ARGUMENTS
None

GENERAL SYNTAX
caller_addr:**long**

DESCRIPTION
This function returns the address of the calling function. Works only for return probes at this time.

NAME
function::print_ubacktrace — Print stack back trace for current task. EXPERIMENTAL!

SYNOPSIS

```
function print_ubacktrace()
```

ARGUMENTS
None

DESCRIPTION
Equivalent to print_ustack(**ubacktrace**), except that deeper stack nesting may be supported. Returns nothing.

NOTE
To get (full) backtraces for user space applications and shared shared libraries not mentioned in the current script run stap with -d /path/to/exe-or-so and/or add --ldd to load all needed unwind data.

NAME
function::sprint_ubacktrace — Return stack back trace for current task as string. EXPERIMENTAL!

SYNOPSIS

```
function sprint_ubacktrace:string()
```

ARGUMENTS
None

DESCRIPTION
Returns a simple backtrace for the current task. One line per address. Includes the symbol name (or hex address if symbol couldn't be resolved) and module name (if found). Includes the offset from the start of the function if found, otherwise the offset will be added to the module (if found, between brackets). Returns the backtrace as string (each line terminated by a newline character). Note that the

returned stack will be truncated to MAXSTRINGLEN, to print fuller and richer stacks use **print_ubacktrace**. Equivalent to sprint_ustack(**ubacktrace**), but more efficient (no need to translate between hex strings and final backtrace string).

NOTE

To get (full) backtraces for user space applications and shared shared libraries not mentioned in the current script run stap with -d /path/to/exe-or-so and/or add --ldd to load all needed unwind data.

NAME

function::print_ubacktrace_brief — Print stack back trace for current task. EXPERIMENTAL!

SYNOPSIS

```
function print_ubacktrace_brief()
```

ARGUMENTS

None

DESCRIPTION

Equivalent to **print_ubacktrace**, but output for each symbol is shorter (just name and offset, or just the hex address of no symbol could be found).

NOTE

To get (full) backtraces for user space applications and shared shared libraries not mentioned in the current script run stap with -d /path/to/exe-or-so and/or add --ldd to load all needed unwind data.

NAME

function::ubacktrace — Hex backtrace of current task stack. EXPERIMENTAL!

SYNOPSIS

```
function ubacktrace:string()
```

ARGUMENTS

None

DESCRIPTION

Return a string of hex addresses that are a backtrace of the stack of the current task. Output may be truncated as per maximum string length. Returns empty string when current probe point cannot determine user backtrace.

NOTE

To get (full) backtraces for user space applications and shared shared libraries not mentioned in the current script run stap with -d /path/to/exe-or-so and/or add --ldd to load all needed unwind data.

NAME
function::task_current — The current task_struct of the current task.

SYNOPSIS

```
function task_current:long()
```

ARGUMENTS
None

GENERAL SYNTAX
task_current:**long**

DESCRIPTION
This function returns the task_struct representing the current process. This address can be passed to the various task_*() functions to extract more task-specific data.

NAME
function::task_parent — The task_struct of the parent task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX
task_parent:long(task:long)

DESCRIPTION
This function returns the parent task_struct of the given task. This address can be passed to the various task_*() functions to extract more task-specific data.

NAME
function::task_state — The state of the task.

SYNOPSIS

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

ARGUMENTS

task

task

task_struct pointer.

GENERAL SYNTAX

task_state:long(task:long)

DESCRIPTION

Return the state of the given task, one of: TASK_RUNNING (0), TASK_INTERRUPTIBLE (1), TASK_UNINTERRUPTIBLE (2), TASK_STOPPED (4), TASK_TRACED (8), EXIT_ZOMBIE (16), EXIT_DEAD (32).

NAME

function::task_execname — The name of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX

task_execname:string(task:long)

DESCRIPTION

Return the name of the given task.

NAME

function::task_pid — The process identifier of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX

task_pid:long (task:long)

DESCRIPTION

This fucntion returns the process id of the given task.

NAME

function::pid2task — The task_struct of the given process identifier.

SYNOPSIS

```
function pid2task:long(pid:long)
```

ARGUMENTS

pid

Process identifier.

DESCRIPTION

Return the task struct of the given process id.

NAME

function::pid2execname — The name of the given process identifier.

SYNOPSIS

```
function pid2execname:string(pid:long)
```

ARGUMENTS

pid

Process identifier.

DESCRIPTION

Return the name of the given process id.

NAME

function::task_tid — The thread identifier of the task.

SYNOPSIS

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

ARGUMENTS

task

task

task_struct pointer.

GENERAL SYNTAX

task_tid:long(task:long)

DESCRIPTION

This function returns the thread id of the given task.

NAME

function::task_gid — The group identifier of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX

task_gid:long(task:long)

DESCRIPTION

This function returns the group id of the given task.

NAME

function::task_egid — The effective group identifier of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX

task_egid:long(task:long)

DESCRIPTION

This function returns the effective group id of the given task.

NAME
function::task_uid — The user identifier of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX
task_uid:long(task:long)

DESCRIPTION
This function returns the user id of the given task.

NAME
function::task_euid — The effective user identifier of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX
task_euid:long(task:long)

DESCRIPTION
This function returns the effective user id of the given task.

NAME
function::task_prio — The priority value of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX

task_prio:long(task:long)

DESCRIPTION

This function returns the priority value of the given task.

NAME

function::task_nice — The nice value of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX

task_nice:long(task:long)

DESCRIPTION

This function returns the nice value of the given task.

NAME

function::task_cpu — The scheduled cpu of the task.

SYNOPSIS

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

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX

task_cpu:long(task:long)

DESCRIPTION
This function returns the scheduled cpu for the given task.

NAME
function::task_open_file_handles — The number of open files of the task.

SYNOPSIS

```
function task_open_file_handles:long(task:long)
```

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX
task_open_file_handles:long(task:long)

DESCRIPTION
This function returns the number of open file handlers for the given task.

NAME
function::task_max_file_handles — The max number of open files for the task.

SYNOPSIS

```
function task_max_file_handles:long(task:long)
```

ARGUMENTS

task

task_struct pointer.

GENERAL SYNTAX
task_max_file_handles:long(task:long)

DESCRIPTION
This function returns the maximum number of file handlers for the given task.

NAME
function::pn — Returns the active probe name.

SYNOPSIS

```
function pn:string()
```

ARGUMENTS

None

GENERAL SYNTAX

pn:**string**

DESCRIPTION

This function returns the script-level probe point associated with a currently running probe handler, including wild-card expansion effects. Context: The current probe point.

CHAPTER 4. TIMESTAMP FUNCTIONS

Each timestamp function returns a value to indicate when a function is executed. These returned values can then be used to indicate when an event occurred, provide an ordering for events, or compute the amount of time elapsed between two time stamps.

NAME

function::get_cycles — Processor cycle count.

SYNOPSIS

```
function get_cycles:long()
```

ARGUMENTS

None

GENERAL SYNTAX

get_cycles:**long**

DESCRIPTION

This function returns the processor cycle counter value if available, else it returns zero. The cycle counter is free running and unsynchronized on each processor. Thus, the order of events cannot determined by comparing the results of the get_cycles function on different processors.

NAME

function::gettimeofday_ns — Number of nanoseconds since UNIX epoch.

SYNOPSIS

```
function gettimeofday_ns:long()
```

ARGUMENTS

None

GENERAL SYNTAX

gettimeofday_ns:**long**

DESCRIPTION

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

NAME

function::gettimeofday_us — Number of microseconds since UNIX epoch.

SYNOPSIS

```
function gettimeofday_us:long()
```

ARGUMENTS
None

GENERAL SYNTAX
gettimeofday_us:**long**

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

NAME
function::gettimeofday_ms — Number of milliseconds since UNIX epoch.

SYNOPSIS

```
function gettimeofday_ms:long()
```

ARGUMENTS
None

GENERAL SYNTAX
gettimeofday_ms:**long**

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

NAME
function::gettimeofday_s — Number of seconds since UNIX epoch.

SYNOPSIS

```
function gettimeofday_s:long()
```

ARGUMENTS
None

GENERAL SYNTAX
gettimeofday_s:**long**

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

CHAPTER 5. TIME STRING UTILITY FUNCTION

Utility function to turn seconds since the epoch (as returned by the timestamp function gettimeofday_s()) into a human readable date/time string.

NAME

function::ctime — Convert seconds since epoch into human readable date/time string.

SYNOPSIS

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

ARGUMENTS

epochsecs

Number of seconds since epoch (as returned by **gettimeofday_s**).

GENERAL SYNTAX

ctime:string(epochsecs:long)

DESCRIPTION

Takes an argument of seconds since the epoch as returned by **gettimeofday_s**. Returns a string of the form

“Wed Jun 30 21:49:08 1993 ”

The string will always be exactly 24 characters. If the time would be unreasonable far in the past (before what can be represented with a 32 bit offset in seconds from the epoch) the returned string will be “a long, long time ago... ”. If the time would be unreasonable far in the future the returned string will be “far far in the future... ” (both these strings are also 24 characters wide).

Note that the epoch (zero) corresponds to

“Thu Jan 1 00:00:00 1970 ”

The earliest full date given by ctime, corresponding to epochsecs -2147483648 is “Fri Dec 13 20:45:52 1901”. The latest full date given by ctime, corresponding to epochsecs 2147483647 is “Tue Jan 19 03:14:07 2038”.

The abbreviations for the days of the week are ‘Sun’, ‘Mon’, ‘Tue’, ‘Wed’, ‘Thu’, ‘Fri’, and ‘Sat’. The abbreviations for the months are ‘Jan’, ‘Feb’, ‘Mar’, ‘Apr’, ‘May’, ‘Jun’, ‘Jul’, ‘Aug’, ‘Sep’, ‘Oct’, ‘Nov’, and ‘Dec’.

Note that the real C library **ctime** function puts a newline ('\n') character at the end of the string that this function does not. Also note that since the kernel has no concept of timezones, the returned time is always in GMT.

CHAPTER 6. MEMORY TAPSET

This family of probe points is used to probe memory-related events or query the memory usage of the current process. It contains the following probe points:

NAME

function::vm_fault_contains — Test return value for page fault reason

SYNOPSIS

```
function vm_fault_contains:long(value:long,test:long)
```

ARGUMENTS

value

The fault_type returned by vm.page_fault.return

test

The type of fault to test for (VM_FAULT_OOM or similar)

NAME

probe::vm.pagefault — Records that a page fault occurred.

SYNOPSIS

```
vm.pagefault
```

VALUES

write_access

Indicates whether this was a write or read access; 1 indicates a write, while 0 indicates a read.

name

Name of the probe point

address

The address of the faulting memory access; i.e. the address that caused the page fault.

CONTEXT

The process which triggered the fault

NAME

probe::vm.pagefault.return — Indicates what type of fault occurred.

SYNOPSIS

```
vm.pagefault.return
```

VALUES

name

Name of the probe point

fault_type

Returns either 0 (VM_FAULT_OOM) for out of memory faults, 2 (VM_FAULT_MINOR) for minor faults, 3 (VM_FAULT_MAJOR) for major faults, or 1 (VM_FAULT_SIGBUS) if the fault was neither OOM, minor fault, nor major fault.

