# VProbes Programming Reference

VMware Workstation 6.5

vmware®

VProbes Programming Reference
Item: EN-000072-01

You can find the most up-to-date technical documentation on our Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

**VMware, Inc.**
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

# Contents

# About This Book

This *VProbes Programming Reference* documents the VProbes facility and its related VP scripting language, VMware$^{®}$ specific facilities for instrumenting virtual machines.

## Revision History

This book is revised with each release of the product or when necessary. A revised version can contain minor or major changes. Table 1 summarizes the significant changes in each version of this guide.

**Table 1.** Revision History

| Revision | Description |
|---|---|
| 20071213 | First draft of this Tech Note for the Workstation 6.5 Friends and Family release. |
| 20080620 | Second draft of this book for delivery with the Workstation 6.5 Beta 2 release. |
| 20080815 | Added bags to the third draft for delivery with the Workstation 6.5 RC release. |
| 20090105 | Corrected typos in curprocname and curpid example on page 31. |

VMware provides many different SDK products targeting different developer communities and platforms. For information about SDK products, go to http://www.vmware.com/support/pubs/sdk_pubs.html. This is also the place to find the most up-to-date documentation.

## Intended Audience

This book is intended for system programmers, application developers, and performance engineers who must instrument execution details of VMware servers and virtual machines.

## Documentation Resources

To access the current versions of VMware API and SDK documentation, go to:

http://www.vmware.com/support/pubs/sdk_pubs.html

To access the current versions of other VMware manuals, go to:

http://www.vmware.com/support/pubs

### Documentation Feedback

VMware welcomes your suggestions for improving our documentation. Send your feedback to:

docfeedback@vmware.com

## Technical Support and Education Resources

The following sections describe the technical support and educational resources available to you.

### Online Support

You can submit questions or post comments to the Developer Community SDKs and APIs forum, which is monitored by VMware technical support and product teams. To access the forum, go to:

http://communities.vmware.com/community/developer

### Support Offerings

To find out how VMware support offerings can help meet your business needs, go to:

http://www.vmware.com/support/services

### VMware Professional Services

VMware Education Services courses offer extensive hands-on labs, case study examples, and course materials designed to be used as on-the-job reference tools. Courses are available onsite, in the classroom, and live online. For onsite pilot programs and implementation best practices, VMware Consulting Services provides offerings to help you assess, plan, build, and manage your virtual environment. To access information about education classes, certification programs, and consulting services, go to:

http://www.vmware.com/services/

# About VProbes 1

This chapter covers these topics:

## Introduction to VProbes

VProbes allows you to transparently instrument a powered-on guest operating system, its running processes, and VMware virtualization software. You can use VProbes to collect data, both dynamically and statically, about the behavior of guest operating systems and interactions with the VMware monitor.

VProbes is an open-ended investigatory tool. You execute VProbes by writing VP scripts for the `vmrun` utility. VP is a limited language with syntax similar to Scheme or Lisp but few other similarities.

VProbes can inspect and record the state of virtual devices, guest operating systems, and virtual machine executables without modifying their state. Because the VP language has a limited stack size and does not have loop constructs, scripts complete in a finite amount of time, avoiding denial of service to virtual machines. Virtual machines need not be recompiled or shut down before investigation. VProbes dynamically inserts scripts into running virtual machines, removes them, and examines output, repeating this sequence as needed. You can also set static probes at periodic intervals or when important events occur in a virtual machine.

CAUTION The VProbes language and interfaces are experimental in this release. The API might change in subsequent releases, and backward compatibility is not guaranteed.

### Design Goals

VProbes had the following design goals:

- **Safe** – VProbes can inspect, record, and compute state of the guest, the VMM (virtual machine monitor), the VMX (virtual machine executable), and virtual devices, without being able to modify state.
- **Dynamic** – No recompilation or downtime is needed for any part of the virtual machine under investigation. To work with VProbes you write simple scripts, dynamically insert them into and remove them from already-running virtual machines, examine the output, and repeat.
- **Static** – Predefined probes are available for execution at periodic intervals, or when critical events occur.
- **Operating system independent** – Differences between guest operating systems are made transparent when possible. For example, most operating systems assign names and numeric identifiers to processes. VP provides the `curprocname` and `curpid` facilities to enable access to this information.
- **Free when disabled** – Being intended for use in production systems, when instrumentation is not enabled, VProbes has no cost in either memory space or CPU time.

These goals are similar to those of the Solaris `DTrace` facility, which was an important influence on VProbes.

## Getting Started with VProbes

This section describes how to enable VProbes and run a VP script.

### Enabling VProbes

Before enabling VProbes, you or the system administrator must add the following line to the VMware host configuration file. This line grants individual users access to VProbes:

```
vprobe.allow = TRUE
```

The host configuration file is located:

- On Linux and ESX:
  `/etc/vmware/config`
- On Windows:
  `C:\Documents and Settings\All Users\Application Data\VMware\VMware Workstation\config.ini`
- On Fusion:
  `/Library/Application Support/VMware Fusion/config`

You can then enable VProbes on a virtual machine by adding the following line to its `.vmx` configuration file in the virtual machine directory. This line enables VProbes:

```
vprobe.enable = TRUE
```

### Components of a VP Script

A VP script can contain one or more probes. Each probe specifies an event and instrumentation code to run when that event occurs. In other words, VP scripts are not executed linearly but rather as a series of callbacks to individual probes when specific events occur. You define probes using the `vprobe` statement. VP scripts use the file extension `.vp`, as shown in section "Running a VP Script" on page 9.

For example, the following hello probe intercepts the VMM1Hz event and calls the `printf` function whenever VMM1Hz occurs, which is once per second.

```
(vprobe VMM1Hz
   (printf "hello!\n"))
```

To print a complete list of events that can be intercepted (also called probe points) for a virtual machine, use the `vmrun` utility's `vprobeListProbes` command:

```
vmrun vprobeListProbes my.vmx
```

VProbes-intercepted events fall into two categories:

- **Static probe points** – Predefined virtual hardware events, such as `VMM1Hz` above, or the delivery of a virtual hardware interrupt to the guest, `GuestIRQ`. For a list, see "Supported Static Probes" on page 28.
- **Dynamic probe points** – Run when the guest execution reaches a given address or symbol in the guest.

  Dynamic probes are specified using the `GUEST:` prefix, followed by a symbol or an address. For example, the following script prints "`Hi from GUEST:system_call`" every time the function `system_call` is executed in a Linux guest. It also prints "`reached 0xc0106ae0`" when the guest reaches the location specified by its hex address.

  ```
  (vprobe GUEST:system_call
    (printf "Hi from %s\n" PROBENAME))
  (vprobe GUEST:0xc0106ae0
    (printf "reached 0xc0106ae0\n"))
  ```

  NOTE For dynamic probe points to work with guest symbols such as `system_call`, you must create a symbol file; see "Symbol Files" on page 9

## Running a VP Script

Use the `vmrun` program to execute VProbes scripts from the command line. The `vmrun` program comes with VMware Workstation, VMware Fusion, VMware Server, and the VIX API. Table 1-1 shows the `vmrun` commands for running VProbes.

