Red Hat Enterprise Linux 6 SystemTap Tapset Reference
For SystemTap in Red Hat Enterprise Linux 6
Edition 0

Author
William Cohen   wcohen@redhat.com
Author
Don Domingo   ddomingo@redhat.com

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.
4. Timestamp Functions

function::timestamp ................................................................. 37

function::gettimeofday ............................................................ 38

4.3. Time string utility function

function::ctime ................................................................. 41

6. Memory Tapset

function::vm_fault ................................................................. 43

probe::vm.fault ................................................................. 43

probe::vm.pagefault ........................................................... 43

function::addr_to_node ......................................................... 44

probe::vm.write ................................................................. 44

probe::vm.write_shared ......................................................... 45

function::vm.mmap ............................................................. 45

probe::vm.munmap ............................................................... 46

probe::vm.brk ................................................................. 46

probe::vm.oomkill ............................................................... 47
7. Task Time Tapset

function::task_utime ................................................................. 57
function::task_utime_tid ......................................................... 57
function::task_stime ............................................................... 57
function::task_stime_tid ........................................................ 57
function::cputime_to_msecs .................................................... 58
function::msecs_to_string ...................................................... 58
function::cputime_to_string ................................................... 58
function::task_time_string ..................................................... 59
function::task_time_string_tid ............................................... 59

8. IO Scheduler and block IO Tapset

probe::ioscheduler.elv_next_request ......................................... 61
probe::ioscheduler.elv_next_request.return ............................. 61
probe::ioscheduler.elv_completed_request ............................... 61
probe::ioscheduler.elv_add_request.kp .................................... 62
probe::ioscheduler.elv_add_request.tp .................................... 63
probe::ioscheduler.elv_add_request ........................................ 63
probe::ioscheduler_trace.elv_completed_request ...................... 64
probe::ioscheduler_trace.elv_issue_request ............................. 64
probe::ioscheduler_trace.elv_requeue_request ......................... 65
probe::ioscheduler_trace.elv_abort_request ............................. 66
probe::ioscheduler_trace.plug ................................................ 66
probe::ioscheduler_trace.unplug_io ........................................ 67
probe::ioscheduler_trace.unplug_timer .................................... 67
probe::ioblock.request .......................................................... 67
probe::ioblock.end ............................................................... 68
probe::ioblock_trace.bounce .................................................. 69
probe::ioblock_trace.request ................................................ 69
probe::ioblock_trace.end ........................................................ 70

9. SCSI Tapset

probe::scsi.ioentry ................................................................. 71
SystemTap Tapset Reference

10. TTY Tapset 77
probe::tty.open ................................................................. 77
probe::tty.release .............................................................. 77
probe::tty.resize .............................................................. 78
probe::tty.ioctl ................................................................. 78
probe::tty.init ................................................................. 79
probe::tty.register ............................................................ 79
probe::tty.unregister .......................................................... 80
probe::tty.poll ................................................................. 80
probe::tty.receive ............................................................. 80
probe::tty.write ............................................................... 81
probe::tty.read ................................................................. 81

11. Networking Tapset 83
probe::netdev.receive .......................................................... 83
probe::netdev.transmit ........................................................ 83
probe::netdev.change_mtu .................................................... 83
probe::netdev.open ............................................................ 84
probe::netdev.close ............................................................ 84
probe::netdev.hard_transmit ............................................... 84
probe::netdev.rx ............................................................... 85
probe::netdev.change_rx_flag ............................................ 85
probe::netdev.set_promiscuity ........................................... 85
probe::netdev.ioctl ............................................................ 86
probe::netdev.register ....................................................... 86
probe::netdev.unregister .................................................... 87
probe::netdev.get_stats ..................................................... 87
probe::netdev.change_mac .................................................. 87
probe::netdev.change_mac .................................................. 87
probe::netdev.change_mac .................................................. 87
probe::netdev.change_mac .................................................. 87
probe::tcp.sendmsg .......................................................... 88
probe::tcp.sendmsg.return ................................................. 88
probe::tcp.recvmsg ........................................................... 88
probe::tcp.recvmsg.return ............................................... 89
probe::tcp.disconnect ........................................................ 90
probe::tcp.disconnect.return ............................................. 90
probe::tcp.setsockopt ......................................................... 91
probe::tcp.setsockopt.return .............................................. 91
probe::tcp.receive ........................................................... 92
probe::udp.sendmsg .......................................................... 93
probe::udp.sendmsg.return ............................................... 93
probe::udp.recvmsg ........................................................... 94
probe::udp.recvmsg.return ............................................... 94
probe::udp.disconnect ......................................................... 94
probe::udp.disconnect.return ............................................. 95
function::ip_ntop .............................................................. 95