NAME

function::addr_to_node — Returns which node a given address belongs to within a NUMA system.

SYNOPSIS

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

ARGUMENTS

addr

The address of the faulting memory access.

GENERAL SYNTAX

addr_to_node:long(addr:long)

DESCRIPTION

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

NAME

probe::vm.write_shared — Attempts at writing to a shared page.

SYNOPSIS

```
vm.write_shared
```

VALUES

name

Name of the probe point

address

The address of the shared write.

CONTEXT

The context is the process attempting the write.

DESCRIPTION

Fires when a process attempts to write to a shared page. If a copy is necessary, this will be followed by a vm.write_shared_copy.

NAME

probe::vm.write_shared_copy — Page copy for shared page write.

SYNOPSIS

```
vm.write_shared_copy
```

VALUES

name

Name of the probe point

zero

Boolean indicating whether it is a zero page (can do a clear instead of a copy).

address

The address of the shared write.

CONTEXT

The process attempting the write.

DESCRIPTION

Fires when a write to a shared page requires a page copy. This is always preceded by a vm.shared_write.

NAME

probe::vm.mmap — Fires when an mmap is requested.

SYNOPSIS

```
vm.mmap
```

VALUES

length

The length of the memory segment

name

Name of the probe point

address

The requested address

CONTEXT

The process calling mmap.

NAME

probe::vm.munmap — Fires when an munmap is requested.

SYNOPSIS

```
vm.munmap
```

VALUES

length

The length of the memory segment

name

Name of the probe point

address

The requested address

CONTEXT

The process calling munmap.

NAME

probe::vm.brk — Fires when a brk is requested (i.e. the heap will be resized).

SYNOPSIS

```
vm.brk
```

VALUES

length

The length of the memory segment

name

Name of the probe point

address

The requested address

CONTEXT

The process calling brk.

NAME

probe::vm.oom_kill — Fires when a thread is selected for termination by the OOM killer.

SYNOPSIS

```
vm.oom_kill
```

VALUES

name

Name of the probe point

task

The task being killed

CONTEXT

The process that tried to consume excessive memory, and thus triggered the OOM.

NAME

probe::vm.kmalloc — Fires when kmalloc is requested.

SYNOPSIS

```
vm.kmalloc
```

VALUES

ptr

Pointer to the kmemory allocated

caller_function

Name of the caller function.

call_site

Address of the kmemory function.

gfp_flag_name

type of kmemory to allocate (in String format)

name

Name of the probe point

bytes_req

Requested Bytes

bytes_alloc

Allocated Bytes

gfp_flags

type of kmemory to allocate

NAME

probe::vm.kmem_cache_alloc — Fires when \

SYNOPSIS

```
vm.kmem_cache_alloc
```

VALUES

ptr

Pointer to the kmemory allocated

caller_function

Name of the caller function.

call_site

Address of the function calling this kmemory function.

gfp_flag_name

Type of kmemory to allocate(in string format)

name

Name of the probe point

bytes_req

Requested Bytes

bytes_alloc

Allocated Bytes

gfp_flags

type of kmemory to allocate

DESCRIPTION

kmem_cache_alloc is requested.

NAME

probe::vm.kmalloc_node — Fires when kmalloc_node is requested.

SYNOPSIS

```
vm.kmalloc_node
```

VALUES

ptr

Pointer to the kmemory allocated

caller_function

Name of the caller function.

call_site

Address of the function caling this kmemory function.

gfp_flag_name

Type of kmemory to allocate(in string format)

name

Name of the probe point

bytes_req

Requested Bytes

bytes_alloc

Allocated Bytes

gfp_flags

type of kmemory to allocate

NAME
probe::vm.kmem_cache_alloc_node — Fires when \

SYNOPSIS

```
vm.kmem_cache_alloc_node
```

VALUES

ptr

Pointer to the kmemory allocated

caller_function

Name of the caller function.

call_site

Address of the function calling this kmemory function.

gfp_flag_name

Type of kmemory to allocate(in string format)

name

Name of the probe point

bytes_req

Requested Bytes

bytes_alloc

Allocated Bytes

gfp_flags

type of kmemory to allocate

DESCRIPTION
kmem_cache_alloc_node is requested.

NAME
probe::vm.kfree — Fires when kfree is requested.

SYNOPSIS

```
vm.kfree
```

VALUES

ptr

Pointer to the kmemory allocated which is returned by kmalloc

caller_function

Name of the caller function.

call_site

Address of the function calling this kmemory function.

name

Name of the probe point

NAME

probe::vm.kmem_cache_free — Fires when \

SYNOPSIS

```
vm.kmem_cache_free
```

VALUES

ptr

Pointer to the kmemory allocated which is returned by kmem_cache

caller_function

Name of the caller function.

call_site

Address of the function calling this kmemory function.

name

Name of the probe point

DESCRIPTION

kmem_cache_free is requested.

NAME

function::proc_mem_size — Total program virtual memory size in pages

SYNOPSIS

-

```
function proc_mem_size:long()
```

ARGUMENTS
None

DESCRIPTION
Returns the total virtual memory size in pages of the current process, or zero when there is no current process or the number of pages couldn't be retrieved.

NAME
function::proc_mem_size_pid — Total program virtual memory size in pages

SYNOPSIS

```
function proc_mem_size_pid:long(pid:long)
```

ARGUMENTS

pid

The pid of process to examine

DESCRIPTION
Returns the total virtual memory size in pages of the given process, or zero when that process doesn't exist or the number of pages couldn't be retrieved.

NAME
function::proc_mem_rss — Program resident set size in pages

SYNOPSIS

```
function proc_mem_rss:long()
```

ARGUMENTS
None

DESCRIPTION
Returns the resident set size in pages of the current process, or zero when there is no current process or the number of pages couldn't be retrieved.

NAME
function::proc_mem_rss_pid — Program resident set size in pages

SYNOPSIS

```
function proc_mem_rss_pid:long(pid:long)
```

ARGUMENTS

pid

The pid of process to examine

DESCRIPTION

Returns the resident set size in pages of the given process, or zero when the process doesn't exist or the number of pages couldn't be retrieved.

NAME

function::proc_mem_shr — Program shared pages (from shared mappings)

SYNOPSIS

```
function proc_mem_shr:long()
```

ARGUMENTS

None

DESCRIPTION

Returns the shared pages (from shared mappings) of the current process, or zero when there is no current process or the number of pages couldn't be retrieved.

NAME

function::proc_mem_shr_pid — Program shared pages (from shared mappings)

SYNOPSIS

```
function proc_mem_shr_pid:long(pid:long)
```

ARGUMENTS

pid

The pid of process to examine

DESCRIPTION

Returns the shared pages (from shared mappings) of the given process, or zero when the process doesn't exist or the number of pages couldn't be retrieved.

NAME

function::proc_mem_txt — Program text (code) size in pages

SYNOPSIS

```
function proc_mem_txt:long()
```

ARGUMENTS

None

DESCRIPTION

Returns the current process text (code) size in pages, or zero when there is no current process or the number of pages couldn't be retrieved.

NAME

function::proc_mem_txt_pid — Program text (code) size in pages

SYNOPSIS

```
function proc_mem_txt_pid:long(pid:long)
```

ARGUMENTS

pid

The pid of process to examine

DESCRIPTION

Returns the given process text (code) size in pages, or zero when the process doesn't exist or the number of pages couldn't be retrieved.

NAME

function::proc_mem_data — Program data size (data + stack) in pages

SYNOPSIS

```
function proc_mem_data:long()
```

ARGUMENTS

None

DESCRIPTION

Returns the current process data size (data + stack) in pages, or zero when there is no current process or the number of pages couldn't be retrieved.

NAME
function::proc_mem_data_pid — Program data size (data + stack) in pages

SYNOPSIS

```
function proc_mem_data_pid:long(pid:long)
```

ARGUMENTS

pid

The pid of process to examine

DESCRIPTION
Returns the given process data size (data + stack) in pages, or zero when the process doesn't exist or the number of pages couldn't be retrieved.

NAME
function::mem_page_size — Number of bytes in a page for this architecture

SYNOPSIS

```
function mem_page_size:long()
```

ARGUMENTS
None

NAME
function::bytes_to_string — Human readable string for given bytes

SYNOPSIS

```
function bytes_to_string:string(bytes:long)
```

ARGUMENTS

bytes

Number of bytes to translate.

DESCRIPTION
Returns a string representing the number of bytes (up to 1024 bytes), the number of kilobytes (when less than 1024K) postfixed by 'K', the number of megabytes (when less than 1024M) postfixed by 'M' or the number of gigabytes postfixed by 'G'. If representing K, M or G, and the number is amount is less than 100, it includes a '.' plus the remainer. The returned string will be 5 characters wide (padding with whitespace at the front) unless negative or representing more than 9999G bytes.

NAME
function::pages_to_string — Turns pages into a human readable string

SYNOPSIS

```
function pages_to_string:string(pages:long)
```

ARGUMENTS

pages

Number of pages to translate.

DESCRIPTION
Multiplies pages by **page_size** to get the number of bytes and returns the result of **bytes_to_string**.

NAME
function::proc_mem_string — Human readable string of current proc memory usage

SYNOPSIS

```
function proc_mem_string:string()
```

ARGUMENTS
None

DESCRIPTION
Returns a human readable string showing the size, rss, shr, txt and data of the memory used by the current process. For example “size: 301m, rss: 11m, shr: 8m, txt: 52k, data: 2248k ”.

NAME
function::proc_mem_string_pid — Human readable string of process memory usage

SYNOPSIS

```
function proc_mem_string_pid:string(pid:long)
```

ARGUMENTS

pid

The pid of process to examine

DESCRIPTION

Returns a human readable string showing the size, rss, shr, txt and data of the memory used by the given process. For example “size: 301m, rss: 11m, shr: 8m, txt: 52k, data: 2248k ”.

CHAPTER 7. TASK TIME TAPSET

This tapset defines utility functions to query time related properties of the current tasks, translate those in miliseconds and human readable strings.

NAME

function::task_utime — User time of the current task

SYNOPSIS

```
function task_utime:long()
```

ARGUMENTS

None

DESCRIPTION

Returns the user time of the current task in cputime. Does not include any time used by other tasks in this process, nor does it include any time of the children of this task.

NAME

function::task_utime_tid — User time of the given task

SYNOPSIS

```
function task_utime_tid:long(tid:long)
```

ARGUMENTS

tid

Thread id of the given task

DESCRIPTION

Returns the user time of the given task in cputime, or zero if the task doesn't exist. Does not include any time used by other tasks in this process, nor does it include any time of the children of this task.

NAME

function::task_stime — System time of the current task

SYNOPSIS

```
function task_stime:long()
```

ARGUMENTS

None

DESCRIPTION

Returns the system time of the current task in cputime. Does not include any time used by other tasks in this process, nor does it include any time of the children of this task.