**Table 1-1.** VProbes Commands in vmrun

| Command | Description |
|---|---|
| `vprobeLoad` | Load a VP script. |
| `vprobeReset` | Disable all running VProbes. |
| `vprobeListProbes` | List available probe points. |
| `vprobeListGlobals` | List global variables. |
| `vprobeVersion` | Display the current VProbes version. |

For example, the following command loads the script given as an argument on the command line into the running virtual machine specified by `my.vmx`, which is the location of the virtual machine's configuration file. The script runs once a second and prints the string `"hello!"` followed by a newline. By default, the output from VProbes is redirected to a file called `vprobe.out` in the virtual machine directory.

```
vmrun vprobeLoad my.vmx '(vprobe VMM1Hz (printf "hello!\n"))'
```

Because the command tool on Windows (`cmd`) permits nesting only of like-type quotes – either single or double quotes but not both – the above `vmrun` command produces the error message "unknown ident windows" and fails. To avoid this behavior, you can install Cygwin and run VP scripts in a standard `bash` shell.

The following command runs the VP script in the file `test.vp`:

```
vmrun vprobeLoad my.vmx "`cat test.vp`"
```

The pair of backquotes (`) interpolate output of the `cat` command into the quoted command-line argument. This works in Bourne and C shell compatible environments, including Cygwin.

## Symbol Files

For dynamic guest probes to work with guest symbols, you must create a symbol file.

On Linux guests, create a symbol file by saving the output of the following command to a text file on the host:

```
cat /proc/kallsyms
```

The `kallsyms` file contains the exported kernel symbol definitions so the Linux kernel modules facility can dynamically link and bind loadable modules.

On Windows, use the WinDbg (Windows debugger) command to extract symbols. WinDbg must be in kernel debugging mode, and `<moduleName>` should be a kernel module:

```
x <moduleName>!*
```

After creating the symbol file on either Linux or Windows, you must add the following line to the virtual machine configuration (`.vmx`) file:

```
vprobe.guestSyms = "<symbolFileName>"
```

The symbol file can have a `.txt` extension.

You can put probes on user-level code in a guest virtual machine, although it is less convenient than putting probes on kernel symbols. You can set a probe on a linear address and get the process name from inside the probe, as in the following VP script example:

```
(vprobe GUEST:0x1234
  (cond ((!(strcmp (curprocname) "app")) (printf "0x1234 in app"))))
```

The not string-compare clause `(!(strcmp))` checks for zero, indicating that the strings are equal. The `printf` keyword prints the probe's address in the process. You can also set kernel-level probes at linear addresses.

Determining the guest address where you want to set the probe is not simple. In this example, the probe fires when the `app` process reaches the `GUEST:0x1234` address. See "curprocname" on page 25 for information about `curprocname`, and "Sample Implementation of curprocname and curpid" on page 31 for an example.

## The Emmett Language

VMware delivers VProbes in products with the `vmrun` utility to support VP scripts. If you prefer a higher level language interface, you have an alternative.

Emmett is a C-like language that allows to you program in high-level constructs and compile your code into VP scripts. It is open source and available for download from the SourceForge Web site. For more information, see Appendix B, "Emmett – A High-Level VProbes Tool," on page 33.

As you can see from the "Emmett Code Sample" on page 34, Emmett is easy to read and understand for programmers accustomed to curly-brace languages like C, Perl, Java, or C#.

# Syntax and Functions Reference 2

This chapter includes the following main sections:

## Syntax

The syntax of VP is similar to that of Scheme or Lisp, but the language is far more limited. You can define variables and functions. Recursive function calls are permitted, but stack space is limited, so probes that recurse too deeply are terminated at runtime.

### Comments

Comments begin with a semicolon and continue to the end of the line.

```
; This is a comment
(setint a 42) ; So is this
```

### Literals

VP supports two types of literals: strings and integers. Both types are similar in appearance and functionality to those found in most programming languages.

- **Strings** – Surround strings with double quotes, using the following escapes: `\n`, `\\`, `\"`, and `\t`. If another backslash escape is used, the backslash is omitted from the string, but the other character is not.

  ```
  "This is a \"string\""
  "So's this\tover here\n"
  "A newline in VP is '\\n'"
  "This is a backslash: \\. This is not: \."
  ```

- **Integers** – Integers are unsigned 64-bit quantities. They can be expressed as either decimal (base 10) or hexadecimal (prefaced by 0x) values. Decimal values can include a leading + or - sign. Floating point is not supported. You can mix hexadecimal and decimal together in an expression.

  ```
  63
  0xffffffc0
  -65536
  +54
  ```

### Statements

Expressions and statements begin with open parenthesis and end with close parenthesis.

## User-Defined Variables

As with most programming languages, you can create variables of various types (integer, string, aggregate). For variable names, you can use any characters except space, parentheses, and all-capitals. Names containing all uppercase letters are reserved by VProbes, so avoid using them for user-defined functions or variables.

The variable type is fixed at declaration time and converting between types is not allowed.

### Integer Variables

You call `definteger` to create an integer variable, which contains an unsigned 64-bit value. The first form of `definteger` allocates space for a named integer variable:

```
(definteger <NAME>)
```

The second form of `definteger` allocates space for the named variable, and assigns `<INITIAL_VALUE>` to it:

```
(definteger <NAME> <INITIAL_VALUE>)
```

Call `setint` to set or reset the value:

```
(setint a 23)
(setint b (& 0xff (>> RAX 56)))
```

In addition to user-defined variables, VProbes also exposes several built-in variables. Generally, these variables provide read-only access to some part of the virtual hardware state, such as the current state of general-purpose registers in the guest. For more information, see "Built-in Global Variables" on page 14.

### String Variables

You call `defstring` to create a named character string of arbitrary length. The first form of `defstring` defines a named string variable:

```
(defstring <NAME>)
```

The second form of `defstring` defines the named string, and allocates space for string `<INITIAL_VALUE>`:

```
(defstring <NAME> "<INITIAL_VALUE>")
```

Assignment to strings is through `setstr`:

```
(setstr string-var "A string\n")
```

### Aggregate and Bag Variables

So far, `printf` is the only method that has been introduced to record data from a probe. However, `printf` is poorly suited for frequently executed probes.

First, the high volume of output requires frequent trips out of the virtual machine monitor and into the user-level output engine, increasing probe cost and diminishing performance of the running virtual machine.

Second, frequent output of data is usually not necessary. In most commonly executed probes, a small set of recorded values accounts for the majority of observations. For example, in a profiler, the "hot path" of the guest workload accounts for nearly all the instruction pointer values seen from a frequently executed probe. In such cases, the overall distribution of values is more useful than each individual value.

To address these problems, VProbes provides aggregate and bag variables. Aggregate variables provide a general method of buffering recorded values. Bag variables provide temporary storage space.