12. Socket Tapset 97
probe::socket.send ............................................................. 97
probe::socket.receive ........................................................ 97
probe::socket.sendmsg ....................................................... 98
13. Kernel Process Tapset

probe::kprocess.create ................................................................. 113
probe::kprocess.start ................................................................. 113
probe::kprocess.exec ................................................................. 113
probe::kprocess.exec_complete .................................................. 114
probe::kprocess.exit ................................................................. 114
probe::kprocess.release ............................................................ 115

14. Signal Tapset

probe::signal.send ................................................................. 117
probe::signal.send.return ......................................................... 117
probe::signal.checkperm ......................................................... 118
probe::signal.checkperm.return ............................................... 119
probe::signal.wakeup ............................................................. 119
probe::signal.check_ignored ..................................................... 120
probe::signal.check_ignored.return ......................................... 120
probe::signal.force_segv ......................................................... 121
probe::signal.force_segv.return .............................................. 121
probe::signal.syskill ............................................................... 121
probe::signal.syskill.return .................................................... 122
probe::signal.sys_tkill ............................................................. 122
probe::signal.syskill.return .................................................... 123
probe::signal.sys_tkill ............................................................. 123
probe::signal.sys_tkill.return .................................................. 124
probe::signal.send_sig_queue .................................................. 124
probe::signal.send_sig_queue.return ....................................... 124
probe::signal.pending ............................................................. 125
probe::signal.pending.return .................................................. 125
probe::signal.handle ............................................................... 126
probe::signal.handle.return ................................................... 126
probe::signal.do_action ........................................................... 127
probe::signal.do_action.return ............................................... 127
probe::signal.procmask ........................................................... 128
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemTap Tapset Reference</td>
<td></td>
</tr>
<tr>
<td>15. Directory-entry (dentry) Tapset</td>
<td>131</td>
</tr>
<tr>
<td>function::d_name</td>
<td>131</td>
</tr>
<tr>
<td>function::reverse_path_walk</td>
<td>131</td>
</tr>
<tr>
<td>function::task_dentry_path</td>
<td>131</td>
</tr>
<tr>
<td>function::d_path</td>
<td>132</td>
</tr>
<tr>
<td>16. Logging Tapset</td>
<td>133</td>
</tr>
<tr>
<td>function::log</td>
<td>133</td>
</tr>
<tr>
<td>function::warn</td>
<td>133</td>
</tr>
<tr>
<td>function::exit</td>
<td>134</td>
</tr>
<tr>
<td>function::error</td>
<td>134</td>
</tr>
<tr>
<td>function::ftrace</td>
<td>134</td>
</tr>
<tr>
<td>17. Random functions Tapset</td>
<td>137</td>
</tr>
<tr>
<td>function::randint</td>
<td>137</td>
</tr>
<tr>
<td>18. String and data retrieving functions Tapset</td>
<td>139</td>
</tr>
<tr>
<td>function::kernel_string</td>
<td>139</td>
</tr>
<tr>
<td>function::kernel_string2</td>
<td>139</td>
</tr>
<tr>
<td>function::kernel_string_n</td>
<td>140</td>
</tr>
<tr>
<td>function::kernel_long</td>
<td>140</td>
</tr>
<tr>
<td>function::kernel_int</td>
<td>141</td>
</tr>
<tr>
<td>function::kernel_short</td>
<td>141</td>
</tr>
<tr>
<td>function::kernel_char</td>
<td>141</td>
</tr>
<tr>
<td>function::kernel_pointer</td>
<td>142</td>
</tr>
<tr>
<td>function::user_string</td>
<td>142</td>
</tr>
<tr>
<td>function::user_string2</td>
<td>143</td>
</tr>
<tr>
<td>function::user_string_warn</td>
<td>143</td>
</tr>
<tr>
<td>function::user_string_quoted</td>
<td>144</td>
</tr>
<tr>
<td>function::user_string_n</td>
<td>144</td>
</tr>
<tr>
<td>function::user_string_n2</td>
<td>145</td>
</tr>
<tr>
<td>function::user_string_n_warn</td>
<td>145</td>
</tr>
<tr>
<td>function::user_string_n_quoted</td>
<td>146</td>
</tr>
<tr>
<td>function::user_short</td>
<td>146</td>
</tr>
<tr>
<td>function::user_short_warn</td>
<td>147</td>
</tr>
<tr>
<td>function::user_int</td>
<td>147</td>
</tr>
<tr>
<td>function::user_int_warn</td>
<td>147</td>
</tr>
<tr>
<td>function::user_long</td>
<td>148</td>
</tr>
<tr>
<td>function::user_long_warn</td>
<td>148</td>
</tr>
<tr>
<td>function::user_long_warn</td>
<td>149</td>
</tr>
<tr>
<td>function::user_char</td>
<td>149</td>
</tr>
<tr>
<td>function::user_char_warn</td>
<td>149</td>
</tr>
<tr>
<td>19. A collection of standard string functions</td>
<td>151</td>
</tr>
<tr>
<td>function::strlen</td>
<td>151</td>
</tr>
<tr>
<td>function::substr</td>
<td>151</td>
</tr>
<tr>
<td>function::stringat</td>
<td>151</td>
</tr>
<tr>
<td>function::isinstr</td>
<td>152</td>
</tr>
<tr>
<td>function::text_str</td>
<td>152</td>
</tr>
<tr>
<td>function::text_strn</td>
<td>152</td>
</tr>
<tr>
<td>function::tokenize</td>
<td>153</td>
</tr>
<tr>
<td>function::str_replace</td>
<td>153</td>
</tr>
<tr>
<td>function::strtol</td>
<td>154</td>
</tr>
<tr>
<td>function::isdigit</td>
<td>155</td>
</tr>
</tbody>
</table>
20. Utility functions for using ansi control chars in logs