NAME

function::task_stime_tid — System time of the given task

SYNOPSIS

```
function task_stime_tid:long(tid:long)
```

ARGUMENTS

tid

Thread id of the given task

DESCRIPTION

Returns the system time of the given task in cputime, or zero if the task doesn't exist. Does not include any time used by other tasks in this process, nor does it include any time of the children of this task.

NAME

function::cputime_to_msecs — Translates the given cputime into milliseconds

SYNOPSIS

```
function cputime_to_msecs:long(cputime:long)
```

ARGUMENTS

cputime

Time to convert to milliseconds.

NAME

function::msecs_to_string — Human readable string for given milliseconds

SYNOPSIS

```
function msecs_to_string:string(msecs:long)
```

ARGUMENTS

msecs

Number of milliseconds to translate.

DESCRIPTION

Returns a string representing the number of milliseconds as a human readable string consisting of "XmY.ZZZs", where X is the number of minutes, Y is the number of seconds and ZZZ is the number of milliseconds.

NAME

function::cputime_to_string — Human readable string for given cputime

SYNOPSIS

```
function cputime_to_string:string(cputime:long)
```

ARGUMENTS

cputime

Time to translate.

DESCRIPTION

Equivalent to calling: msec_to_string (cputime_to_msecs (cputime).

NAME

function::task_time_string — Human readable string of task time usage

SYNOPSIS

```
function task_time_string:string()
```

ARGUMENTS

None

DESCRIPTION

Returns a human readable string showing the user and system time the current task has used up to now. For example "usr: 0m12.908s, sys: 1m6.851s".

NAME

function::task_time_string_tid — Human readable string of task time usage

SYNOPSIS

```
function task_time_string_tid:string(tid:long)
```

ARGUMENTS

tid

Thread id of the given task

DESCRIPTION

Returns a human readable string showing the user and system time the given task has used up to now. For example “usr: 0m12.908s, sys: 1m6.851s”.

CHAPTER 8. IO SCHEDULER AND BLOCK IO TAPSET

This family of probe points is used to probe block IO layer and IO scheduler activities. It contains the following probe points:

NAME

probe::ioscheduler.elv_next_request — Fires when a request is retrieved from the request queue

SYNOPSIS

```
ioscheduler.elv_next_request
```

VALUES

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled

NAME

probe::ioscheduler.elv_next_request.return — Fires when a request retrieval issues a return signal

SYNOPSIS

```
ioscheduler.elv_next_request.return
```

VALUES

disk_major

Disk major number of the request

rq

Address of the request

name

Name of the probe point

disk_minor

Disk minor number of the request

rq_flags

Request flags

NAME

probe::ioscheduler.elv_completed_request — Fires when a request is completed

SYNOPSIS

```
ioscheduler.elv_completed_request
```

VALUES

disk_major

Disk major number of the request

rq

Address of the request

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled

disk_minor

Disk minor number of the request

rq_flags

Request flags

NAME

probe::ioscheduler.elv_add_request.kp — kprobe based probe to indicate that a request was added to the request queue

SYNOPSIS

```
ioscheduler.elv_add_request.kp
```

VALUES

disk_major

Disk major number of the request

rq

Address of the request

q

pointer to request queue

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled

disk_minor

Disk minor number of the request

rq_flags

Request flags

NAME

probe::ioscheduler.elv_add_request.tp — tracepoint based probe to indicate a request is added to the request queue.

SYNOPSIS

```
ioscheduler.elv_add_request.tp
```

VALUES

disk_major

Disk major no of request.

rq

Address of request.

q

Pointer to request queue.

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled.

disk_minor

Disk minor number of request.

rq_flags

Request flags.

NAME

probe::ioscheduler.elv_add_request — probe to indicate request is added to the request queue.

SYNOPSIS

```
ioscheduler.elv_add_request
```

VALUES

disk_major

Disk major no of request.

rq

Address of request.

q

Pointer to request queue.

elevator_name

The type of I/O elevator currently enabled.

disk_minor

Disk minor number of request.

rq_flags

Request flags.

NAME

probe::ioscheduler_trace.elv_completed_request — Fires when a request is

SYNOPSIS

```
ioscheduler_trace.elv_completed_request
```

VALUES

disk_major

Disk major no of request.

rq

Address of request.

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled.

disk_minor

Disk minor number of request.

rq_flags

Request flags.

DESCRIPTION

completed.

NAME

probe::ioscheduler_trace.elv_issue_request — Fires when a request is

SYNOPSIS

```
ioscheduler_trace.elv_issue_request
```

VALUES

disk_major

Disk major no of request.

rq

Address of request.

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled.

disk_minor

Disk minor number of request.

rq_flags

Request flags.

DESCRIPTION

scheduled.

NAME

probe::ioscheduler_trace.elv_requeue_request — Fires when a request is

SYNOPSIS

```
ioscheduler_trace.elv_requeue_request
```

VALUES

disk_major

Disk major no of request.

rq

Address of request.

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled.

disk_minor

Disk minor number of request.

rq_flags

Request flags.

DESCRIPTION

put back on the queue, when the hadware cannot accept more requests.

NAME

probe::ioscheduler_trace.elv_abort_request — Fires when a request is aborted.

SYNOPSIS

```
ioscheduler_trace.elv_abort_request
```

VALUES

disk_major

Disk major no of request.

rq

Address of request.

name

Name of the probe point

elevator_name

The type of I/O elevator currently enabled.

disk_minor

Disk minor number of request.

rq_flags

Request flags.

NAME

probe::ioscheduler_trace.plug — Fires when a request queue is plugged;

SYNOPSIS

```
ioscheduler_trace.plug
```

VALUES

name

Name of the probe point

rq_queue

request queue

DESCRIPTION

ie, requests in the queue cannot be serviced by block driver.

NAME

probe::ioscheduler_trace.unplug_io — Fires when a request queue is unplugged;

SYNOPSIS

```
ioscheduler_trace.unplug_io
```

VALUES

name

Name of the probe point

rq_queue

request queue

DESCRIPTION

Either, when number of pending requests in the queue exceeds threshold or, upon expiration of timer that was activated when queue was plugged.

NAME

probe::ioscheduler_trace.unplug_timer — Fires when unplug timer associated

SYNOPSIS

```
ioscheduler_trace.unplug_timer
```

VALUES

name

Name of the probe point

rq_queue

request queue

DESCRIPTION

with a request queue expires.

NAME

probe::ioblock.request — Fires whenever making a generic block I/O request.

SYNOPSIS

```
ioblock.request
```

VALUES

None

DESCRIPTION

name - name of the probe point ***devname*** - block device name ***ino*** - i-node number of the mapped file ***sector*** - beginning sector for the entire bio ***flags*** - see below BIO_UPTODATE 0 ok after I/O completion BIO_RW_BLOCK 1 RW_AHEAD set, and read/write would block BIO_EOF 2 out-out-bounds error BIO_SEG_VALID 3 nr_hw_seg valid BIO_CLONED 4 doesn't own data BIO_BOUNCED 5 bio is a bounce bio BIO_USER_MAPPED 6 contains user pages BIO_EOPNOTSUPP 7 not supported

rw - binary trace for read/write request ***vcnt*** - bio vector count which represents number of array element (page, offset, length) which make up this I/O request ***idx*** - offset into the bio vector array ***phys_segments*** - number of segments in this bio after physical address coalescing is performed ***hw_segments*** - number of segments after physical and DMA remapping hardware coalescing is

performed ***size*** - total size in bytes ***bdev*** - target block device ***bdev_contains*** - points to the device object which contains the partition (when bio structure represents a partition) ***p_start_sect*** - points to the start sector of the partition structure of the device

CONTEXT
The process makes block I/O request

NAME
probe::ioblock.end — Fires whenever a block I/O transfer is complete.

SYNOPSIS

```
ioblock.end
```

VALUES
None

DESCRIPTION
name - name of the probe point ***devname*** - block device name ***ino*** - i-node number of the mapped file ***bytes_done*** - number of bytes transferred ***sector*** - beginning sector for the entire bio ***flags*** - see below BIO_UPTODATE 0 ok after I/O completion BIO_RW_BLOCK 1 RW_AHEAD set, and read/write would block BIO_EOF 2 out-out-bounds error BIO_SEG_VALID 3 nr_hw_seg valid BIO_CLONED 4 doesn't own data BIO_BOUNCED 5 bio is a bounce bio BIO_USER_MAPPED 6 contains user pages BIO_EOPNOTSUPP 7 not supported ***error*** - 0 on success ***rw*** - binary trace for read/write request ***vcnt*** - bio vector count which represents number of array element (page, offset, length) which makes up this I/O request ***idx*** - offset into the bio vector array ***phys_segments*** - number of segments in this bio after physical address coalescing is performed. ***hw_segments*** - number of segments after physical and DMA remapping hardware coalescing is performed ***size*** - total size in bytes

CONTEXT
The process signals the transfer is done.

NAME
probe::ioblock_trace.bounce — Fires whenever a buffer bounce is needed for at least one page of a block IO request.

SYNOPSIS

```
ioblock_trace.bounce
```

VALUES
None

DESCRIPTION
name - name of the probe point ***q*** - request queue on which this bio was queued. ***devname*** - device for which a buffer bounce was needed. ***ino*** - i-node number of the mapped file ***bytes_done*** - number of bytes transferred ***sector*** - beginning sector for the entire bio ***flags*** - see below BIO_UPTODATE 0

ok after I/O completion BIO_RW_BLOCK 1 RW_AHEAD set, and read/write would block BIO_EOF 2 out-out-bounds error BIO_SEG_VALID 3 nr_hw_seg valid BIO_CLONED 4 doesn't own data BIO_BOUNCED 5 bio is a bounce bio BIO_USER_MAPPED 6 contains user pages BIO_EOPNOTSUPP 7 not supported ***rw*** - binary trace for read/write request ***vcnt*** - bio vector count which represents number of array element (page, offset, length) which makes up this I/O request ***idx*** - offset into the bio vector array ***phys_segments*** - number of segments in this bio after physical address coalescing is performed. ***size*** - total size in bytes ***bdev*** - target block device ***bdev_contains*** - points to the device object which contains the partition (when bio structure represents a partition) ***p_start_sect*** - points to the start sector of the partition structure of the device

CONTEXT

The process creating a block IO request.

NAME

probe::ioblock_trace.request — Fires just as a generic block I/O request is created for a bio.

SYNOPSIS

```
ioblock_trace.request
```

VALUES

None

DESCRIPTION

name - name of the probe point ***q*** - request queue on which this bio was queued. ***devname*** - block device name ***ino*** - i-node number of the mapped file ***bytes_done*** - number of bytes transferred ***sector*** - beginning sector for the entire bio ***flags*** - see below BIO_UPTODATE 0 ok after I/O completion BIO_RW_BLOCK 1 RW_AHEAD set, and read/write would block BIO_EOF 2 out-out-bounds error BIO_SEG_VALID 3 nr_hw_seg valid BIO_CLONED 4 doesn't own data BIO_BOUNCED 5 bio is a bounce bio BIO_USER_MAPPED 6 contains user pages BIO_EOPNOTSUPP 7 not supported

rw - binary trace for read/write request ***vcnt*** - bio vector count which represents number of array element (page, offset, length) which make up this I/O request ***idx*** - offset into the bio vector array ***phys_segments*** - number of segments in this bio after physical address coalescing is performed. ***size*** - total size in bytes ***bdev*** - target block device ***bdev_contains*** - points to the device object which contains the partition (when bio structure represents a partition) ***p_start_sect*** - points to the start sector of the partition structure of the device

CONTEXT

The process makes block I/O request

NAME

probe::ioblock_trace.end — Fires whenever a block I/O transfer is complete.

