Red Hat Enterprise Linux 5 SystemTap Tapset Reference

For SystemTap in Red Hat Enterprise Linux 5
Edition 1

Author                          William Cohen  
Author                          Don Domingo

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

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.
SystemTap Tapset Reference

print_ubacktrace ................................................................. 18
ubacktrace ........................................................................ 18

4. Timestamp Functions 19
get_cycles ......................................................................... 19

5. Memory Tapset 21
vm_fault_contains ............................................................ 21
vm.pagefault ................................................................. 21
vm.pagefault.return ....................................................... 21
addr_to_node ................................................................... 22
vm.write_shared ........................................................... 22
vm.write_shared_copy .................................................... 22
vm.mmap ......................................................................... 23
vm.munmap ..................................................................... 23
vm.brk .......................................................................... 24
vm.oom_kill ................................................................... 24

6. IO Scheduler Tapset 25
ioscheduler.elv_next_request ........................................... 25
ioscheduler.elv_next_request.return ............................... 25
ioscheduler.elv_add_request ........................................... 25
ioscheduler.elv_completed_request .................................. 26

7. SCSI Tapset 27
scsi.iointry ................................................................. 27
scsi.iodispatching ......................................................... 27
scsi.iodone ................................................................. 28
scsi.iocompleted .......................................................... 28

8. Networking Tapset 31
netdev.receive ............................................................... 31
netdev.transmit ............................................................. 31
tcp.sendmsg ................................................................... 31
tcp.sendmsg.return ....................................................... 32
tcp.rcvmsg ................................................................... 32
tcp.rcvmsg.return ....................................................... 33
tcp.disconnect .............................................................. 34
tcp.disconnect.return ................................................... 34
tcp.setsockopt .............................................................. 35
tcp.setsockopt.return .................................................... 35
tcp.receive ................................................................. 36
udp.sendmsg ................................................................. 36
udp.sendmsg.return ....................................................... 37
udp.rcvmsg ................................................................... 37
udp.rcvmsg.return ....................................................... 38
udp.disconnect .............................................................. 38
udp.disconnect.return ................................................... 39
ip_ntop .......................................................................... 39

9. Socket Tapset 41
socket.send ................................................................. 41
socket.receive ............................................................. 41
socket.sendmsg ............................................................ 42
socket.sendmsg.return ................................................... 43
socket.rcvmsg .............................................................. 44
socket.rcvmsg.return ..................................................... 44
socket.aio_write .......................................................... 45
socket.aio_write.return ............................................. 46
socket.aio_read ............................................................ 47
socket.aio_read.return ................................................. 47
socket.writev ............................................................... 48
socket.writev.return .................................................... 49
socket.readv .................................................................. 50
socket.readv.return ....................................................... 51
socket.create .............................................................. 51
socket.create.return .................................................... 52
socket.close .................................................................. 53
socket.close.return ...................................................... 53
sock_prot_num2str ......................................................... 54
sock_prot_str2num ........................................................ 54
sock_fam_num2str ........................................................ 54
sock_fam_str2num ........................................................ 55
sock_state_num2str ....................................................... 55
sock_state_str2num ....................................................... 55

10. Kernel Process Tapset .................................................. 57
    kprocess.create .......................................................... 57
    kprocess.start .......................................................... 57
    kprocess.exec ........................................................... 57
    kprocess.exec_complete .............................................. 58
    kprocess.exit ............................................................ 58
    kprocess.release ........................................................ 59

11. Signal Tapset ............................................................ 61
    signal.send ............................................................... 61
    signal.send.return ..................................................... 61
    signal.checkperm ........................................................ 62
    signal.checkperm.return .......................................... 63
    signal.wakeup ............................................................ 63
    signal.check_ignored .................................................. 64
    signal.check_ignored.return ........................................ 64
    signal.force_segv ....................................................... 65
    signal.force_segv.return .............................................. 65
    signal.syskill ............................................................ 65
    signal.syskill.return .................................................. 66
    signal.sys_tkill .......................................................... 66
    signal.sys_tkill.return ................................................. 66
    signal.sys_tgkill .......................................................... 67
    signal.sys_tgkill.return ................................................. 67
    signal.send_sig_queue ................................................... 67
    signal.send_sig_queue.return ..................................... 68
    signal.pending .......................................................... 68
    signal.pending.return ................................................ 68
    signal.handle ........................................................... 69
    signal.handle.return .................................................. 69
    signal.do_action ........................................................ 70
    signal.do_action.return .............................................. 70
    signal.procmask ........................................................ 71
    signal.flush .............................................................. 71