An aggregate is a hash-like data structure that contains integer values. An arbitrary number of integer and string keys can be used as indexes. The aggregate structure is perhaps the most useful VProbes data type, because it makes possible a wide variety of applications that would otherwise be difficult to script.

## Aggregate Variables

You define aggregates with `defaggr`, specifying the name, number of integer keys, and number of string keys:

```
(defaggr a 1 0)   ; Aggregate a with one integer key
(defaggr b 0 1)   ; Aggregate b with one string key
(defaggr c 2 1)   ; Aggregate c with two integer keys and one string key
```

You can add to an aggregate using the `aggr` statement. The `aggr` statement takes the following arguments:

- Name of the aggregate
- List containing the integer keys
- List containing the string keys
- Integer value to add to the aggregate

If an aggregate lacks either integer keys or string keys, insert an empty list where the keys would otherwise appear. The values in either the string or the integer list can be literals, variables of the appropriate type, or expressions returning the appropriate type.

```
(aggr a (CR2) () 1)        ; add 1 to a[CR2]
(aggr b () ("string") 5)   ; add 5 to b["string"]
```

Aggregate variables can be read only through the `logaggr` statement. The `logaggr` statement saves the entire contents of the aggregate to the log in `vprobe.out`.

```
(logaggr a)
```

The `clearaggr` statement deletes the contents of an aggregate. The `logaggr` and `clearaggr` statements are often found together, especially within the `VMM1Hz` static probe.

```
(clearaggr a)
```

Here are a few examples of tasks facilitated by aggregates:

```
; Track the number of #PF (page faults) by fault address. On the x86,
; the fault address during a page fault is contained in the CR2 register.
(defaggr pf 1 0)
(vprobe Guest_PF
        (aggr pf (CR2) () 1))
(vprobe VMM1Hz
        (logaggr pf)
        (clearaggr pf))
```

Sample output:

```
aggr: pf (1 integer key) (0 string keys)
  pf[0xfffff9800bf8e000] == avg 1 count 1 min 1 max 1 latest 1
  pf[0xfffff9800408a01c] == avg 1 count 1 min 1 max 1 latest 1
  pf[0xfffff980044ebded] == avg 1 count 1 min 1 max 1 latest 1
  pf[0xfffff9800984c410] == avg 1 count 1 min 1 max 1 latest 1
```

That is, in the past second, the guest took four pagefaults at four different addresses.

```
; Watch guest IRQs. ARG0 contains the interrupt vector.
(defaggr irqs 1 0)
(vprobe Guest_IRQ
        (aggr irqs (ARG0) () 1))
(vprobe VMM1Hz
        (logaggr irqs)
        (clearaggr irqs))
```

This sample output indicates that, in the past second, the guest took one IRQ with vector 0xa1, seven IRQs with vector 0xa9, and seven IRQs with vector 0xef:

```
aggr: irqs (1 integer key) (0 string keys)
  irqs[0xa1] == avg 1 count 1 min 1 max 1 latest 1
  irqs[0xa9] == avg 1 count 7 min 1 max 1 latest 1
  irqs[0xef] == avg 1 count 7 min 1 max 1 latest 1
```

### Bag Variables

A bag is a temporary scratch location for storing integer key-value pairs. You declare bags with `defbag`, specifying name and size. The size declares the number of key-value pairs it can hold.

```
(defbag bagVar size)
```

You can configure the number of bytes available for bag storage using the `vprobe.bagBytes` option in the virtual machine's `.vmx` file.

You insert key-value pairs with `baginsert`, which returns 1 on success and 0 on failure. Duplicate keys are allowed. An insert fails if configured bag size is exceeded or if the total storage is full at the time of insert.

```
(baginsert bagVar key val)
```

You remove a key-value pair with `bagremove`, which deletes the key-value pair from the bag, and returns the (former) value for the specified key.

```
(bagremove bagVar key)
```

If you call `bagremove` twice on the same bag variable within a short time, a race condition might result, so multiple `bagremove` operations would return the same value. However given a successful `baginsert`, it is guaranteed that at least one `bagremove` operation with the key from `baginsert` will succeed.

## Built-in Global Variables

Built-in variables provide access to some portion of the virtual hardware state. These variables are read-only. Each variable has the following components:

- Name, in all uppercase letters, for example `RIP`.
- Type, either a 64-bit integer or a string.
- Scope, which can be per-VCPU (virtual CPU), per-PCPU (physical CPU), or global.

  In a multiprocessor virtual machine, a probe might fire on more than one physical or virtual processor, perhaps concurrently. Per-VCPU variables can vary from VCPU to VCPU, and per-PCPU variables depend on the physical processor where the probe fires. Global variables have a single value.

### Variables for Virtual CPU Registers

Table 2-1 lists the built-in variables for virtual CPU registers.

**Table 2-1.** Virtual CPU Registers

| Name | Type | Scope | Description |
|---|---|---|---|
| RIP | Integer | Per-VCPU | Value of the running virtual CPU's instruction pointer register. |
| RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, R10, R11, R12, R13, R14, R15 | Integer | Per-VCPU | Value of the running virtual CPU's corresponding general purpose register. |
| RFLAGS | Integer | Per-VCPU | Value of the running virtual CPU's flags register. |
| CS, SS, DS, ES, FS, GS | Integer | Per-VCPU | Value of the running virtual CPU's corresponding segment register. |
| CR0, CR2, CR3, CR4, CR8 | Integer | Per-VCPU | Value of the running virtual CPU's control registers. |
| DR6, DR7 | Integer | Per-VCPU | Value of the running virtual CPU's debug registers. |
| EFER | Integer | Per-VCPU | Value of the running virtual CPU's EFER registers. |
| KERNELGSBASE | Integer | Per-VCPU | Value of the running virtual CPU's KERNELGSBASE MSR. |
| FSBASE, GSBASE | Integer | Per-VCPU | Base of the running virtual CPU's FS, GS segment descriptors. |
| APIC_BASEPA | Integer | Per-VCPU | Physical address of the virtual APIC. |

## Variables for Hardware Data

Table 2-2 lists the built-in variables for hardware data.

**Table 2-2.** Hardware Data

| Name | Type | Scope | Description |
|---|---|---|---|
| TSC | Integer | Per-PCPU | Current 64-bit value of the hardware time stamp counter, as returned by the RDTSC instruction on the physical CPU where the probe fires. |
| TSC_HZ | Integer | Per-PCPU | Frequency of the TSC on the physical CPU where the probe fires. |
| PTSC | Integer | Global | Value of the pseudo time stamp counter abstraction. The PTSC behaves similarly to the TSC, except it is guaranteed to increase monotonically over time and appears the same on all physical CPUs in the system. In contrast, the TSC is sensitive to power-management related changes in clock speed, and each CPU in the system can contain a different TSC value. |
| VCPUID | Integer | Per-VCPU | Unique integer identifying the current virtual CPU. |
| PCPUID | Integer | Per-PCPU | Unique integer identifying the current physical CPU. |
| THREADID | Integer | Per-Thread | Unique integer identifying the current user-level thread. |
| NUMVCPUS | Integer | Global | Count of virtual CPUs in the virtual machine. |

All the built-in global variables listed in Table 2-1 and Table 2-2 correspond to architectural state. Refer to either of the following for more information:

- *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*
- *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*

## Hardware Virtualization State

When hardware virtualization (HV) is in use, a variety of information about HV state is present. Because Intel and AMD offer different facilities for hardware virtualization (VT and AMD-V, respectively), the HV variables available reflect the vendor of the physical CPU.