- function::ansi_clear_screen ................................................................. 157
- function::ansi_set_color ........................................................................ 157
- function::ansi_set_color2 ....................................................................... 158
- function::ansi_set_color3 ....................................................................... 158
- function::ansi_reset_color ..................................................................... 159
- function::ansi_new_line .......................................................................... 159
- function::ansi_cursor_move .................................................................... 159
- function::ansi_cursor_hide .................................................................... 160
- function::ansi_cursor_save ..................................................................... 160
- function::ansi_cursor_restore ................................................................. 161
- function::ansi_cursor_show ..................................................................... 161
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 Fonts¹ 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/
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:

<table>
<thead>
<tr>
<th>books</th>
<th>Desktop</th>
<th>documentation</th>
<th>drafts</th>
<th>mss</th>
<th>photos</th>
<th>stuff</th>
<th>svn</th>
</tr>
</thead>
<tbody>
<tr>
<td>books_tests</td>
<td>Desktop1</td>
<td>downloads</td>
<td>images</td>
<td>notes</td>
<td>scripts</td>
<td>svgs</td>
<td></td>
</tr>
</tbody>
</table>

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

```java
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. 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. 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: doc-SystemTap_Tapset_Reference

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.
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.
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.

```systemtap
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

Probes 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.
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**

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.
Chapter 3. Context Functions

Synopsis

\[ function \text{ pid:long() } \]

Arguments
None

General Syntax

pid:long

Description
This function returns the ID of a target process.