A. Revision History .......................................................... 73
Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the Liberation Fonts\(^1\) set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

**Mono-spaced Bold**

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

- Press **Enter** to execute the command.
- Press **Ctrl+Alt+F2** to switch to the first virtual terminal. Press **Ctrl+Alt+F1** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

- File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

**Proportional Bold**

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System → Preferences → Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

\(^1\) [https://fedorahosted.org/liberation-fonts/](https://fedorahosted.org/liberation-fonts/)
Close to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a gedit file, choose Applications → Accessories → Character Map from the main menu bar. Next, choose Search → Find… from the Character Map menu bar, type the name of the character in the Search field and click Next. The character you sought will be highlighted in the Character Table. Double-click this highlighted character to place it in the Text to copy field and then click the Copy button. Now switch back to your document and choose Edit → Paste from the gedit menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

**Mono-spaced Bold Italic or Proportional Bold Italic**

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type ssh username@domain.name at a shell prompt. If the remote machine is example.com and your username on that machine is john, type ssh john@example.com.

The mount -o remount file-system command remounts the named file system. For example, to remount the /home file system, the command is mount -o remount /home.

To see the version of a currently installed package, use the rpm -q package command. It will return a result as follows: package-version-release.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a DocBook publishing system.

### 1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in mono-spaced roman and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in mono-spaced roman but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;
import javax.naming.InitialContext;
```
public class ExClient
{
    public static void main(String args[])
    throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("EchoBean");
        EchoHome home = (EchoHome) ref;
        Echo echo = home.create();

        System.out.println("Created Echo");
        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}

1.3. Notes and Warnings
Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

---

**Note**

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

---

**Important**

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

---

**Warning**

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

---

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at [http://access.redhat.com](http://access.redhat.com). Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at [https://www.redhat.com/mailman/listinfo](https://www.redhat.com/mailman/listinfo). Click on the name of any mailing list to subscribe to that list or to access the list archives.
2.2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: http://bugzilla.redhat.com/ against the product Red_Hat_Enterprise_Linux.

When submitting a bug report, be sure to mention the manual's identifier: Tapset_Reference_Guide

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.
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.
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:

```c
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`.

```plaintext
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 across 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:

```c
/**
 * 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:

```c
/**
 * 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:

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

For example:

```c
/**
 * 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.
Chapter 2. Tapset Development Guidelines

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)
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**

print_regs — Print a register dump.

**Synopsis**

```plaintext
function print_regs()
```

**Arguments**

None

**Name**

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

**Synopsis**

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

**Arguments**

None

**Name**

pid — Returns the ID of a target process.

**Synopsis**

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

**Arguments**

None

**Name**

tid — Returns the thread ID of a target process.
Chapter 3. Context Functions

Synopsis

function tid:long()

Arguments
None

Name
ppid — Returns the process ID of a target process's parent process.

Synopsis

function ppid:long()

Arguments
None

Name
pgrp — Returns the process group ID of the current process.

Synopsis

function pgrp:long()

Arguments
None

Name
sid — Returns the session ID of the current process.

Synopsis

function sid:long()

Arguments
None
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
pexecname — Returns the execname of a target process's parent process.

Synopsis

function pexecname:string()

Arguments
None

Name
gid — Returns the group ID of a target process.

Synopsis

function gid:long()

Arguments
None

Name
egid — Returns the effective gid of a target process.

Synopsis

function egid:long()

Arguments
None

Name
uid — Returns the user ID of a target process.

Synopsis

function uid:long()
Chapter 3. Context Functions

**Arguments**
None

**Name**
euid — Return the effective uid of a target process.

**Synopsis**

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

Arguments
None

**Name**
cpu — Returns the current cpu number.

**Synopsis**

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

Arguments
None

**Name**
pp — Return the probe point associated with the currently running probe handler,

**Synopsis**

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

Arguments
None

**Description**
including alias and wildcard expansion effects

**Context**
The current probe point.
**Name**

registers_valid — Determines validity of `register` and `u_register` in current context.

**Synopsis**

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

**Arguments**

None

**Description**

Return 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**

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

**Synopsis**

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

**Arguments**

None

**Description**

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

**Name**

is_return — Determines if probe point is a return probe.