SYNOPSIS

```
ioblock_trace.end
```

VALUES
None

DESCRIPTION
name - name of the probe point ***q*** - request queue on which this bio was queued. ***devname*** - block device name ***ino*** - i-node number of the mapped file ***bytes_done*** - number of bytes transferred ***sector*** - beginning sector for the entire bio ***flags*** - see below BIO_UPTODATE 0 ok after I/O completion BIO_RW_BLOCK 1 RW_AHEAD set, and read/write would block BIO_EOF 2 out-out-bounds error BIO_SEG_VALID 3 nr_hw_seg valid BIO_CLONED 4 doesn't own data BIO_BOUNCED 5 bio is a bounce bio BIO_USER_MAPPED 6 contains user pages BIO_EOPNOTSUPP 7 not supported

rw - binary trace for read/write request ***vcnt*** - bio vector count which represents number of array element (page, offset, length) which makes up this I/O request ***idx*** - offset into the bio vector array ***phys_segments*** - number of segments in this bio after physical address coalescing is performed. ***size*** - total size in bytes ***bdev*** - target block device ***bdev_contains*** - points to the device object which contains the partition (when bio structure represents a partition) ***p_start_sect*** - points to the start sector of the partition structure of the device

CONTEXT
The process signals the transfer is done.

CHAPTER 9. SCSI TAPSET

This family of probe points is used to probe SCSI activities. It contains the following probe points:

NAME

probe::scsi.ioentry — Prepares a SCSI mid-layer request

SYNOPSIS

```
scsi.ioentry
```

VALUES

disk_major

The major number of the disk (-1 if no information)

device_state_str

The current state of the device, as a string

device_state

The current state of the device

req_addr

The current struct request pointer, as a number

disk_minor

The minor number of the disk (-1 if no information)

NAME

probe::scsi.iodispatching — SCSI mid-layer dispatched low-level SCSI command

SYNOPSIS

```
scsi.iodispatching
```

VALUES

device_state_str

The current state of the device, as a string

dev_id

The scsi device id

channel

The channel number

data_direction

The data_direction specifies whether this command is from/to the device 0 (DMA_BIDIRECTIONAL), 1 (DMA_TO_DEVICE), 2 (DMA_FROM_DEVICE), 3 (DMA_NONE)

lun

The lun number

request_bufflen

The request buffer length

host_no

The host number

device_state

The current state of the device

data_direction_str

Data direction, as a string

req_addr

The current struct request pointer, as a number

request_buffer

The request buffer address

NAME

probe::scsi.iodone — SCSI command completed by low level driver and enqueued into the done queue.

SYNOPSIS

```
scsi.iodone
```

VALUES

device_state_str

The current state of the device, as a string

dev_id

The scsi device id

channel

The channel number

data_direction

The data_direction specifies whether this command is from/to the device.

lun

The lun number

host_no

The host number

data_direction_str

Data direction, as a string

device_state

The current state of the device

scsi_timer_pending

1 if a timer is pending on this request

req_addr

The current struct request pointer, as a number

NAME

probe::scsi.iocompleted — SCSI mid-layer running the completion processing for block device I/O requests

SYNOPSIS

```
scsi.iocompleted
```

VALUES

device_state_str

The current state of the device, as a string

dev_id

The scsi device id

channel

The channel number

data_direction

The data_direction specifies whether this command is from/to the device

lun

The lun number

host_no

The host number

data_direction_str

Data direction, as a string

device_state

The current state of the device

req_addr

The current struct request pointer, as a number

goodbytes

The bytes completed

NAME

probe::scsi.ioexecute — Create mid-layer SCSI request and wait for the result

SYNOPSIS

```
scsi.ioexecute
```

VALUES

retries

Number of times to retry request

device_state_str

The current state of the device, as a string

dev_id

The scsi device id

channel

The channel number

data_direction

The data_direction specifies whether this command is from/to the device.

lun

The lun number

timeout

Request timeout in seconds

request_bufflen

The data buffer buffer length

host_no

The host number

data_direction_str

Data direction, as a string

device_state

The current state of the device

request_buffer

The data buffer address

NAME

probe::scsi.set_state — Order SCSI device state change

SYNOPSIS

```
scsi.set_state
```

VALUES

state_str

The new state of the device, as a string

dev_id

The scsi device id

channel

The channel number

state

The new state of the device

old_state_str

The current state of the device, as a string

lun

The lun number

old_state

The current state of the device

host_no

The host number

CHAPTER 10. TTY TAPSET

This family of probe points is used to probe TTY (Teletype) activities. It contains the following probe points:

NAME

probe::tty.open — Called when a tty is opened

SYNOPSIS

```
tty.open
```

VALUES

inode_state

the inode state

file_name

the file name

file_mode

the file mode

file_flags

the file flags

inode_number

the inode number

inode_flags

the inode flags

NAME

probe::tty.release — Called when the tty is closed

SYNOPSIS

```
tty.release
```

VALUES

inode_state

the inode state

file_name

the file name

file_mode

the file mode

file_flags

the file flags

inode_number

the inode number

inode_flags

the inode flags

NAME

probe::tty.resize — Called when a terminal resize happens

SYNOPSIS

```
tty.resize
```

VALUES

new_ypixel

the new ypixel value

old_col

the old col value

old_xpixel

the old xpixel

old_ypixel

the old ypixel

name

the tty name

old_row

the old row value

new_row

the new row value

new_xpixel

the new xpixel value

new_col

the new col value

NAME

probe::tty.ioctl — called when a ioctl is request to the tty

SYNOPSIS

```
tty.ioctl
```

VALUES

cmd

the ioctl command

arg

the ioctl argument

name

the file name

NAME

probe::tty.init — Called when a tty is being initalized

SYNOPSIS

```
tty.init
```

VALUES

driver_name

the driver name

name

the driver .dev_name name

module

the module name

NAME

probe::tty.register — Called when a tty device is registred

SYNOPSIS

```
tty.register
```

VALUES

driver_name

the driver name

name

the driver .dev_name name

index

the tty index requested

module

the module name

NAME

probe::tty.unregister — Called when a tty device is being unregistered

SYNOPSIS

```
tty.unregister
```

VALUES

driver_name

the driver name

name

the driver .dev_name name

index

the tty index requested

module

the module name

NAME

probe::tty.poll — Called when a tty device is being polled

SYNOPSIS

```
tty.poll
```

VALUES

file_name

the tty file name

wait_key

the wait queue key

NAME

probe::tty.receive — called when a tty receives a message

SYNOPSIS

```
tty.receive
```

VALUES

driver_name

the driver name

count

The amount of characters received

name

the name of the module file

fp

The flag buffer

cp

the buffer that was received

index

The tty Index

id

the tty id

NAME

probe::tty.write — write to the tty line

SYNOPSIS

```
tty.write
```

VALUES

driver_name

the driver name

buffer

the buffer that will be written

file_name

the file name lreated to the tty

nr

The amount of characters

NAME

probe::tty.read — called when a tty line will be read

SYNOPSIS

```
tty.read
```

VALUES

driver_name

the driver name

buffer

the buffer that will receive the characters

file_name

the file name lreated to the tty

nr

The amount of characters to be read

CHAPTER 11. NETWORKING TAPSET

This family of probe points is used to probe the activities of the network device and protocol layers.

NAME

probe::netdev.receive — Data received from network device.

SYNOPSIS

```
netdev.receive
```

VALUES

protocol

Protocol of received packet.

dev_name

The name of the device. e.g: eth0, ath1.

length

The length of the receiving buffer.

NAME

probe::netdev.transmit — Network device transmitting buffer

SYNOPSIS

```
netdev.transmit
```

VALUES

protocol

The protocol of this packet(defined in include/linux/if_ether.h).

dev_name

The name of the device. e.g: eth0, ath1.

length

The length of the transmit buffer.

truesize

The size of the data to be transmitted.

NAME

probe::netdev.change_mtu — Called when the netdev MTU is changed

SYNOPSIS

```
netdev.change_mtu
```

VALUES

dev_name

The device that will have the MTU changed

new_mtu

The new MTU

old_mtu

The current MTU

NAME

probe::netdev.open — Called when the device is opened

SYNOPSIS

```
netdev.open
```

VALUES

dev_name

The device that is going to be opened

NAME

probe::netdev.close — Called when the device is closed

SYNOPSIS

```
netdev.close
```

VALUES

dev_name

The device that is going to be closed

NAME

probe::netdev.hard_transmit — Called when the devices is going to TX (hard)

SYNOPSIS

```
netdev.hard_transmit
```

VALUES

protocol

The protocol used in the transmission

dev_name

The device scheduled to transmit

length

The length of the transmit buffer.

truesize

The size of the data to be transmitted.

NAME

probe::netdev.rx — Called when the device is going to receive a packet

SYNOPSIS

```
netdev.rx
```

VALUES

protocol

The packet protocol

dev_name

The device received the packet

NAME

probe::netdev.change_rx_flag — Called when the device RX flag will be changed

SYNOPSIS

```
netdev.change_rx_flag
```

VALUES

dev_name

The device that will be changed

flags

The new flags

NAME

probe::netdev.set_promiscuity — Called when the device enters/leaves promiscuity

SYNOPSIS

```
netdev.set_promiscuity
```

VALUES

dev_name

The device that is entering/leaving promiscuity mode

enable

If the device is entering promiscuity mode

inc

Count the number of promiscuity openers

disable

If the device is leaving promiscuity mode

NAME

probe::netdev.ioctl — Called when the device suffers an IOCTL

SYNOPSIS

```
netdev.ioctl
```

VALUES

cmd

The IOCTL request

arg

The IOCTL argument (usually the netdev interface)

NAME

probe::netdev.register — Called when the device is registered

SYNOPSIS

```
netdev.register
```

VALUES

dev_name

The device that is going to be registered

NAME

probe::netdev.unregister — Called when the device is being unregistered

SYNOPSIS

```
netdev.unregister
```

VALUES

dev_name

The device that is going to be unregistered

NAME

probe::netdev.get_stats — Called when someone asks the device statistics

SYNOPSIS

```
netdev.get_stats
```

VALUES

dev_name

The device that is going to provide the statistics

NAME

probe::netdev.change_mac — Called when the netdev_name has the MAC changed

SYNOPSIS

```
netdev.change_mac
```

VALUES

dev_name

The device that will have the MTU changed

new_mac

The new MAC address

mac_len

The MAC length

old_mac

The current MAC address

NAME

probe::tcp.sendmsg — Sending a tcp message

SYNOPSIS

```
tcp.sendmsg
```

VALUES

name

Name of this probe

size

Number of bytes to send

sock

Network socket

CONTEXT

The process which sends a tcp message

NAME
probe::tcp.sendmsg.return — Sending TCP message is done

SYNOPSIS

```
tcp.sendmsg.return
```

VALUES

name

Name of this probe

size

Number of bytes sent or error code if an error occurred.