Name

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

Synopsis

\[ function \text{ tid:long() } \]

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

\[ function \text{ ppid:long() } \]

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.
Chapter 3. Context Functions

Synopsis

\texttt{function pexecname:string()}

Arguments
None

General Syntax
\texttt{pexecname:string}

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

Name
\texttt{function::gid — Returns the group ID of a target process.}

Synopsis

\texttt{function gid:long()}

Arguments
None

General Syntax
\texttt{gid:long}

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

Name
\texttt{function::egid — Returns the effective gid of a target process.}

Synopsis

\texttt{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.
Chapter 3. Context Functions

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**

```
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.
```
Chapter 3. Context Functions

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**
```plaintext
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**
```plaintext
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
### Chapter 3. Context Functions

#### 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 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_arg` — Fetch argument from command line

#### Synopsis

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

**Arguments**

- None

**General Syntax**

`cmdline_arg:string`

**Description**

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

**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 environment variables from the current process. Returns the empty string when the environment variables 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
evtn_var:sting(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.
Chapter 3. Context Functions

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 function returns the name of the kernel module containing the probe point, if known.
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.

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.

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

Synopsis

```
```
Chapter 3. Context Functions

Arguments

\texttt{symdata:string(addr:long)}

Arguments

\texttt{addr}  
The address to translate.

General Syntax

\texttt{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

Arguments

\texttt{usymname:string(addr:long)}

Arguments

\texttt{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 \texttt{addr}.

Name

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

Synopsis

Arguments

\texttt{usymdata:string(addr:long)}

Arguments

\texttt{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

```plaintext
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

```plaintext
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
Chapter 3. Context Functions

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

Name
function::sprint_backtrace:string() — 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**

```plaintext
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 per maximum string length (MAXSTRINGLEN).

**Name**

`function::task_backtrace — Hex backtrace of an arbitrary task`

---

**Synopsis**

```plaintext
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**

```plaintext
function caller:string()
```
Chapter 3. Context Functions

Arguments
None

General Syntax
caller::string

Description
This function returns the address and name of the calling function. This is equivalent to calling:
\texttt{sprintf("0x\%x", \text{symname}(\text{caller\_addr}, \text{caller\_addr}))}
Works only for return probes at this time.

Name
function::caller\_addr — Return caller address

Synopsis

\begin{verbatim}
function caller\_addr:long()
\end{verbatim}

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

\begin{verbatim}
function print\_ubacktrace()
\end{verbatim}

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

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

Arguments

```c
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

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

Arguments

```c
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.
Chapter 3. Context Functions

Synopsis

```c
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

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

Arguments

task
task_struct pointer.

General Syntax

task_pid:long (task:long)

Description

This function returns the process id of the given task.

Name

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

Synopsis

```c
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_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.
Chapter 3. Context Functions

Synopsis

\[ \text{function task_gid:long(task:long)} \]

Arguments

\[ \text{task} \]

\[
\text{task_struct pointer.}
\]

General Syntax

\[ \text{task_gid:long(task:long)} \]

Description

This function returns the group id of the given task.

Name

\[ \text{function::task_egid — The effective group identifier of the task.} \]

Synopsis

\[ \text{function task_egid:long(task:long)} \]

Arguments

\[ \text{task} \]

\[
\text{task_struct pointer.}
\]

General Syntax

\[ \text{task_egid:long(task:long)} \]

Description

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

Name

\[ \text{function::task_uid — The user identifier of the task.} \]

Synopsis

\[ \text{function task_uid:long(task:long)} \]
Arguments

\textit{task}

\begin{verbatim}
    task struct pointer.
\end{verbatim}

General Syntax

\texttt{task\_uid\_long(task:long)}

Description

This function returns the user id of the given task.

Name

\texttt{function::task\_euid} — The effective user identifier of the task.

Synopsis

\begin{verbatim}
function task\_euid\_long(task:long)
\end{verbatim}

Arguments

\textit{task}

\begin{verbatim}
    task struct pointer.
\end{verbatim}

General Syntax

\texttt{task\_euid\_long(task:long)}

Description

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

Name

\texttt{function::task\_prio} — The priority value of the task.

Synopsis

\begin{verbatim}
function task\_prio\_long(task:long)
\end{verbatim}

Arguments

\textit{task}

\begin{verbatim}
    task struct pointer.
\end{verbatim}

General Syntax

\texttt{task\_prio\_long(task:long)}
Chapter 3. Context Functions

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

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

**Arguments**

*task*

- task_struct pointer.