**Synopsis**

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

**Arguments**

None

**Description**

Return 1 if the probe point is a return probe. *Deprecated.*
Chapter 3. Context Functions

**Name**
target — Return the process ID of the target process.

**Synopsis**

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

**Arguments**
None

**Name**
stack_size — Return the size of the kernel stack.

**Synopsis**

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

**Arguments**
None

**Name**
stack_used — Returns the amount of kernel stack used.

**Synopsis**

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

**Arguments**
None

**Description**
Determines how many bytes are currently used in the kernel stack.

**Name**
stack_unused — Returns the amount of kernel stack currently available.

**Synopsis**

```
function stackunused:long()
```
**Arguments**
None

**Description**
Determines how many bytes are currently available in the kernel stack.

**Name**
uaddr — User space address of current running task. EXPERIMENTAL.

**Synopsis**

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

**Arguments**
None

**Description**
Returns the address in userspace that the current task was at when the probe occured. 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**
print_stack — Print out stack from string.

**Synopsis**

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

**Arguments**

- **stk**
  String with list of hexadecimal 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**.

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**
probefunc — Return the probe point's function name, if known.
Chapter 3. Context Functions

**Synopsis**

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

**Arguments**

None

**Name**

`probemod` — Return the probe point's module name, if known.

**Synopsis**

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

**Arguments**

None

**Name**

`modname` — Return the kernel module name loaded at the address.

**Synopsis**

```plaintext
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**

`symname` — Return the symbol associated with the given address.
**Function**

**symname:** \textit{string}(\textit{addr}:long)

**Arguments**

\textit{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 \textit{addr}.

**Name**

\textit{symdata} — Return the symbol and module offset for the address.

**Synopsis**

\begin{verbatim}
function symdata:string(addr:long)
\end{verbatim}

**Arguments**

\textit{addr}

The address to translate.

**Description**

Returns the (function) symbol name associated with the given address if known, plus the module name (between brackets) and the offset inside the module, plus the size of the symbol function. 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**

\textit{usymname} — Return the symbol of an address in the current task. EXPERIMENTAL!

**Synopsis**

\begin{verbatim}
function usymname:string(addr:long)
\end{verbatim}

**Arguments**

\textit{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 \textit{addr}.


Chapter 3. Context Functions

Name
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, plus the module name (between brackets) and the offset inside the module (shared library), plus the size of the symbol function. If any element is not known it will be ommitted and if the symbol name is unknown it will return the hex string for the given address.

Name

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

Synopsis

function print_ustack(stk:string)

Arguments

stk
String with list of hexidecimal 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

print_backtrace — Print stack back trace

Synopsis

function print_backtrace()
**Arguments**

None

**Description**

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

**Name**

backtrace — Hex backtrace of current stack

**Synopsis**

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

**Arguments**

None

**Description**

Return a string of hex addresses that are a backtrace of the stack. Output may be truncated as per maximum string length.

**Name**

caller — Return name and address of calling function

**Synopsis**

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

**Arguments**

None

**Description**

Return the address and name of the calling function.

**This is equivalent to calling**

`sprintf("s 0x", symname(caller_addr, caller_addr)) Works only for return probes at this time.`

**Name**

caller_addr — Return caller address
Chapter 3. Context Functions

Synopsis

\textit{function caller_addr:long()}

Arguments

None

Description

Return the address of the calling function. \textit{Works only for return probes at this time.}

Name

\texttt{print_ubacktrace} — Print stack back trace for current task. EXPERIMENTAL!

Synopsis

\textit{function print_ubacktrace()}

Arguments

None

Description

Equivalent to \texttt{print_ustack(ubacktrace)}, except that deeper stack nesting may be supported. Return nothing.

Name

\texttt{ubacktrace} — Hex backtrace of current task stack. EXPERIMENTAL!

Synopsis

\textit{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.
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

get_cycles — Processor cycle count.

Synopsis

function get_cycles:long()

Arguments

None

Description

Return the processor cycle counter value, or 0 if unavailable.
Memory Tapset
This family of probe points is used to probe memory-related events. It contains the following probe points:

**Name**
vm_fault_contains — Test return value for page fault reason

**Synopsis**

```plaintext
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**
vm.pagefault — Records that a page fault occurred.

**Synopsis**

```plaintext
vm.pagefault
```

**Values**

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

- **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**
vm.pagefault.return — Indicates what type of fault occurred.

**Synopsis**

```plaintext
vm.pagefault.return
```
Values

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

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.

Name

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

Synopsis

vm.write_shared

Values

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

vm.write_shared_copy — Page copy for shared page write.

Synopsis

vm.write_shared_copy
Values

zer
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
vm.mmap — Fires when an mmap is requested.

Synopsis

```c
vm.mmap
```

Values

length
The length of the memory segment

address
The requested address

Context
The process calling mmap.

Name
vm.munmap — Fires when an munmap is requested.

Synopsis

```c
vm.munmap
```

Values

length
The length of the memory segment
address
The requested address

Context
The process calling munmap.