CONTEXT
The process which sends a tcp message

NAME
probe::tcp.recvmsg — Receiving TCP message

SYNOPSIS

```
tcp.recvmsg
```

VALUES

saddr

A string representing the source IP address

daddr

A string representing the destination IP address

name

Name of this probe

sport

TCP source port

dport

TCP destination port

size

Number of bytes to be received

sock

Network socket

CONTEXT

The process which receives a tcp message

NAME

probe::tcp.recvmsg.return — Receiving TCP message complete

SYNOPSIS

```
tcp.recvmsg.return
```

VALUES

saddr

A string representing the source IP address

daddr

A string representing the destination IP address

name

Name of this probe

sport

TCP source port

dport

TCP destination port

size

Number of bytes received or error code if an error occurred.

CONTEXT

The process which receives a tcp message

NAME

probe::tcp.disconnect — TCP socket disconnection

SYNOPSIS

```
tcp.disconnect
```

VALUES

saddr

A string representing the source IP address

daddr

A string representing the destination IP address

flags

TCP flags (e.g. FIN, etc)

name

Name of this probe

sport

TCP source port

dport

TCP destination port

sock

Network socket

CONTEXT

The process which disconnects tcp

NAME

probe::tcp.disconnect.return — TCP socket disconnection complete

SYNOPSIS

```
tcp.disconnect.return
```

VALUES

ret

Error code (0: no error)

name

Name of this probe

CONTEXT

The process which disconnects tcp

NAME

probe::tcp.setsockopt — Call to **setsockopt**

SYNOPSIS

```
tcp.setsockopt
```

VALUES

optstr

Resolves optname to a human-readable format

level

The level at which the socket options will be manipulated

optlen

Used to access values for **setsockopt**

name

Name of this probe

optname

TCP socket options (e.g. TCP_NODELAY, TCP_MAXSEG, etc)

sock

Network socket

CONTEXT

The process which calls setsockopt

NAME

probe::tcp.setsockopt.return — Return from **setsockopt**

SYNOPSIS

```
tcp.setsockopt.return
```

VALUES

ret

Error code (0: no error)

name

Name of this probe

CONTEXT

The process which calls setsockopt

NAME

probe::tcp.receive — Called when a TCP packet is received

SYNOPSIS

```
tcp.receive
```

VALUES

urg

TCP URG flag

protocol

Packet protocol from driver

psh

TCP PSH flag

name

Name of the probe point

rst

TCP RST flag

dport

TCP destination port

saddr

A string representing the source IP address

daddr

A string representing the destination IP address

ack

TCP ACK flag

fin

TCP FIN flag

syn

TCP SYN flag

sport

TCP source port

iphdr

IP header address

NAME

probe::udp.sendmsg — Fires whenever a process sends a UDP message

SYNOPSIS

```
udp.sendmsg
```

VALUES

name

The name of this probe

size

Number of bytes sent by the process

sock

Network socket used by the process

CONTEXT

The process which sent a UDP message

NAME

probe::udp.sendmsg.return — Fires whenever an attempt to send a UDP message is completed

SYNOPSIS

```
udp.sendmsg.return
```

VALUES

name

The name of this probe

size

Number of bytes sent by the process

CONTEXT
The process which sent a UDP message

NAME
probe::udp.recvmsg — Fires whenever a UDP message is received

SYNOPSIS

```
udp.recvmsg
```

VALUES

name

The name of this probe

size

Number of bytes received by the process

sock

Network socket used by the process

CONTEXT
The process which received a UDP message

NAME
probe::udp.recvmsg.return — Fires whenever an attempt to receive a UDP message received is completed

SYNOPSIS

```
udp.recvmsg.return
```

VALUES

name

The name of this probe

size

Number of bytes received by the process

CONTEXT
The process which received a UDP message

NAME
probe::udp.disconnect — Fires when a process requests for a UDP disconnection

SYNOPSIS

```
udp.disconnect
```

VALUES

flags

Flags (e.g. FIN, etc)

name

The name of this probe

sock

Network socket used by the process

CONTEXT
The process which requests a UDP disconnection

NAME
probe::udp.disconnect.return — UDP has been disconnected successfully

SYNOPSIS

```
udp.disconnect.return
```

VALUES

ret

Error code (0: no error)

name

The name of this probe

CONTEXT
The process which requested a UDP disconnection

NAME
function::ip_ntop — returns a string representation from an integer IP number

SYNOPSIS

```
function ip_ntop:string(addr:long)
```

ARGUMENTS

addr

the ip represented as an integer

CHAPTER 12. SOCKET TAPSET

This family of probe points is used to probe socket activities. It contains the following probe points:

NAME
probe::socket.send — Message sent on a socket.

SYNOPSIS

```
socket.send
```

VALUES

success

Was send successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message sent (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT
The message sender

NAME
probe::socket.receive — Message received on a socket.

SYNOPSIS

```
socket.receive
```

VALUES

success

Was send successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message received (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT

The message receiver

NAME

probe::socket.sendmsg — Message is currently being sent on a socket.

SYNOPSIS

```
socket.sendmsg
```

VALUES

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Message size in bytes

type

Socket type value

family

Protocol family value

CONTEXT

The message sender

DESCRIPTION

Fires at the beginning of sending a message on a socket via the **sock_sendmsg** function

---

NAME

probe::socket.sendmsg.return — Return from socket.sendmsg.

SYNOPSIS

```
socket.sendmsg.return
```

VALUES

success

Was send successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message sent (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT

The message sender.

DESCRIPTION

Fires at the conclusion of sending a message on a socket via the **sock_sendmsg** function

---

NAME

probe::socket.recvmsg — Message being received on socket

SYNOPSIS

```
socket.recvmsg
```

VALUES

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Message size in bytes

type

Socket type value

family

Protocol family value

CONTEXT
The message receiver.

DESCRIPTION
Fires at the beginning of receiving a message on a socket via the **sock_recvmsg** function

---

NAME
probe::socket.recvmsg.return — Return from Message being received on socket

SYNOPSIS

```
socket.recvmsg.return
```

VALUES

success

Was receive successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message received (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT
The message receiver.

DESCRIPTION

Fires at the conclusion of receiving a message on a socket via the **sock_recvmsg** function.

NAME

probe::socket.aio_write — Message send via **sock_aio_write**

SYNOPSIS

```
socket.aio_write
```

VALUES

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Message size in bytes

type

Socket type value

family

Protocol family value

CONTEXT

The message sender

DESCRIPTION

Fires at the beginning of sending a message on a socket via the **sock_aio_write** function

---

NAME

probe::socket.aio_write.return — Conclusion of message send via **sock_aio_write**

SYNOPSIS

```
socket.aio_write.return
```

VALUES

success

Was receive successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message received (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT

The message receiver.

DESCRIPTION

Fires at the conclusion of sending a message on a socket via the **sock_aio_write** function

---

NAME

probe::socket.aio_read — Receiving message via **sock_aio_read**

SYNOPSIS

```
socket.aio_read
```

VALUES

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Message size in bytes

type

Socket type value

family

Protocol family value

CONTEXT

The message sender

DESCRIPTION

Fires at the beginning of receiving a message on a socket via the **sock_aio_read** function

---

NAME

probe::socket.aio_read.return — Conclusion of message received via **sock_aio_read**

SYNOPSIS

```
socket.aio_read.return
```

VALUES

success

Was receive successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message received (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT

The message receiver.

DESCRIPTION

Fires at the conclusion of receiving a message on a socket via the **sock_aio_read** function

---

NAME

probe::socket.writev — Message sent via **socket_writev**

SYNOPSIS

```
socket.writev
```

VALUES

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Message size in bytes

type

Socket type value

family

Protocol family value

CONTEXT

The message sender

DESCRIPTION

Fires at the beginning of sending a message on a socket via the **sock_writev** function

---

NAME

probe::socket.writev.return — Conclusion of message sent via **socket_writev**

SYNOPSIS

```
socket.writev.return
```

VALUES

success

Was send successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message sent (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT
The message receiver.

DESCRIPTION
Fires at the conclusion of sending a message on a socket via the **sock_writev** function

---

NAME
probe::socket.readv — Receiving a message via **sock_readv**

SYNOPSIS

```
socket.readv
```

VALUES

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Message size in bytes

type

Socket type value

family

Protocol family value

CONTEXT
The message sender

DESCRIPTION
Fires at the beginning of receiving a message on a socket via the **sock_readv** function

---

NAME

probe::socket.readv.return — Conclusion of receiving a message via **sock_readv**

SYNOPSIS

```
socket.readv.return
```

VALUES

success

Was receive successful? (1 = yes, 0 = no)

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

size

Size of message received (in bytes) or error code if success = 0

type

Socket type value

family

Protocol family value

CONTEXT

The message receiver.

DESCRIPTION

Fires at the conclusion of receiving a message on a socket via the **sock_readv** function

---

NAME

probe::socket.create — Creation of a socket

SYNOPSIS

```
socket.create
```

VALUES

protocol

Protocol value

name

Name of this probe

requester

Requested by user process or the kernel (1 = kernel, 0 = user)

type

Socket type value

family

Protocol family value

CONTEXT

The requester (see requester variable)

DESCRIPTION

Fires at the beginning of creating a socket.

NAME

probe::socket.create.return — Return from Creation of a socket

SYNOPSIS

```
socket.create.return
```

VALUES

success

Was socket creation successful? (1 = yes, 0 = no)

protocol

Protocol value

err

Error code if success == 0

name

Name of this probe

requester

Requested by user process or the kernel (1 = kernel, 0 = user)

type

Socket type value

family

Protocol family value

CONTEXT

The requester (user process or kernel)

DESCRIPTION

Fires at the conclusion of creating a socket.

NAME

probe::socket.close — Close a socket

SYNOPSIS

```
socket.close
```

VALUES

protocol

Protocol value

flags

Socket flags value

name

Name of this probe

state

Socket state value

type

Socket type value

family

Protocol family value

CONTEXT

The requester (user process or kernel)

DESCRIPTION
Fires at the beginning of closing a socket.

NAME
probe::socket.close.return — Return from closing a socket

SYNOPSIS

```
socket.close.return
```

VALUES

name

Name of this probe

CONTEXT
The requester (user process or kernel)

DESCRIPTION
Fires at the conclusion of closing a socket.

NAME
function::sock_prot_num2str — Given a protocol number, return a string representation.

SYNOPSIS

```
function sock_prot_num2str:string(proto:long)
```

ARGUMENTS

proto

The protocol number.

NAME
function::sock_prot_str2num — Given a protocol name (string), return the corresponding protocol number.

SYNOPSIS

```
function sock_prot_str2num:long(proto:string)
```

ARGUMENTS

proto

The protocol name.

NAME

function::sock_fam_num2str — Given a protocol family number, return a string representation.

SYNOPSIS

```
function sock_fam_num2str:string(family:long)
```

ARGUMENTS

family

The family number.