**General Syntax**

```c
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

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

**Arguments**

*task*

- task_struct pointer.

**General Syntax**

```c
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

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

**Arguments**

None
Chapter 3. Context Functions

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**

```plaintext
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**

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

**Arguments**

None

**General Syntax**

`gettimeofday_ns:long`

**Description**

This function returns the number of nanoseconds since the UNIX epoch.
Chapter 4. Timestamp Functions

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
gmtimeofday_s:long

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


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”.


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.
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.
Chapter 6. Memory Tapset

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
```
Chapter 6. Memory Tapset

**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**

```c
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**

```c
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 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

Name

probe::vm.kmem_cache_alloc_node — Fires when \n
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
probe::vm.kmem_cache_free — Fires when \n
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
Chapter 6. Memory Tapset

**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
Chapter 6. Memory Tapset

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**

```plaintext
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 remainder. 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**

```plaintext
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**
Chapter 6. Memory Tapset

**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”.

Task Time Tapset

This tapset defines utility functions to query time related properties of the current tasks, translate those in milliseconds 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


Chapter 7. Task Time Tapset

\[
\text{function} \quad \text{task_stime:long()}
\]

**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**

\[
\text{function} \quad \text{task_stime_tid:long}(\text{tid:long})
\]

**Arguments**
\textit{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**

\[
\text{function} \quad \text{cputime_to_msecs:long}(\text{cputime:long})
\]

**Arguments**
\textit{cputime}
- Time to convert to milliseconds.

**Name**
function::msecs_to_string — Human readable string for given milliseconds

**Synopsis**
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

\[ function \text{ task\_time\_string\_tid:}\text{string}(\text{tid:}\text{long}) \]

Arguments

\textit{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”.
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
Chapter 8. IO Scheduler and block IO Tapset

**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.
Chapter 8. IO Scheduler and block IO Tapset

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 completed.

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

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 hardware cannot accept more requests.

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

**Synopsis**

```c
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**

```c
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
Chapter 8. IO Scheduler and block IO Tapset

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-of-bounds error
BIO_SEGMENT_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-of-bounds error
BIO_SEGMENT_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-of-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
Chapter 8. IO Scheduler and block IO Tapset

completion BIO_RW_BLOCK 1 RW_AHEAD set, and read/write would block BIO_EOF 2 out-of-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-of-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.
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)
Chapter 9. SCSI Tapset

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

```c
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
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**

```c
tty.open
```

**Values**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inode_state</td>
<td>the inode state</td>
</tr>
<tr>
<td>file_name</td>
<td>the file name</td>
</tr>
<tr>
<td>file_mode</td>
<td>the file mode</td>
</tr>
<tr>
<td>file_flags</td>
<td>the file flags</td>
</tr>
<tr>
<td>inode_number</td>
<td>the inode number</td>
</tr>
<tr>
<td>inode_flags</td>
<td>the inode flags</td>
</tr>
</tbody>
</table>

**Name**

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

**Synopsis**

```c
tty.release
```

**Values**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inode_state</td>
<td>the inode state</td>
</tr>
<tr>
<td>file_name</td>
<td>the file name</td>
</tr>
<tr>
<td>file_mode</td>
<td>the file mode</td>
</tr>
</tbody>
</table>
Chapter 10. TTY Tapset

- **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 initialized

**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 registered

**Synopsis**

tty.register

**Values**

*driver_name*
the driver name

*name*
the driver .dev_name name

*index*
the tty index requested
Chapter 10. TTY Tapset

module
  the module name

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

Synopsis

```c
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

```c
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

```c
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 treated to the tty

*nr*
  The amount of characters

Name

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

Synopsis
Chapter 10. TTY Tapset

tty.read

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver_name</td>
</tr>
<tr>
<td>the driver name</td>
</tr>
</tbody>
</table>

| buffer |
| the buffer that will receive the characters |

| file_name |
| the file name treated to the tty |

| nr |
| The amount of characters to be read |
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
Chapter 11. Networking Tapset

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
Chapter 11. Networking Tapset

**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**

```c
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**

```c
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**

```c
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
Chapter 11. Networking Tapset

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
Chapter 11. Networking Tapset

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
Chapter 11. Networking Tapset

**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**