### Intel Virtual Machine Control Structure (VMCS)

The variables in Table 2-3 are available when running a hardware-virtualization-enabled guest on an Intel processor. Documentation for these variables is available in the *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide* in tables H-5, H-6, and H-10.

**Table 2-3.** Built-in Variables for Intel Processors

| Name | Type |
|---|---|
| VMCS_EXIT_REASON | Integer |
| VMCS_INTR_INFO | Integer |
| VMCS_INTR_ERR | Integer |
| VMCS_IDTVEC_INFO | Integer |
| VMCS_IDTVEC_ERR | Integer |
| VMCS_INSTRLEN | Integer |
| VMCS_VMENTRY_INTR_INFO | Integer |
| VMCS_VMENTRYXCP_ERR | Integer |
| VMCS_EXIT_QUAL | Integer |

### AMD Virtual Machine Control Block (VMCB) State Save Area

The variables in Table 2-4 are available when running a hardware-virtualization-enabled guest on an AMD processor. Documentation for these variables is available in the *AMD64 Architecture Programmer's Manual, Volume 2: System Programming* in table B-1.

**Table 2-4.** Built-in Variables for AMD Processors

| Name | Type |
|---|---|
| VMCB_EXITCODE | Integer |
| VMCB_EXITINFO1 | Integer |
| VMCB_EXITINFO2 | Integer |
| VMCB_EXITINTINFO | Integer |
| VMCB_EVENTINJ | Integer |
| VMCB_TLBCTL | Integer |
| VMCB_VAPIC | Integer |

## Probes

The `vprobe` keyword defines the entry point of probe execution:

```
(vprobe <PROBE> expressions)
```

Each VP script must contain one or more probes. You can insert a probe multiple times; duplicate probes are executed one after the other.

The following probe prints a message when the guest operating system powers on and executes its boot sector:

```
(vprobe GUEST:0x7c00
  (printf "Executing boot sector!\n"))
```

Probes can be divided into four types: static, dynamic, data, and periodic.

### Static Probes

Static probes are predefined probes that execute at implementation points or architecturally significant points. For example, `Disk_IOStart` and `Disk_IOFinish` execute when virtual disk I/O begins and completes.

Some static probes, for example `In` and `Out`, have arguments passed to them by the execution engine. These arguments are contained in global variables designated `ARG<n>`, where `<n>` is a number from 0 to 9. The table of supported static probes contains the argument list, if any, of each static probe. Here is a sample script with three static probes:

```
; Calculate the latency of disk I/O in microseconds.
(definteger tsc-start)
(defaggr latencies 0 1)
(defun usec (ticks)
  (/ (* 1000000 ticks) TSC_HZ))
(vprobe Disk_IOStart
  (setint tsc-start TSC))
(vprobe Disk_IOFinish
  (aggr latencies () ("uSec") (usec (- TSC tsc-start))))
(vprobe VMM1Hz
  (logaggr latencies)
  (clearaggr latencies))
```

Here is sample output from the above script:

```
aggr: latencies (0 integer keys) (1 string key)
  latencies["uSec"] == avg 26 count 1 min 26 max 26 latest 26
aggr: latencies (0 integer keys) (1 string key)
  latencies["uSec"] == avg 46 count 9 min 28 max 84 latest 30
aggr: latencies (0 integer keys) (1 string key)
  latencies["uSec"] == avg 49 count 78 min 11 max 319 latest 17
```

See Table 3-3, "Static Probes," on page 28 for a list of supported static probes.

## Dynamic Probes

Dynamic probes run when the guest executes an instruction at the specified probe point, which is the guest linear address. The syntax of a dynamic probe point is `GUEST:<guest linear address>`. The following script uses a dynamic probe to determine the guest's boot device:

```
; Print the boot device.
(defstring device)
(definteger dl)
(vprobe GUEST:0x7c00
  (setint dl (& RDX 0xff))
  (cond ((== dl 0x80)
         (setstr device "hard drive"))
        ((== dl 0)
         (setstr device "floppy drive"))
        (1
         (setstr device "unknown device")))
  (printf "Booting from %s (0x%x)\n" device dl))
```

Here is sample output from the above script:

```
Booting from hard drive (0x80)
```

## Data Probes

Data probes are executed when a guest linear address is either read from or written to, depending on the type of data probe. The syntax for a read probe point is `GUEST_READ:<guest linear address>`, and the syntax for guest a probe point is `GUEST_WRITE:<guest linear address>`. Here is a sample script with guest write data probes:

```
(vprobe GUEST_WRITE:0xa0000
  (printf "Write to VGA graphics RAM.\n"))
(vprobe GUEST_WRITE:0xb8000
  (printf "Write to VGA text RAM.\n"))
```

Here is sample output from the above script:

```
Write to VGA graphics RAM.
Write to VGA text RAM.
Write to VGA graphics RAM.
Write to VGA text RAM.
```

## Periodic Probes

Periodic probes are executed at arbitrary microsecond intervals. The syntax for a periodic probe point is `USEC:<microsecond interval>`. Periodic probes are useful for monitoring changes over time in guest state. The following script has a periodic probe that executes every one-third of a second:

```
; Display the elapsed seconds three times per second.
(definteger tickCount 0)
(vprobe USEC:333333
  (setint tickCount (+ tickCount 333333))
  (printf "%d seconds have elapsed.\n" (/ tickCount 1000000)))
```

Here is sample output from the above script:

```
0 seconds have elapsed.
0 seconds have elapsed.
0 seconds have elapsed.
1 seconds have elapsed.
1 seconds have elapsed.
1 seconds have elapsed.
2 seconds have elapsed.
2 seconds have elapsed.
2 seconds have elapsed.
```

## Conditional Expressions

VP's equivalent of `if` and `switch` statements from other languages is `cond`. The `cond` expression takes one or more lists, each of which contain two expressions, and iterates through them sequentially, evaluating the first expression in each list, and, if it evaluates to true (non-zero), the second expression in that list is evaluated and the remainder of the lists in the `cond` expression are skipped.

The general form of a `cond` expression is:

```
(cond (condition0 expression0)
      (condition1 expression1)
      ...
      (conditionN expressionN))
```

For example:

```
   (cond ((== 1 0) (printf "A"))
         ((== 1 1) (printf "B"))
         (1        (printf "C")))
```

would print `"B"` to the log because `(== 1 1)` is the first expression to evaluate as TRUE,

```
   (cond ((== 1 0) (printf "A"))
         ((== 0 1) (printf "B")))
```

would not print anything to the log because nothing evaluated as TRUE, and

```
   (cond ((== 1 0) (printf "A"))
         (1        (printf "B")))
```

would print `"B"` to the log because 1 is TRUE.