NAME

function::sock_fam_str2num — Given a protocol family name (string), return the corresponding

SYNOPSIS

```
function sock_fam_str2num:long(family:string)
```

ARGUMENTS

family

The family name.

DESCRIPTION

protocol family number.

NAME

function::sock_state_num2str — Given a socket state number, return a string representation.

SYNOPSIS

```
function sock_state_num2str:string(state:long)
```

ARGUMENTS

state

The state number.

NAME

function::sock_state_str2num — Given a socket state string, return the corresponding state number.

SYNOPSIS

```
function sock_state_str2num:long(state:string)
```

ARGUMENTS

state

The state name.

CHAPTER 13. KERNEL PROCESS TAPSET

This family of probe points is used to probe process-related activities. It contains the following probe points:

NAME
probe::kprocess.create — Fires whenever a new process is successfully created

SYNOPSIS

```
kprocess.create
```

VALUES

new_pid

The PID of the newly created process

CONTEXT
Parent of the created process.

DESCRIPTION
Fires whenever a new process is successfully created, either as a result of fork (or one of its syscall variants), or a new kernel thread.

NAME
probe::kprocess.start — Starting new process

SYNOPSIS

```
kprocess.start
```

VALUES
None

CONTEXT
Newly created process.

DESCRIPTION
Fires immediately before a new process begins execution.

NAME
probe::kprocess.exec — Attempt to exec to a new program

SYNOPSIS

```
kprocess.exec
```

VALUES

filename

The path to the new executable

CONTEXT

The caller of exec.

DESCRIPTION

Fires whenever a process attempts to exec to a new program.

NAME

probe::kprocess.exec_complete — Return from exec to a new program

SYNOPSIS

```
kprocess.exec_complete
```

VALUES

success

A boolean indicating whether the exec was successful

errno

The error number resulting from the exec

CONTEXT

On success, the context of the new executable. On failure, remains in the context of the caller.

DESCRIPTION

Fires at the completion of an exec call.

NAME

probe::kprocess.exit — Exit from process

SYNOPSIS

```
kprocess.exit
```

VALUES

code

The exit code of the process

CONTEXT

The process which is terminating.

DESCRIPTION

Fires when a process terminates. This will always be followed by a kprocess.release, though the latter may be delayed if the process waits in a zombie state.

NAME

probe::kprocess.release — Process released

SYNOPSIS

```
kprocess.release
```

VALUES

pid

PID of the process being released

task

A task handle to the process being released

CONTEXT

The context of the parent, if it wanted notification of this process' termination, else the context of the process itself.

DESCRIPTION

Fires when a process is released from the kernel. This always follows a kprocess.exit, though it may be delayed somewhat if the process waits in a zombie state.

CHAPTER 14. SIGNAL TAPSET

This family of probe points is used to probe signal activities. It contains the following probe points:

NAME

probe::signal.send — Signal being sent to a process

SYNOPSIS

```
signal.send
```

VALUES

send2queue

Indicates whether the signal is sent to an existing sigqueue

name

The name of the function used to send out the signal

task

A task handle to the signal recipient

sinfo

The address of siginfo struct

si_code

Indicates the signal type

sig_name

A string representation of the signal

sig

The number of the signal

shared

Indicates whether the signal is shared by the thread group

sig_pid

The PID of the process receiving the signal

pid_name

The name of the signal recipient

CONTEXT

The signal's sender.

NAME
probe::signal.send.return — Signal being sent to a process completed

SYNOPSIS

```
signal.send.return
```

VALUES

retstr

The return value to either __group_send_sig_info, specific_send_sig_info, or send_sigqueue

send2queue

Indicates whether the sent signal was sent to an existing sigqueue

name

The name of the function used to send out the signal

shared

Indicates whether the sent signal is shared by the thread group.

CONTEXT
The signal's sender. (correct?)

DESCRIPTION
Possible __group_send_sig_info and specific_send_sig_info return values are as follows;

0 -- The signal is sucessfully sent to a process,

WHICH MEANS THAT
(1) the signal was ignored by the receiving process, (2) this is a non-RT signal and the system already has one queued, and (3) the signal was successfully added to the sigqueue of the receiving process.

-EAGAIN -- The sigqueue of the receiving process is overflowing, the signal was RT, and the signal was sent by a user using something other than **`kill`**.

Possible send_group_sigqueue and send_sigqueue return values are as follows;

0 -- The signal was either sucessfully added into the sigqueue of the receiving process, or a SI_TIMER entry is already queued (in which case, the overrun count will be simply incremented).

1 -- The signal was ignored by the receiving process.

-1 -- (send_sigqueue only) The task was marked exiting, allowing * posix_timer_event to redirect it to the group leader.

NAME

probe::signal.checkperm — Check being performed on a sent signal

SYNOPSIS

```
signal.checkperm
```

VALUES

name

Name of the probe point

task

A task handle to the signal recipient

sinfo

The address of the siginfo structure

si_code

Indicates the signal type

sig_name

A string representation of the signal

sig

The number of the signal

pid_name

Name of the process receiving the signal

sig_pid

The PID of the process receiving the signal

NAME

probe::signal.checkperm.return — Check performed on a sent signal completed

SYNOPSIS

```
signal.checkperm.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.wakeup — Sleeping process being wakened for signal

SYNOPSIS

```
signal.wakeup
```

VALUES

resume

Indicates whether to wake up a task in a STOPPED or TRACED state

state_mask

A string representation indicating the mask of task states to wake. Possible values are TASK_INTERRUPTIBLE, TASK_STOPPED, TASK_TRACED, and TASK_INTERRUPTIBLE.

pid_name

Name of the process to wake

sig_pid

The PID of the process to wake

NAME

probe::signal.check_ignored — Checking to see signal is ignored

SYNOPSIS

```
signal.check_ignored
```

VALUES

sig_name

A string representation of the signal

sig

The number of the signal

pid_name

Name of the process receiving the signal

sig_pid

The PID of the process receiving the signal

NAME

probe::signal.check_ignored.return — Check to see signal is ignored completed

SYNOPSIS

```
signal.check_ignored.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.force_segv — Forcing send of SIGSEGV

SYNOPSIS

```
signal.force_segv
```

VALUES

name

Name of the probe point

sig_name

A string representation of the signal

sig

The number of the signal

pid_name

Name of the process receiving the signal

sig_pid

The PID of the process receiving the signal

NAME

probe::signal.force_segv.return — Forcing send of SIGSEGV complete

SYNOPSIS

```
signal.force_segv.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.syskill — Sending kill signal to a process

SYNOPSIS

```
signal.syskill
```

VALUES

name

Name of the probe point

sig_name

A string representation of the signal

sig

The specific signal sent to the process

pid_name

The name of the signal recipient

sig_pid

The PID of the process receiving the signal

NAME

probe::signal.syskill.return — Sending kill signal completed

SYNOPSIS

```
signal.syskill.return
```

VALUES

None

NAME

probe::signal.sys_tkill — Sending a kill signal to a thread

SYNOPSIS

```
signal.sys_tkill
```

VALUES

name

Name of the probe point

sig_name

A string representation of the signal

sig

The specific signal sent to the process

pid_name

The name of the signal recipient

sig_pid

The PID of the process receiving the kill signal

DESCRIPTION

The tkill call is analogous to kill(2), except that it also allows a process within a specific thread group to be targeted. Such processes are targeted through their unique thread IDs (TID).

NAME

probe::signal.systkill.return — Sending kill signal to a thread completed

SYNOPSIS

```
signal.systkill.return
```

VALUES

retstr

The return value to either __group_send_sig_info,

name

Name of the probe point

NAME

probe::signal.sys_tgkill — Sending kill signal to a thread group

SYNOPSIS

```
signal.sys_tgkill
```

VALUES

name

Name of the probe point

sig_name

A string representation of the signal

sig

The specific kill signal sent to the process

tgid

The thread group ID of the thread receiving the kill signal

pid_name

The name of the signal recipient

sig_pid

The PID of the thread receiving the kill signal

DESCRIPTION

The tgkill call is similar to tkill, except that it also allows the caller to specify the thread group ID of the thread to be signalled. This protects against TID reuse.

NAME

probe::signal.sys_tgkill.return — Sending kill signal to a thread group completed

SYNOPSIS

```
signal.sys_tgkill.return
```

VALUES

retstr

The return value to either __group_send_sig_info,

name

Name of the probe point

NAME

probe::signal.send_sig_queue — Queuing a signal to a process

SYNOPSIS

```
signal.send_sig_queue
```

VALUES

sigqueue_addr

The address of the signal queue

name

Name of the probe point

sig_name

A string representation of the signal

sig

The queued signal

pid_name

Name of the process to which the signal is queued

sig_pid

The PID of the process to which the signal is queued

NAME

probe::signal.send_sig_queue.return — Queuing a signal to a process completed

SYNOPSIS

```
signal.send_sig_queue.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.pending — Examining pending signal

SYNOPSIS

```
signal.pending
```

VALUES

name

Name of the probe point

sigset_size

The size of the user-space signal set

sigset_add

The address of the user-space signal set (sigset_t)

DESCRIPTION

This probe is used to examine a set of signals pending for delivery to a specific thread. This normally occurs when the do_sigpending kernel function is executed.

NAME

probe::signal.pending.return — Examination of pending signal completed

SYNOPSIS

```
signal.pending.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.handle — Signal handler being invoked

SYNOPSIS

```
signal.handle
```

VALUES

regs

The address of the kernel-mode stack area

sig_code

The si_code value of the siginfo signal

name

Name of the probe point

sig_mode

Indicates whether the signal was a user-mode or kernel-mode signal

sinfo

The address of the siginfo table

sig_name

A string representation of the signal

oldset_addr

The address of the bitmask array of blocked signals

sig

The signal number that invoked the signal handler

ka_addr

The address of the k_sigaction table associated with the signal

---

NAME

probe::signal.handle.return — Signal handler invocation completed

SYNOPSIS

```
signal.handle.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.do_action — Examining or changing a signal action

SYNOPSIS

```
signal.do_action
```

VALUES

sa_mask

The new mask of the signal

name

Name of the probe point

sig_name

A string representation of the signal

oldsigact_addr

The address of the old sigaction struct associated with the signal

sig

The signal to be examined/changed

sa_handler

The new handler of the signal

sigact_addr

The address of the new sigaction struct associated with the signal

---

NAME

probe::signal.do_action.return — Examining or changing a signal action completed

SYNOPSIS

```
signal.do_action.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.procmask — Examining or changing blocked signals

SYNOPSIS

```
signal.procmask
```

VALUES

how

Indicates how to change the blocked signals; possible values are SIG_BLOCK=0 (for blocking signals), SIG_UNBLOCK=1 (for unblocking signals), and SIG_SETMASK=2 for setting the signal mask.

name

Name of the probe point

oldsigset_addr

The old address of the signal set (sigset_t)

sigset

The actual value to be set for sigset_t (correct?)

sigset_addr

The address of the signal set (sigset_t) to be implemented

NAME

probe::signal.procmask.return — Examining or changing blocked signals completed

SYNOPSIS

```
signal.procmask.return
```

VALUES

retstr

Return value as a string

name

Name of the probe point

NAME

probe::signal.flush — Flushing all pending signals for a task

SYNOPSIS