```
```

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

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
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
```
Chapter 12. 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**

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

**Synopsis**

```plaintext
socket.sendmsg
```

**Values**

*protocol*

Protocol value

*flags*

Socket flags value

*name*

Name of this probe

*state*

Socket state value

*size*

Message size in bytes
**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.
Chapter 12. Socket Tapset

**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**

```c
socket.aio_write
```

**Values**

*protocol*

Protocol value

*flags*

Socket flags value

*name*

Name of this probe
Chapter 12. 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**
probe::socket.aio_write.return — Conclusion of message send via `sock_aio_write`

**Synopsis**

```
socket.aio_write.return
```

**Values**

\( success \)
Was receive successful? \( 1 = \text{yes}, 0 = \text{no} \)

\( protocol \)
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
Chapter 12. Socket Tapset

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**

```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**

probe::socket.create — Creation of a socket

**Synopsis**

```plaintext
socket.create
```
Chapter 12. 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

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

108
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
Chapter 12. 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

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)
Chapter 12. Socket Tapset

**Arguments**

*state*

The state name.
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
Chapter 13. 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
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.
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 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,
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**

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

**Synopsis**

|  |  |

Values

name
    Name of the probe point

task
    A task handle to the signal recipient

info
    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

```plaintext
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

```plaintext
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

```plaintext
signal.syskill
```
Values

name
Name of the probe point

sig_name
A string representation of the signal

did
The specific signal sent to the process

Values

Name

Name of the probe point

pid_name
The name of the signal recipient

Name

None

Values

Name

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

Synopsis

signal.syskill.return

Values

name
Name of the probe point

Values

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
**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
Chapter 14. Signal Tapset

**Name**
name
   Name of the probe point

**Name**
probe::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

- **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**

```plaintext
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
Chapter 14. Signal Tapset

**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)
Chapter 15. Directory-entry (dentry) Tapset

Arguments

*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

```plaintext
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.
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

\[ \text{function log(msg:string)} \]

Arguments

\( \text{msg} \)

The formatted message string.

General Syntax

\( \text{log(msg:string)} \)

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 efficient as printf and should be used only for urgent messages.

Name

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

Synopsis

\[ \text{function warn(msg:string)} \]

Arguments

\( \text{msg} \)

The formatted message string.

General Syntax

\( \text{warn (msg:string)} \)

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 character is not a newline, the one is added.
Name
function::exit — Start shutting down probing script.

Synopsis

```plaintext
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

```plaintext
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

```plaintext
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.
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 $2^{20}$. 
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

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

Arguments

```plaintext
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

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

Arguments

```plaintext
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)
Chapter 18. String and data retrieving functions Tapset

**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**

```
kern_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**

```
kern_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)
```
Chapter 18. String and data retrieving functions Tapset

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)`
Chapter 18. String and data retrieving functions Tapset

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 “\(<\text{unknown}>\)” 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

```plaintext
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 “\(<\text{unknown}>\)” 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

```plaintext
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_int: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.
Chapter 18. String and data retrieving functions Tapset

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_int_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.
Chapter 18. String and data retrieving functions Tapset

**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.
A collection of standard string functions

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

Name

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

Synopsis

\[
\text{function } strlen: \text{long}(s:\text{string})
\]

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

\[
\text{function } substr: \text{string}(str:\text{string},\text{start:long},\text{length:long})
\]

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)
Chapter 19. A collection of standard string functions

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 the string 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)
Chapter 19. A collection of standard string functions

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

```plaintext
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

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

Arguments
str
String to check.
Chapter 19. A collection of standard string functions

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.
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 foreground color. Black (30), Blue (34), Green (32), Cyan (36), Red (31), Purple (35), Brown (33), Light Gray (37).
Chapter 20. Utility functions for using ansi control chars in logs

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

Synopsis

```plaintext
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 foreground 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

```plaintext
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 foreground 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

```plaintext
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

```plaintext
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.
Chapter 20. Utility functions for using ansi control chars in logs

Synopsis

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

Arguments

- **x**
  - Row to move the cursor to.

- **y**
  - Column to move the cursor to.

General Syntax

```
ansi_cursor_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

```
anzi_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.