The result of a `cond` expression is the result of the expression that executed.

## Do Expressions

In some circumstances, such as the expressions in a conditional, only one expression is allowed syntactically. In these situations, the `do` operator can execute more than one expression sequentially. In a `do` expression, the first element is `do`, and all the remaining elements are themselves expressions.

```
(do (expression1)
    (expression2)
    ...
    (expressionN))
```

The result of the entire `do` expression is the result of the last expression evaluated.

```
; This logs the integer value 9 because (* 3 3) is the last expression evaluated.
(logint (do (+ 1 1)
            (* 3 3)))
```

## Functions

User-defined functions, built-in operators, and other built-in functions are all treated the same syntactically. The first element in the expression is the function, and the remaining elements are the arguments to the function. The return value of a function can be either an integer or a string, depending on how the function evaluates. The syntax for calling a function is:

```
(function-name arg0 arg1 ...)
```

### User-Defined Functions

You create user-defined functions using the `defun` statement. A user-defined function can take zero or more arguments. You do not explicitly specify the return type of a user-defined function; the return type results from the last expression evaluated in the function. The general syntax for creating a user-defined function is:

```
(defun function_name (arguments)
       expressions)
```

For example:

```
; No arguments, returns integer.
(defun return-cr0 ()
  CR0)
; Two arguments, returns integer.
(defun add-two (a b)
  (+ a b))
; No arguments, returns string.
(defun logmarker ()
  (logaggr () ("marker"))
```

## Built-in Operators

In VP, built-in functions are used in the same manner as operators in other languages:

```
(operator exp1 exp2)
```

Arguments `exp1` and `exp2` can be an integer literal, an integer variable, or an expression that evaluates to an integer. Most operator type functions have two names, one alphabetic and one symbolic, such as `div` and `/`. Either form is acceptable, although the symbolic form is more common.

The following tables list the various types of operators.

- Table 2-5, "Arithmetic Operations," on page 19
- Table 2-6, "Bit Operations," on page 19
- Table 2-7, "Logical Operations," on page 19
- Table 2-8, "Comparison Operations," on page 20

**Table 2-5.** Arithmetic Operations

| Operator | Alphabetic Name | Operation | Example |
|---|---|---|---|
| + | `add` | Addition | `(+ a 23)` |
| – | `sub` | Subtraction | `(– 53 0xa)` |
| * | `mul` | Multiplication | `(* 6 7)` |
| / | `div` | Division | `(/ 10 3) ; result is 3` |
| % | `mod` | Modulo | `(% RAX 2)` |

**Table 2-6.** Bit Operations

| Operator | Alphabetic Name | Operation | Example |
|---|---|---|---|
| << | `lshift` | Left shift | `(<< 1 31)` |
| >> | `rshift` | Right shift | `(>> CR0 31)` |
| ^ | `xor` | Exclusive or | `(^ var1 var2)` |
| ~ |  | Bitwise not | `(~ 0x5e5e) ; Unary operator` |
| \| |  | Bitwise or | `(\| pte 0xc00)` |
| & |  | Bitwise and | `(& CR0 0x80000000)` |

**Table 2-7.** Logical Operations

| Operator | Alphabetic Name | Operation | Example |
|---|---|---|---|
| && | `and` | Logical and | `(&& flag-a flag-b)` |
| \|\| | `or` | Logical or | `(\|\| flag-a flag-b)` |
| ! | `not` | Logical not | `(! now) ; Unary operator` |

**Table 2-8.** Comparison Operations

| Operator | Alphabetic Name | Operation | Example |
|---|---|---|---|
| `>` | | Greater than | `(> CR3 0x10000)` |
| `<` | | Less than | `(< addr 0xffffffff)` |
| `>=` | | Greater than or equal to | `(>= a b)` |
| `<=` | | Less than or equal to | `(<= ptr 0xb8000)` |
| `==` | | Equal to | `(== addr invalid-addr)` |
| `!=` | | Not equal to | `(!= CR3 pt)` |

Both `||` and && are shortcut operators and work in a similar manner as the corresponding operators in other languages, such as C, Java, and Perl. The second argument to && is not evaluated if the first argument evaluates false (zero), and that the second argument to `||` is not evaluated if the first argument evaluates true (non-zero).

## Built-In Functions

Built-in functions can be of type STRING or INTEGER, depending on whether they evaluate to a string or an integer. The following function definitions note the type, along with the types of arguments the function takes. If a function takes different argument lists, multiple argument lists are documented. When a parameter to a function is specified as being of a certain type, that means that the parameter can either be a literal of that type, a variable of that type, or an expression that evaluates to that type.

### logstr

Print a string to the log.

Return type: INTEGER

Arguments:

STRING: The string to print to the log.

The return value indicates success (1) or failure (0).

Sample script:

```
; Tell the world hello once per second.
(vprobe VMM1Hz
    (logstr "Hello world!\n"))
```

Sample output:

```
Hello world!
Hello world!
Hello world!
```

### logint

Print an integer to the log.

Return type: INTEGER

Arguments:

INTEGER: The integer to print to the log.

The return value indicates success (1) or failure (0).

Sample script:

```
; Log the value of CR3 before the guest modifies the register.

(vprobe Guest_CR3Write
        (logint CR3)
        (logstr "\n"))
```

Sample output:

```
160600064
515477504
160600064
139001856
399241216
```

## sprintf

Store formatted data in a string variable.

Return type: INTEGER

Arguments:

STRING VARIABLE: Destination variable.

STRING: Format string.

Zero or more arguments, depending on the contents of the format string.

`sprintf` works in the same manner as the C library's `sprintf`. The return value indicates success (1) or failure (0).

Sample script:

```
; Log the guest's instruction pointer (RIP) ten times per second
(defstring my-rip)
(vprobe VMM10Hz
        (sprintf my-rip "Current RIP is 0x%016x\n" RIP)
        (logstr my-rip))
```

Sample output:

```
Current RIP is 0x0000000000002c21
Current RIP is 0xfffff8000189c0f1
Current RIP is 0xfffff80001ce9133
Current RIP is 0xfffff80001857670
```

## printf

Print formatted data to the log.

Return type: INTEGER

Arguments:

STRING VARIABLE: Destination variable.

STRING: Format string.

Zero or more arguments, depending on the contents of the format string.

`printf` works in the same manner as the C library's `printf`, except that its output is directed to the log instead of `stdout`. The return value indicates success (1) or failure (0).

Sample script:

```
; Log the guest's instruction pointer (RIP) ten times per second
(vprobe VMM10Hz
        (printf "Current RIP is 0x%016x" RIP))
```

Sample output:

```
Current RIP is 0xfffff98002ad5588
Current RIP is 0xfffff80001ce9131
Current RIP is 0xfffff80001857674
Current RIP is 0xfffff80001ce9131
```

### strcmp

Compare two strings lexicographically. The value returned is less than zero if the first string is less than the second, zero if they are equal, and greater than zero if the first string is greater than the second.