```
signal.flush
```

VALUES

name

Name of the probe point

task

The task handler of the process performing the flush

pid_name

The name of the process associated with the task performing the flush

sig_pid

The PID of the process associated with the task performing the flush

CHAPTER 15. DIRECTORY-ENTRY (DENTRY) TAPSET

This family of functions is used to map kernel VFS directory entry pointers to file or full path names.

NAME

function::d_name — get the dirent name

SYNOPSIS

```
function d_name:string(dentry:long)
```

ARGUMENTS

dentry

Pointer to dentry.

DESCRIPTION

Returns the dirent name (path basename).

NAME

function::reverse_path_walk — get the full dirent path

SYNOPSIS

```
function reverse_path_walk:string(dentry:long)
```

ARGUMENTS

dentry

Pointer to dentry.

DESCRIPTION

Returns the path name (partial path to mount point).

NAME

function::task_dentry_path — get the full dentry path

SYNOPSIS

```
function task_dentry_path:string(task:long,dentry:long,vfsmnt:long)
```

ARGUMENTS

task

task

task_struct pointer.

dentry

direntry pointer.

vfsmnt

vfsmnt pointer.

DESCRIPTION

Returns the full dirent name (full path to the root), like the kernel d_path function.

NAME

function::d_path — get the full nameidata path

SYNOPSIS

```
function d_path:string(nd:long)
```

ARGUMENTS

nd

Pointer to nameidata.

DESCRIPTION

Returns the full dirent name (full path to the root), like the kernel d_path function.

CHAPTER 16. LOGGING TAPSET

This family of functions is used to send simple message strings to various destinations.

NAME

function::log — Send a line to the common trace buffer.

SYNOPSIS

```
function log(msg:string)
```

ARGUMENTS

msg

The formatted message string.

GENERAL SYNTAX

log(msg:string)

DESCRIPTION

This function logs data. log sends the message immediately to staprun and to the bulk transport (relayfs) if it is being used. If the last character given is not a newline, then one is added. This function is not as effecient as printf and should be used only for urgent messages.

NAME

function::warn — Send a line to the warning stream.

SYNOPSIS

```
function warn(msg:string)
```

ARGUMENTS

msg

The formatted message string.

GENERAL SYNTAX

warn (msg:string)

DESCRIPTION

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

NAME
function::exit — Start shutting down probing script.

SYNOPSIS

```
function exit()
```

ARGUMENTS
None

GENERAL SYNTAX
exit

DESCRIPTION
This only enqueues a request to start shutting down the script. New probes will not fire (except "end" probes), but all currently running ones may complete their work.

NAME
function::error — Send an error message.

SYNOPSIS

```
function error(msg:string)
```

ARGUMENTS

msg

The formatted message string.

DESCRIPTION
An implicit end-of-line is added. staprun prepends the string "ERROR:". Sending an error message aborts the currently running probe. Depending on the MAXERRORS parameter, it may trigger an **exit**.

NAME
function::ftrace — Send a message to the ftrace ring-buffer.

SYNOPSIS

```
function ftrace(msg:string)
```

ARGUMENTS

msg

The formatted message string.

DESCRIPTION

If the ftrace ring-buffer is configured & available, see /debugfs/tracing/trace for the message. Otherwise, the message may be quietly dropped. An implicit end-of-line is added.

CHAPTER 17. RANDOM FUNCTIONS TAPSET

These functions deal with random number generation.

NAME

function::randint — Return a random number between [0,n)

SYNOPSIS

```
function randint:long(n:long)
```

ARGUMENTS

n

Number past upper limit of range, not larger than 2**20.

CHAPTER 18. STRING AND DATA RETRIEVING FUNCTIONS TAPSET

Functions to retrieve strings and other primitive types from the kernel or a user space programs based on addresses. All strings are of a maximum length given by MAXSTRINGLEN.

NAME

function::kernel_string — Retrieves string from kernel memory.

SYNOPSIS

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

ARGUMENTS

addr

The kernel address to retrieve the string from.

GENERAL SYNTAX

kernel_string:string(addr:long)

DESCRIPTION

This function returns the null terminated C string from a given kernel memory address. Reports an error on string copy fault.

NAME

function::kernel_string2 — Retrieves string from kernel memory with alternative error string.

SYNOPSIS

```
function kernel_string2:string(addr:long,err_msg:string)
```

ARGUMENTS

addr

The kernel address to retrieve the string from.

err_msg

The error message to return when data isn't available.

GENERAL SYNTAX

kernel_string2:string(addr:long, err_msg:string)

DESCRIPTION

This function returns the null terminated C string from a given kernel memory address. Reports the given error message on string copy fault.

NAME

function::kernel_string_n — Retrieves string of given length from kernel memory.

SYNOPSIS

```
function kernel_string_n:string(addr:long,n:long)
```

ARGUMENTS

addr

The kernel address to retrieve the string from.

n

The maximum length of the string (if not null terminated).

GENERAL SYNTAX

kernel_string_n:string(addr:long, n:long)

DESCRIPTION

Returns the C string of a maximum given length from a given kernel memory address. Reports an error on string copy fault.

NAME

function::kernel_long — Retrieves a long value stored in kernel memory.

SYNOPSIS

```
function kernel_long:long(addr:long)
```

ARGUMENTS

addr

The kernel address to retrieve the long from.

GENERAL SYNTAX

kernel_long:long(addr:long)

DESCRIPTION

Returns the long value from a given kernel memory address. Reports an error when reading from the given address fails.

NAME
function::kernel_int — Retrieves an int value stored in kernel memory.

SYNOPSIS

```
function kernel_int:long(addr:long)
```

ARGUMENTS

addr

The kernel address to retrieve the int from.

DESCRIPTION
Returns the int value from a given kernel memory address. Reports an error when reading from the given address fails.

NAME
function::kernel_short — Retrieves a short value stored in kernel memory.

SYNOPSIS

```
function kernel_short:long(addr:long)
```

ARGUMENTS

addr

The kernel address to retrieve the short from.

GENERAL SYNTAX
kernel_short:long(addr:long)

DESCRIPTION
Returns the short value from a given kernel memory address. Reports an error when reading from the given address fails.

NAME
function::kernel_char — Retrieves a char value stored in kernel memory.

SYNOPSIS

```
function kernel_char:long(addr:long)
```

ARGUMENTS

addr

The kernel address to retrieve the char from.

GENERAL SYNTAX

kernel_char:long(addr:long)

DESCRIPTION

Returns the char value from a given kernel memory address. Reports an error when reading from the given address fails.

NAME

function::kernel_pointer — Retrieves a pointer value stored in kernel memory.

SYNOPSIS

```
function kernel_pointer:long(addr:long)
```

ARGUMENTS

addr

The kernel address to retrieve the pointer from.

GENERAL SYNTAX

kernel_pointer:long(addr:long)

DESCRIPTION

Returns the pointer value from a given kernel memory address. Reports an error when reading from the given address fails.

NAME

function::user_string — Retrieves string from user space.

SYNOPSIS

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

ARGUMENTS

addr

The user space address to retrieve the string from.

GENERAL SYNTAX

user_string:string(addr:long)

DESCRIPTION

Returns the null terminated C string from a given user space memory address. Reports “<unknown>” on the rare cases when userspace data is not accessible.

NAME

function::user_string2 — Retrieves string from user space with alternative error string.

SYNOPSIS

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

ARGUMENTS

addr

The user space address to retrieve the string from.

err_msg

The error message to return when data isn't available.

GENERAL SYNTAX

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

DESCRIPTION

Returns the null terminated C string from a given user space memory address. Reports the given error message on the rare cases when userspace data is not accessible.

NAME

function::user_string_warn — Retrieves string from user space.

SYNOPSIS

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

ARGUMENTS

addr

The user space address to retrieve the string from.

GENERAL SYNTAX

user_string_warn:string(addr:long)

DESCRIPTION

Returns the null terminated C string from a given user space memory address. Reports “<unknown>” on the rare cases when userspace data is not accessible and warns (but does not abort) about the failure.

NAME

function::user_string_quoted — Retrieves and quotes string from user space.

SYNOPSIS

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

ARGUMENTS

addr

The user space address to retrieve the string from.

GENERAL SYNTAX

user_string_quoted:string(addr:long)

DESCRIPTION

Returns the null terminated C string from a given user space memory address where any ASCII characters that are not printable are replaced by the corresponding escape sequence in the returned string. Reports “NULL” for address zero. Returns “<unknown>” on the rare cases when userspace data is not accessible at the given address.

NAME

function::user_string_n — Retrieves string of given length from user space.

SYNOPSIS

```
function user_string_n:string(addr:long,n:long)
```

ARGUMENTS

addr

The user space address to retrieve the string from.

n

The maximum length of the string (if not null terminated).

GENERAL SYNTAX

user_string_n:string(addr:long, n:long)

DESCRIPTION

Returns the C string of a maximum given length from a given user space address. Returns "<unknown>" on the rare cases when userspace data is not accessible at the given address.

NAME

function::user_string_n2 — Retrieves string of given length from user space.

SYNOPSIS

```
function user_string_n2:string(addr:long,n:long,err_msg:string)
```

ARGUMENTS

addr

The user space address to retrieve the string from.

n

The maximum length of the string (if not null terminated).

err_msg

The error message to return when data isn't available.

GENERAL SYNTAX

user_string_n2:string(addr:long, n:long, err_msg:string)

DESCRIPTION

Returns the C string of a maximum given length from a given user space address. Returns the given error message string on the rare cases when userspace data is not accessible at the given address.

NAME

function::user_string_n_warn — Retrieves string from user space.

SYNOPSIS

```
function user_string_n_warn:string(addr:long,n:long)
```

ARGUMENTS

addr

The user space address to retrieve the string from.

n

The maximum length of the string (if not null terminated).

GENERAL SYNTAX
user_string_n_warn:string(addr:long, n:long)

DESCRIPTION
Returns up to n characters of a C string from a given user space memory address. Reports "<unknown>" on the rare cases when userspace data is not accessible and warns (but does not abort) about the failure.

NAME
function::user_string_n_quoted — Retrieves and quotes string from user space.

SYNOPSIS

```
function user_string_n_quoted:string(addr:long,n:long)
```

ARGUMENTS

addr

The user space address to retrieve the string from.

n

The maximum length of the string (if not null terminated).

GENERAL SYNTAX
user_string_n_quoted:string(addr:long, n:long)

DESCRIPTION
Returns up to n characters of a C string from the given user space memory address where any ASCII characters that are not printable are replaced by the corresponding escape sequence in the returned string. Reports "NULL" for address zero. Returns "<unknown>" on the rare cases when userspace data is not accessible at the given address.

NAME
function::user_short — Retrieves a short value stored in user space.

SYNOPSIS

```
function user_short:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the short from.

GENERAL SYNTAX
user_short:long(addr:long)

DESCRIPTION
Returns the short value from a given user space address. Returns zero when user space data is not accessible.

NAME
function::user_short_warn — Retrieves a short value stored in user space.