Name
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

address
The requested address

Context
The process calling brk.

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

Synopsis

vm.oom_kill

Values

task
The task being killed

Context
The process that tried to consume excessive memory, and thus triggered the OOM.
IO Scheduler Tapset
This family of probe points is used to probe IO scheduler activities. It contains the following probe points:

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

Synopsis
ioscheduler.elv_next_request

Values
elevator_name
   The type of I/O elevator currently enabled

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

Synopsis
ioscheduler.elv_next_request.return

Values
req_flags
   Request flags

req
   Address of the request

disk_major
   Disk major number of the request

disk_minor
   Disk minor number of the request

Name
ioscheduler.elv_add_request — A request was added to the request queue

Synopsis
ioscheduler.elv_add_request
Values

`req_flags`
- Request flags

`req`
- Address of the request

`disk_major`
- Disk major number of the request

`elevator_name`
- The type of I/O elevator currently enabled

`disk_minor`
- Disk minor number of the request

Name

`ioscheduler.elv_completed_request` — Fires when a request is completed

Synopsis

```
ioscheduler.elv_completed_request
```

Values

`req_flags`
- Request flags

`req`
- Address of the request

`disk_major`
- Disk major number of the request

`elevator_name`
- The type of I/O elevator currently enabled

`disk_minor`
- Disk minor number of the request
Chapter 7.

**SCSI Tapset**

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

**Name**

scsi.ioentry — Prepares a SCSI mid-layer request

**Synopsis**

```plaintext
scsi.ioentry
```

**Values**

*disk_major*

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

*device_state*

The current state of the device.

*disk_minor*

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

**Name**

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

**Synopsis**

```plaintext
scsi.iodispatching
```

**Values**

*lun*

The lun number

*req_bufflen*

The request buffer length

*host_no*

The host number

*device_state*

The current state of the device.

*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)

**request_buffer**
The request buffer address

**Name**
scsi.iiodone — SCSI command completed by low level driver and enqueued into the done queue.

**Synopsis**
```
scsi.iiodone
```

**Values**

**lun**
The lun number

**host_no**
The host number

**device_state**
The current state of the device

**dev_id**
The scsi device id

**channel**
The channel number

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

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

**Synopsis**
```
scsi.iocompleted
```

**Values**

**lun**
The lun number

**host_no**
The host number
`device_state`
The current state of the device

`dev_id`
The scsi device id

`channel`
The channel number

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

`goodbytes`
The bytes completed.
Networking Tapset

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

Name
netdev.receive — Data recieved from network device.

Synopsis

netdev.receive

Values

protocol
Protocol of recieved packet.

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

length
The length of the receiving buffer.

Name
netdev.transmit — Network device transmitting buffer

Synopsis

netdev.transmit

Values

protocol
The protocol of this packet.

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

length
The length of the transmit buffer.

truesize
The size of the the data to be transmitted.

Name
tcp.sendmsg — Sending a tcp message
Chapter 8. Networking Tapset

**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**
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**
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
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
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
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**
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**
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
tcp.receive — Called when a TCP packet is received

Synopsis
tcp.receive

Values
urg  
TCP URG flag

psh  
TCP PSH flag

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

syn  
TCP SYN flag

fin  
TCP FIN flag

sport  
TCP source port

Name
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

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

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

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

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

**Name**
socket.send — Message sent on a socket.

**Synopsis**

```plaintext
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**
socket.receive — Message received on a socket.

**Synopsis**

```plaintext
socket.receive
```
Chapter 9. Socket Tapset

**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**

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

**Synopsis**

```python
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**
`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
`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
`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**
socket.aio_write — Message send via `sock_aio_write`

**Synopsis**

```c
socket.aio_write
```

**Values**

**protocol**
Protocol value

**flags**
Socket flags value

**name**
Name of this probe
Chapter 9. Socket Tapset

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
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
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
socket.aio_read.return — Conclusion of message received via sock_aio_read
Chapter 9. Socket Tapset

**Synopsis**

```plaintext
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**

socket.writev — Message sent via `socket_writev`

**Synopsis**

```plaintext
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**
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
Chapter 9. Socket Tapset

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

Synopsis

```plaintext
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

socket.create — Creation of a socket

Synopsis

```plaintext
socket.create
```
Chapter 9. Socket Tapset

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
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
**Context**
The requester (user process or kernel)

**Description**
Fires at the conclusion of creating a socket.

**Name**
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**
socket.close.return — Return from closing a socket
Chapter 9. Socket Tapset

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

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

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

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

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

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

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

state
  The state name.
Chapter 10.

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

**Name**
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**
kprocess.start — Starting new process

**Synopsis**

```
kprocess.start
```

**Values**

None

**Context**

Newly created process.

**Description**

Fires immediately before a new process begins execution.

**Name**
kprocess.exec — Attempt to exec to a new program
Chapter 10. Kernel Process Tapset

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
kprocess.exec_complete — Return from exec to a new program

Synopsis

kprocess.exec_complete

Values

success
A boolean indicating whether the exec was successful

erro
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
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
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.
Signal Tapset

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

**Name**
signal.send — Signal being sent to a process

**Synopsis**

```plaintext
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**
signal.send.return — Signal being sent to a process completed
Chapter 11. Signal Tapset

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.