Return type: INTEGER

Arguments:

STRING_1

STRING_2

Sample script:

```
; VP script that keeps track of how a guest is doing.

; Subfunction that returns the string "Calamity!" if it was called during execution
; of Guest_TripleFault (triple fault event), and "OK!" otherwise.
(defun event ()
   (cond ((== 0 (strcmp PROBENAME "Guest_TripleFault"))
          "Calamity!")
         (1  ; default case
          "OK!")))
(vprobe VMM1Hz
        (printf "Guest status is %s\n" (event)))
(vprobe Guest_TripleFault
        (printf "Guest status is %s\n" (event)))
```

Sample output:

```
Guest status is OK!
Guest status is OK!
Guest status is OK!
```

### getguest

Copy 8 bytes of memory from a linear address in the guest address space. If the address is invalid (that is, if it is not mapped in), the current probe is cancelled.

Return type: INTEGER

Arguments:

INTEGER: Linear address to access.

STRING: Guest symbol to access

The return value is the 8 bytes of memory as an integer.

Sample script:

```
; Print the speed of the Time Stamp Counter (TSC) if it is enabled for
; Linux 2.6 kernels.

; Both tsc_enabled and current_tsc_khz are 4 bytes, so they must be masked when read.
(definteger tsc_enabled)
(definteger tsc_khz)
(vprobe VMM1Hz
        (setint tsc_enabled (& 0xffffffff (getguest "tsc_enabled")))
        (cond (tsc_enabled
               (do (setint tsc_khz (& 0xffffffff (getguest "current_tsc_khz")))
                   (printf "TSC kHz: %d\n" tsc_khz)))
              (1
               (printf "TSC not enabled\n"))))
```

Sample output:

```
TSC kHz: 2992171
TSC kHz: 2992171
TSC kHz: 2992171
```

### getgueststr

Copy a NULL-terminated sequence of bytes from a linear address in the guest address space, or a guest symbol name with an optional offset, to a string variable. If a guest symbol name is used, then a guest symbol table must have been provided at power-on time using the `vprobe.guestSyms` option in the `.vmx` file. The length of the resulting string is limited to 256 bytes. If the address is invalid (that is, not currently mapped in) or the guest symbol name does not exist, the current probe is cancelled.

Return type: INTEGER

Arguments:

STRING VARIABLE: Destination for string.

INTEGER: Linear address.

or

STRING VARIABLE: Destination for string.

STRING: Guest symbol.

INTEGER: Optional offset from the guest symbol.

The return value indicates success (1) or failure (0).

Sample script:

```
; Print the saved Linux command line for 32 bit Linux.
(defstring command_line)
(definteger saved_command_line)
(vprobe VMM1Hz
        (setint saved_command_line (& 0xffffffff (getguest "saved_command_line")))
        (getgueststr command_line saved_command_line)
        (printf "Linux command line (at %#x):\n%s\n" saved_command_line command_line))
```

Sample output:

```
Linux command line (at 0xc1402000):
BOOT_IMAGE=/casper/vmlinuz file=/cdrom/preseed/ubuntu.seed boot=casper
               initrd=/casper/initrd.gz quiet splash --
```

### gueststack

Obtain the current guest stack. If the guest is not using frame pointers, the results have little or no meaning.

Return type: INTEGER

Arguments:

STRING VARIABLE: Destination for stack information.

The return value indicates success (1) or failure (0).

Sample script:

```
; Print the guest's stack once per second.
(defstring stack)
(vprobe VMM1Hz
   (gueststack stack)
   (printf "Stack: %s\n" stack))
```

Sample output:

```
Stack:GUEST_0xfffffffffb8006fc_0xfffffffffb823415_0xfffffffffb800875_0xfefdef7e
Stack:GUEST_0xfffffffffb9c24f7_0xfffffffffb9c107f_0xfffffffffb9c14ae_0xfffffffffb801383
Stack:GUEST_0xfffffffffb819562_0xfffffffffb8397f4_0xfffffffffb831998
Stack:GUEST_0xfffffffffb800a75_0xfffffffffb823415_0xfffffffffb800a5a_0x8045ac808045ab0
Stack:GUEST_0xfec8eb35_0xfec8a4bc_0xfec8a6bc_0xfec938e7_0xfec9099b_0xfec90aa7_0xfec974c6_0xfe
               c97bfe_0xfec86bed_0xfec863f8_0xfec834c3_0xfee6f1ff_0x809f258_0x809f6b1_0x80989
               5d_0x81652ed_0x809eda7_0x80b2a51_0x8077d40
```

## getguestphys

Copy 8 bytes of memory from a physical address in the guest address space. If the address is invalid, the current probe is cancelled.

Return type: INTEGER

Arguments:

INTEGER: Guest physical address to access.

Sample script:

```
; Read the APIC version.
(vprobe VMMLoad
  (printf "VCPU %d: APIC version = %x\n" VCPUID
          (getguestphys (+ APIC_BASEPA 48))))
```

Sample output:

```
VCPU 0: APIC version = 40011
```

## getsystemtime

Return the host time in microseconds.

Return type: INTEGER

Arguments:

INTEGER VARIABLE: Destination for system time.

The return value indicates success (1) or failure (0).

Sample script:

```
; Approximate the TSC frequency.
(definteger lastTsc 0)
(definteger tsc 0)
(definteger lastSystemTime 0)
(definteger systemTime 0)
(definteger tscFreq 0)

(defun readTime ()
  (setint lastSystemTime systemTime)
  (setint lastTsc tsc)
  (getsystemtime systemTime)
  (setint tsc TSC))
(defun calculateFreq (tscDiff systemTimeDiff)
  (cond ((!= 0 systemTimeDiff)
         (/ (* 1000000 tscDiff) systemTimeDiff))
        (1
         0)))
(vprobe VMX1Hz
  (readTime)
  (cond ((!= 0 lastTsc)
         (setint tscFreq (calculateFreq (- tsc lastTsc)
                                        (- systemTime lastSystemTime)))))
  (printf "Calculated TSC HZ %d, Actual TSC HZ %d (difference %d)\n"
          tscFreq TSC_HZ (- TSC_HZ tscFreq)))
```

Sample output:

```
Calculated TSC HZ 2593105783, Actual TSC HZ 2593119000 (difference 13217)
Calculated TSC HZ 2593105624, Actual TSC HZ 2593119000 (difference 13376)
Calculated TSC HZ 2593104806, Actual TSC HZ 2593119000 (difference 14194)
Calculated TSC HZ 2593104154, Actual TSC HZ 2593119000 (difference 14846)
Calculated TSC HZ 2593106513, Actual TSC HZ 2593119000 (difference 12487)
```

## Guest-Specific Functions

Several functions are not built into the language but are conventionally provided by user-defined, guest-specific functions. See "Sample Implementation of curprocname and curpid" on page 31 for an example of `curprocname` and `curpid` defined for 64-bit Linux 2.6 series kernels.

### curpid

Get the process ID of the current process within the guest.

Return type: INTEGER

Arguments: None

The return value is the current process ID.