SYNOPSIS

```
function user_short_warn:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the short from.

GENERAL SYNTAX
user_short_warn:long(addr:long)

DESCRIPTION
Returns the short value from a given user space address. Returns zero when user space and warns (but does not abort) about the failure.

NAME
function::user_int — Retrieves an int value stored in user space.

SYNOPSIS

```
function user_int:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the int from.

GENERAL SYNTAX
user_int:long(addr:long)

DESCRIPTION
Returns the int value from a given user space address. Returns zero when user space data is not accessible.

NAME
function::user_int_warn — Retrieves an int value stored in user space.

SYNOPSIS

```
function user_int_warn:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the int from.

GENERAL SYNTAX
user_ing_warn:long(addr:long)

DESCRIPTION
Returns the int value from a given user space address. Returns zero when user space and warns (but does not abort) about the failure.

NAME
function::user_long — Retrieves a long value stored in user space.

SYNOPSIS

```
function user_long:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the long from.

GENERAL SYNTAX
user_long:long(addr:long)

DESCRIPTION
Returns the long value from a given user space address. Returns zero when user space data is not accessible. Note that the size of the long depends on the architecture of the current user space task (for those architectures that support both 64/32 bit compat tasks).

NAME
function::user_long_warn — Retrieves a long value stored in user space.

SYNOPSIS

```
function user_long_warn:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the long from.

GENERAL SYNTAX

user_long_warn:long(addr:long)

DESCRIPTION

Returns the long value from a given user space address. Returns zero when user space and warns (but does not abort) about the failure. Note that the size of the long depends on the architecture of the current user space task (for those architectures that support both 64/32 bit compat tasks).

NAME

function::user_char — Retrieves a char value stored in user space.

SYNOPSIS

```
function user_char:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the char from.

GENERAL SYNTAX

user_char:long(addr:long)

DESCRIPTION

Returns the char value from a given user space address. Returns zero when user space data is not accessible.

NAME

function::user_char_warn — Retrieves a char value stored in user space.

SYNOPSIS

```
function user_char_warn:long(addr:long)
```

ARGUMENTS

addr

The user space address to retrieve the char from.

GENERAL SYNTAX

user_char_warn:long(addr:long)

DESCRIPTION

Returns the char value from a given user space address. Returns zero when user space and warns (but does not abort) about the failure.

CHAPTER 19. A COLLECTION OF STANDARD STRING FUNCTIONS

Functions to get the length, a substring, getting at individual characters, string seaching, escaping, tokenizing, and converting strings to longs.

NAME

function::strlen — Returns the length of a string.

SYNOPSIS

```
function strlen:long(s:string)
```

ARGUMENTS

s

the string

GENERAL SYNTAX

strlen: long (str:string)

DESCRIPTION

This function returns the length of the string, which can be zero up to MAXSTRINGLEN.

NAME

function::substr — Returns a substring.

SYNOPSIS

```
function substr:string(str:string,start:long,length:long)
```

ARGUMENTS

str

The string to take a substring from

start

Starting position. 0 = start of the string.

length

Length of string to return.

GENERAL SYNTAX

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

DESCRIPTION

Returns the substring of the up to the given length starting at the given start position and ending at given stop position.

NAME

function::stringat — Returns the char at a given position in the string.

SYNOPSIS

```
function stringat:long(str:string,pos:long)
```

ARGUMENTS

str

The string to fetch the character from.

pos

The position to get the character from. 0 = start of the string.

GENERAL SYNTAX

stringat:long(srt:string, pos:long)

DESCRIPTION

This function returns the character at a given position in the string or zero if thestring doesn't have as many characters.

NAME

function::isinstr — Returns whether a string is a substring of another string.

SYNOPSIS

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

ARGUMENTS

s1

String to search in.

s2

Substring to find.

GENERAL SYNTAX

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

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

NAME
function::text_str — Escape any non-printable chars in a string.

SYNOPSIS

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

ARGUMENTS

input

The string to escape.

GENERAL SYNTAX
text_str:string (input:string)

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

NAME
function::text_strn — Escape any non-printable chars in a string.

SYNOPSIS

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

ARGUMENTS

input

The string to escape.

len

Maximum length of string to return. 0 means MAXSTRINGLEN.

quoted

Put double quotes around the string. If input string is truncated it will have "..." after the second quote.

GENERAL SYNTAX
text_strn:string (input:string, len:long, quoted:long)

DESCRIPTION

This function accepts a string of designated length, and any ASCII characters that are not printable are replaced by the corresponding escape sequence in the returned string.

NAME

function::tokenize — Return the next non-empty token in a string.

SYNOPSIS

```
function tokenize:string(input:string,delim:string)
```

ARGUMENTS

input

String to tokenize. If NULL, returns the next non-empty token in the string passed in the previous call to **tokenize**.

delim

Token delimiter. Set of characters that delimit the tokens.

GENERAL SYNTAX

tokenize:string (input:string, delim:string)

DESCRIPTION

This function returns the next non-empty token in the given input string, where the tokens are delimited by characters in the delim string. If the input string is non-NULL, it returns the first token. If the input string is NULL, it returns the next token in the string passed in the previous call to tokenize. If no delimiter is found, the entire remaining input string is returned. It returns NULL when no more tokens are available.

NAME

function::str_replace — str_replace Replaces all instances of a substring with another.

SYNOPSIS

```
function
str_replace:string(prnt_str:string,srch_str:string,rplc_str:string)
```

ARGUMENTS

prnt_str

The string to search and replace in.

srch_str

The substring which is used to search in prnt_str string.

rplc_str

The substring which is used to replace srch_str.

GENERAL SYNTAX

str_replace:string(prnt_str:string, srch_str:string, rplc_str:string)

DESCRIPTION

This function returns the given string with substrings replaced.

NAME

function::strtol — strtol - Convert a string to a long.

SYNOPSIS

```
function strtol:long(str:string,base:long)
```

ARGUMENTS

str

String to convert.

base

The base to use

GENERAL SYNTAX

strtol:long (str:string, base:long)

DESCRIPTION

This function converts the string representation of a number to an integer. The base parameter indicates the number base to assume for the string (eg. 16 for hex, 8 for octal, 2 for binary).

NAME

function::isdigit — Checks for a digit.

SYNOPSIS

```
function isdigit:long(str:string)
```

ARGUMENTS

str

String to check.

GENERAL SYNTAX

isdigit:long(str:string)

DESCRIPTION

Checks for a digit (0 through 9) as the first character of a string. Returns non-zero if true, and a zero if false.

CHAPTER 20. UTILITY FUNCTIONS FOR USING ANSI CONTROL CHARS IN LOGS

Utility functions for logging using ansi control characters. This lets you manipulate the cursor position and character color output and attributes of log messages.

NAME

function::ansi_clear_screen — Move cursor to top left and clear screen.

SYNOPSIS

```
function ansi_clear_screen()
```

ARGUMENTS

None

GENERAL SYNTAX

ansi_clear_screen

DESCRIPTION

Sends ansi code for moving cursor to top left and then the ansi code for clearing the screen from the cursor position to the end.

NAME

function::ansi_set_color — Set the ansi Select Graphic Rendition mode.

SYNOPSIS

```
function ansi_set_color(fg:long)
```

ARGUMENTS

fg

Foreground color to set.

GENERAL SYNTAX

ansi_set_color(fh:long)

DESCRIPTION

Sends ansi code for Select Graphic Rendition mode for the given forground color. Black (30), Blue (34), Green (32), Cyan (36), Red (31), Purple (35), Brown (33), Light Gray (37).

NAME

function::ansi_set_color2 — Set the ansi Select Graphic Rendition mode.

SYNOPSIS

```
function ansi_set_color2(fg:long,bg:long)
```

ARGUMENTS

fg

Foreground color to set.

bg

Background color to set.

GENERAL SYNTAX

ansi_set_color2(fg:long, bg:long)

DESCRIPTION

Sends ansi code for Select Graphic Rendition mode for the given forground color, Black (30), Blue (34), Green (32), Cyan (36), Red (31), Purple (35), Brown (33), Light Gray (37) and the given background color, Black (40), Red (41), Green (42), Yellow (43), Blue (44), Magenta (45), Cyan (46), White (47).

NAME

function::ansi_set_color3 — Set the ansi Select Graphic Rendition mode.

SYNOPSIS

```
function ansi_set_color3(fg:long,bg:long,attr:long)
```

ARGUMENTS

fg

Foreground color to set.

bg

Background color to set.

attr

Color attribute to set.

GENERAL SYNTAX

ansi_set_color3(fg:long, bg:long, attr:long)

DESCRIPTION

Sends ansi code for Select Graphic Rendition mode for the given forground color, Black (30), Blue (34), Green (32), Cyan (36), Red (31), Purple (35), Brown (33), Light Gray (37), the given background color, Black (40), Red (41), Green (42), Yellow (43), Blue (44), Magenta (45), Cyan (46), White (47) and the

color attribute All attributes off (0), Intensity Bold (1), Underline Single (4), Blink Slow (5), Blink Rapid (6), Image Negative (7).

NAME
function::ansi_reset_color — Resets Select Graphic Rendition mode.

SYNOPSIS

```
function ansi_reset_color()
```

ARGUMENTS
None

GENERAL SYNTAX
ansi_reset_color

DESCRIPTION
Sends ansi code to reset foreground, background and color attribute to default values.

NAME
function::ansi_new_line — Move cursor to new line.

SYNOPSIS

```
function ansi_new_line()
```

ARGUMENTS
None

GENERAL SYNTAX
ansi_new_line

DESCRIPTION
Sends ansi code new line.

NAME
function::ansi_cursor_move — Move cursor to new coordinates.

SYNOPSIS

```
function ansi_cursor_move(x:long,y:long)
```

ARGUMENTS

x

x

Row to move the cursor to.

y

Colomn to move the cursor to.

GENERAL SYNTAX

ansi_curos_move(x:long, y:long)

DESCRIPTION

Sends ansi code for positioning the cursor at row x and column y. Coordinates start at one, (1,1) is the top-left corner.

NAME

function::ansi_cursor_hide — Hides the cursor.

SYNOPSIS

```
function ansi_cursor_hide()
```

ARGUMENTS

None

GENERAL SYNTAX

ansi_cusor_hide

DESCRIPTION

Sends ansi code for hiding the cursor.

NAME

function::ansi_cursor_save — Saves the cursor position.

SYNOPSIS

```
function ansi_cursor_save()
```

ARGUMENTS

None

GENERAL SYNTAX

ansi_cursor_save

DESCRIPTION

Sends ansi code for saving the current cursor position.

NAME
function::ansi_cursor_restore — Restores a previously saved cursor position.

SYNOPSIS

```
function ansi_cursor_restore()
```

ARGUMENTS
None

GENERAL SYNTAX
ansi_cursor_restore

DESCRIPTION
Sends ansi code for restoring the current cursor position previously saved with **ansi_cursor_save**.

NAME
function::ansi_cursor_show — Shows the cursor.

SYNOPSIS

```
function ansi_cursor_show()
```

ARGUMENTS
None

GENERAL SYNTAX
ansi_cursor_show

DESCRIPTION
Sends ansi code for showing the cursor.