Description
Possible __group_send_sig_info and specific_send_sig_info return values are as follows;

0 -- The signal is successfully 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 successfully 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
signal.checkperm — Check being performed on a sent signal

Synopsis

```
signal.checkperm
```
**Values**

*name*
   
   Name of the probe point; default value is `signal.checkperm`

*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**

`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; default value is `signal.checkperm`

**Name**

`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

**signal.check_ignored** — Checking to see signal is ignored

Synopsis

```none
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

**signal.check_ignored.return** — Check to see signal is ignored completed

Synopsis

```none
signal.check_ignored.return
```

Values

**retstr**
Return value as a string

**name**
Name of the probe point; default value is **signal.checkperm**
Name
signal.force_segv — Forcing send of SIGSEGV

Synopsis

```
signal.force_segv
```

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
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; default value is force_sigsegv

Name
signal.syskill — Sending kill signal to a process

Synopsis

```
signal.syskill
```
Chapter 11. Signal Tapset

Values

sig
The specific signal sent to the process

pid
The PID of the process receiving the signal

Name

signal.syskill.return — Sending kill signal completed

Synopsis

signal.syskill.return

Values

None

Name

signal.sys_tkill — Sending a kill signal to a thread

Synopsis

signal.sys_tkill

Values

sig_name
The specific signal sent to the process

sig
The specific signal sent to the process

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 targetted. Such processes are targetted through their unique thread IDs (TID).

Name

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

Synopsis
signal.sys_tgkill

**Values**
None

**Name**
signal.sys_tgkill — Sending kill signal to a thread group

**Synopsis**

signal.sys_tgkill

**Values**

- **sig_name**
  A string representation of the signal

- **sig**
  The specific kill signal sent to the process

- **pid**
  The PID of the thread receiving the kill signal

- **tgid**
  The thread group ID 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**
signal.sys_tgkill.return — Sending kill signal to a thread group completed

**Synopsis**

signal.sys_tgkill.return

**Values**
None

**Name**
signal.send_sig_queue — Queuing a signal to a process
Chapter 11. Signal Tapset

Synopsis

```python
signal.send_sig_queue
```

Values

- `sigqueue_addr`
  - The address of the signal queue

- `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

`signal.send_sig_queue.return` — Queuing a signal to a process completed

Synopsis

```python
signal.send_sig_queue.return
```

Values

- `retstr`
  - Return value as a string

Name

`signal.pending` — Examining pending signal

Synopsis

```python
signal.pending
```

Values

- `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_siptending` kernel function is executed.

Name
signal.pending.return — Examination of pending signal completed

Synopsis
```plaintext
signal.pending.return
```

Values
- `retstr`
  Return value as a string

Name
signal.handle — Signal handler being invoked

Synopsis
```plaintext
signal.handle
```

Values
- `regs`
  The address of the kernel-mode stack area
- `sig_code`
  The `si_code` value of the `siginfo` signal
- `sig_mode`
  Indicates whether the signal was a user-mode or kernel-mode signal
- `sinfo`
  The address of the `siginfo` table
- `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
signal.handle.return — Signal handler invocation completed
Chapter 11. Signal Tapset

Synopsis

signal.handle.return

Values

retstr
Return value as a string

Name

signal.do_action — Examining or changing a signal action

Synopsis

signal.do_action

Values

sa_mask
The new mask 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

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

Synopsis

signal.do_action.return

Values

retstr
Return value as a string
**Name**
signal.procmask — Examining or changing blocked signals

**Synopsis**

```bash
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.

**oldsigset_addr**
The old address of the signal set (`sigset_t`)

**sigset**
The actual value to be set for `sigset_t`

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

**Name**
signal.flush — Flusing all pending signals for a task

**Synopsis**

```bash
signal.flush
```

**Values**

**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
Appendix A. Revision History

Revision 1.0       Wed Jun 17 2009       Don Domingo ddomingo@redhat.com
                     building book in RHEL