Sample script:

```
; Print the PID of the process running
; when a hardware interrupt is being delivered.
(vprobe Guest_IRQ
        (printf "Current PID during IRQ: %d\n" (curpid)))
```

Sample output:

```
Current PID during IRQ: 0
Current PID during IRQ: 1978
Current PID during IRQ: 0
```

### curprocname

Get the process name of the current process within the guest.

Return type: STRING

Arguments: None

Sample script:

```
; Print the name of the process running
; when a hardware interrupt
; is being delivered.
(vprobe Guest_IRQ
        (printf "Current process during IRQ: %s\n" (curprocname)))
```

Sample output:

```
Current process during IRQ: swapper
Current process during IRQ: ata/0
Current process during IRQ: swapper
Current process during IRQ: swapper
```

# Configuration and Static Probes Reference 3

This chapter includes additional information in the following sections:

## VProbes Versioning

You can configure automated code generation utilities to indicate which version of VProbes you are targeting. Table 3-1 shows what happens with version mismatches.

**Table 3-1.** Version Mismatch Actions

| **If the specified major version is:** | **This happens:** |
|---|---|
| Higher than the current major version | Execution engine refuses to run. |
| Lower than the current major version | Warning is issued about version mismatch, attempts to run. |
| Same but specified minor version is higher | Warning is issued about possible incompatibilities, attempts to run. |
| Same but specified minor version is same or less | Execution engine runs the script normally. |

For example, the following `version` compiler directive specifies VProbes major version 0 and minor version 2:

```
(version "0.2")
```

## VMX Configuration

Table 3-2 lists the VProbes parameters that you can set in the virtual machine's `.vmx` configuration file.

**Table 3-2.** VProbes-Related Configuration Parameters

| **Parameter** | **Description** |
|---|---|
| `vprobe.allow` | Globally allows the site-wide use of VProbes. |
| `vprobe.enable` | Enables VProbes for this virtual machine. |
| `vprobe.vpFile` | Alternate method for specifying a VP source file. |
| `vprobe.outFile` | Specifies a different file for VProbes output. |
| `vprobe.guestSyms` | Specifies the guest symbol file. |
| `vprobe.syscalls` | Specifies file with a list equating `syscall` numbers with names. |

## Supported Static Probes

During probe fire, the name of the executing probe is available in the global string variable PROBENAME.

Table 3-3 lists the static probes that VProbes supports.

**Table 3-3.** Static Probes

| Probe Name | Description | Arguments |
|---|---|---|
| VMM1Hz | Periodic event that happens once a second. | None |
| VMM10Hz | Periodic event that happens ten times a second. | None |
| Guest_[DE, DB, NMI, BP, OF, BR, UD, NM, DF, TS, RESERVED_FAULT, NP, SS, GP, PF, MF, AC, MC, XF] | Guest faults. For instance, Guest_PF is the probe that is run before a page fault is delivered to the guest. | ARG0: Error code, if the fault uses one |
| Guest_CR3Write | Run immediately before guest writes to the CR3 register. | None |
| Guest_IRQ | Run before a hardware interrupt is delivered to the guest. | ARG0: IRQ number |
| Guest_TripleFault | Triple fault event. | None |
| HV_Exit | Exit from hardware-assisted direct execution when using VT or AMD-V. | None |
| HV_Resume | Virtual machine resuming hardware-assisted direct execution when using VT or AMD-V. | None |
| HV_NPF | Run before delivery of a page fault when running with hardware-assisted paging. | None |
| In | Run when the guest executes an In instruction. | ARG0: Port;<br>ARG1: Operand size;<br>ARG2: Rep count |
| Out | Run when the guest executes an Out instruction. | ARG0: Port;<br>ARG1: Operand size;<br>ARG2: Rep count |
| SMM_SMIPre | Run before delivery of an SMI. | None |
| SMM_SMIPost | Run immediately after delivery of an SMI. | None |
| SMM_RSMPre | Run before execution of RSM. | None |
| SMM_RSMPost | Run immediately after execution of RSM. | None |
| Disk_IOStart | Executed at the start of disk I/O. | ARG0: Adapter type;<br>ARG1: Controller ID;<br>ARG2: Drive ID;<br>ARG3: 1 for read, 0 for write;<br>ARG4: Unique I/O identifier;<br>ARG5: Start sector;<br>ARG6: Number of sectors;<br>ARG7: Number of scatter/gather entries |
| Disk_IOFinish | Executed upon completion of disk I/O. | ARG0: Adapter type;<br>ARG1: Controller ID;<br>ARG2: Drive ID;<br>ARG3: 1 for read, 0 for write;<br>ARG4: Unique I/O identifier;<br>ARG5: Start sector;<br>ARG6: Number of sectors;<br>ARG7: Number of scatter/gather entries |
| VMMLoad | Executed in VMM context once when the VP script is loaded. | None |
| VMMUnload | Executed in VMM context once when VP script is unloaded. | None |
| VMXLoad | Executed in VMX context once when the VP script is loaded. | None |
| VMXUnload | Executed in VMX context once when VP script is unloaded. | None |

## Limitations

The VProbes facility imposes a number of limitations:

- The translation cache size is limited.

  VP is translated at runtime and has its object code placed into a fixed-size translation cache. Scripts that do not fit into the cache are not executed. You can increase the size of the cache to the maximum of 64KB by setting `vprobe.maxCodeBytes=65536` in the `.vmx` file.

- The stack size is limited.

  The stack used for VP execution is limited. As such, although recursion is permitted, the depth of recursion is limited. If the stack size limit is reached, a runtime error occurs, and the execution engine terminates the probe at runtime.

- Limited number of probes.

  Each guest virtual machine is limited to 32 probes.

- Aggregates are write-only.

  Although aggregates can have values written to them during execution, a VP script cannot access the values within an aggregate.

- Probe and variable names limited.

  Probe and variable names are limited to 64 bytes, including the terminating null byte.

- Strings have a limited size.

  Strings are limited to 256 bytes, including the terminating null byte.

- The number of arguments is limited.

  The limited stack size places a burden on function arguments. This affects user-defined functions, functions such as `printf` and `scanf`, and aggregates.

- Writing aggregates to the log is expensive.

  Writing aggregates to the log using `logaggr` involves a number of resource-intensive steps, making it a very expensive function. VMware recommends that you not call `logaddr` more than once a second, because it can cause a noticeable performance degradation in the virtual machine.

- Clearing aggregates is expensive.

  Clearing an aggregate using `clearaggr` is an expensive operation. VMware recommends that you not call `clearaggr` more than once a second, because it can cause a noticeable performance degradation in the virtual machine.

# Code Samples

A

This appendix provides the following code samples:

- "Sample Implementation of curprocname and curpid" on page 31
- "Script Example for vptop" on page 32
- "Guest Stack During Page Fault Handling" on page 32

## Sample Implementation of curprocname and curpid

This VP script defines functions for both the current process name and the current process ID.

```
(defstring _procName)
(definteger _pidOffset)
(definteger _nameOffset)

; guestload --
; guestloadstr --
;   Checked wrappers around getguest that return 0 for reads of the null page.
(defun guestload (addr)
  (cond
    ((< addr 4096) 0) ;; Read null for null references
    (1 (getguest addr))))
(defun guestload32 (addr)
  (& 0xffffffff (guestload addr)))
(defun guestloadstr (str addr)
  (cond
    ((< addr 4096) (setstr str "<NULL>"))
    (1 (getgueststr str addr))))
; curthrptr --
;   Return pointer to kernel thread-private data for the current process
;   on the current VCPU. This might be either GSBASE or KERNELGSBASE;
;   testing the CPL isn't *quite* right, because there's a short window
;   immediately after the hardware syscall where the right value is still
;   in KERNELGSBASE.
(defun curthrptr ()
  (cond ((== _pidOffset 0)
         (do
             (setint _pidOffset (offatret "sys_getpid"))
             (setint _nameOffset (offatstrcpy "get_task_comm")))))
  (cond
    ((>= GSBASE 0x100000000) GSBASE)
    (1 KERNELGSBASE)))
(defun curprocname ()
  (guestloadstr _procName (+ _nameOffset (guestload (curthrptr))))
  _procName)
(defun curpid ()
  (guestload32 (+ _pidOffset (guestload (curthrptr)))))
```

## Script Example for vptop

The `vptop` application is an application like `top`, a Linux utility showing the dominant processes on a system. Windows Task Manager is similar if you select the Processes tab. The `vptop` application works by aggregating the current process name and IRQ once per interrupt. In this manner, it is possible to track which applications are running. If operating systems were guaranteed to use the same IRQ for the timer interrupt, then it would be possible for `vptop` be much more accurate, but this VProbes example works well and can track processes running on any guest OS.

```
; vptop.vp
;    Top-like VP script
; running is an aggregate that keeps a count of active process by name
; and which IRQ interrupted the process.
(defaggr running 1 1)
; Guest_IRQ is the probe that does the actual work
(vprobe Guest_IRQ
        (aggr running (ARG0) ((curprocname)) 1))
; Print and clear the aggregate once per second
(vprobe VMM1Hz
        (logaggr running)
        (clearaggr running))
```

## Guest Stack During Page Fault Handling

This script prints the guest stack before each page fault is delivered. You can write a postprocessor that walks the resulting stack dump, equating addresses with symbols that indicate which processes are causing page faults.

```
(definteger stack-pointer)
; Dump the guest stack for a guest that is not in long mode.
(defun dump-stack-32 (level)
  (cond (level
         (do (setint stack-pointer (+ RSP (* (- level 1) 4)))
             (printf "%2d (%08x): %08x\n"
                     (- level 1)
                     stack-pointer
                     (& 0xffffffff (getguest stack-pointer)))
             (print-stack-32 (- level 1))))))
; Dump the guest stack for a guest in long mode.
(defun dump-stack-64 (level)
  (cond (level
         (do (setint stack-pointer (+ RSP (* (- level 1) 8)))
             (printf "%2d (%016x): %016x\n"
                     (- level 1)
                     stack-pointer
                     (getguest stack-pointer))
             (print-stack-64 (- level 1))))))
; Dump the guest stack.
; If bit 8 of the EFER register is set, the guest is in long mode and each entry
; on the stack is 8 bytes. Otherwise each entry is 4 bytes.
(defun dump-stack (level)
  (cond ((& 1 (>> EFER 8))
         (dump-stack-64 level))
        (1
         (dump-stack-32 level))))
(vprobe Guest_PF
   (dump-stack 16))
```

# Emmett – A High-Level VProbes Tool B

The VP scripting language is deliberately limited in its power and flexibility, to reduce overhead and minimize risk to system integrity. You can think of VP as the VProbes "assembly language" rather than as a tool intended for daily use. With that in mind, VMware engineers developed the Emmett language as a sample implementation of a higher level tool encapsulating all the functionality of VProbes.

## About the Emmett Language

Like Perl or Java, Emmett is a high-level C-like language that provides a powerful and intuitive interface to VProbes. It uses a much higher level of abstraction than VP: aggregates follow an intuitive syntax, data types and formats are supported, and code is generally easier to read, write, and comprehend.

Emmett is a small language that provides C-style types, expressions, and conditional operators. Additionally, it has syntactic support for aggregation, and automatic inference of type for undeclared variables.

### Finding the Emmett Toolkit

VMware made Emmett available as open source, licensed with a variant of the flexible BSD legal agreement. You can download the software source code as a Subversion (SVN) repository:

http://sourceforge.net/projects/vprobe-toolkit

For download instructions, click **Code > SVN**. To see source, click **Code > SVN Browse**.

The `vprobe-toolkit` includes the `vprobe` wrapper script that runs the Emmett compiler on your source code, loads the resulting VP script into a target virtual machine, and organizes printed output. For example, the first command displays sorted aggregates, while the second displays formatted stacks:

```
vprobe -a
vprobe -s
```

The `cookbook` subdirectory of the downloaded `vprobe-toolkit` contains code examples.

### Using Emmett with VP Scripts

Suppose you wanted to produce the following VP script:

```
(defaggr a 1 1)
(vprobe USEC:1001
    (do
        (aggr a (VCPUID) ((curprocname)) 1)
        (logaggr a )))
```

The Emmett program that compiles into the above script looks like this:

```
USEC:1001
{
    a[VCPUID, curprocname()]++;
    logaggr(a);
}
```

## Emmett Code Sample

Compare this Emmett code with the VP script in "Sample Implementation of curprocname and curpid" on page 31. This `guestload` code provides the `curprocname` and `curpid` functions for Linux 2.6 kernels:

```
string _procName;
int _pidOffset;
int _nameOffset;
#
# guestload --
# guestloadstr --
#   Checked wrappers around getguest that return 0 for reads of the null page.

int guestload(int addr)
{
   _rval = 0;
   if (addr > 4095) {
      _rval = getguest(addr);
   }
   return _rval;
}

int guestload32(int addr)
{
   return addr & 0xffffffff;
}

void guestloadstr(string dest, int addr)
{
   if (addr > 4095) {
      getgueststr(dest, addr);
   }
}

# curthrptr --
#   Return pointer to kernel thread-private data for the current process on the
#   current VCPU. This might be either GSBASE or KERNELGSBASE; testing the CPL
#   isn't *quite* right, because there's a short window immediately after the
#   hardware syscall where the right value is still in KERNELGSBASE.

int curthrptr()
{
   if (_pidOffset == 0) {
      _pidOffset = offatret("sys_getpid");
      _nameOffset = offatstrcpy("get_task_comm");
   }
   if (GSBASE >= 0x100000000) {
      _gsb = GSBASE;
   } else {
      _gsb = KERNELGSBASE;
   }
   return _gsb;
}

string curprocname()
{
   guestloadstr(_procName, _nameOffset + guestload(curthrptr()));
   return _procName;
}

int curpid()
{
   return guestload32(_pidOffset + guestload(curthrptr()));
}
```