

VProbes Programming Reference

VMware Workstation 8.x

VMware Fusion 4.x

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see <http://www.vmware.com/support/pubs>.

EN-000612-00

vmware[®]

You can find the most up-to-date technical documentation on the VMware 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

Copyright © 2008–2011 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at <http://www.vmware.com/go/patents>.

VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

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

Contents

About This Book	5
1 Introduction to VProbes	7
What VProbes Can Do	7
Advantages of VProbes	7
Enabling and Running VProbes	8
Enabling VProbes	8
Running a VP Script	8
Components of a VP Script	8
Dynamic Probes and Guest Symbols	9
The Emmett Language	9
Finding the Emmett Toolkit	9
Deprecation Notice	9
2 Syntax and Functions Reference	11
VP Syntax	11
Comments	11
Literals	11
Statements	12
User-Defined Variables	12
Integer Variables (scope = thread local)	12
String Variables (scope = thread local)	12
Aggregate and Bag Variables (scope = global)	12
Aggregate Variables (scope = global)	13
Bag Variables (scope = global)	14
Built-in Global Variables	14
Current Probe State	14
Variables for Virtual CPU Registers	15
Variables for Hardware Data	15
Hardware Virtualization State	16
Intel Virtual Machine Control Structure (VMCS)	16
AMD Virtual Machine Control Block (VMCB) State Save Area	16
Probes	17
Static Probes	17
Dynamic Probes	17
Data Probes	18
Periodic Probes	18
Conditional Expressions	18
Do Expressions	19
Functions	19
User-Defined Functions	19
Built-In Operators	19
Built-In Functions	21
logstr	21
logint	21
sprintf	21
printf	22

strcmp	22	
getguest	23	
getgueststr	23	
gueststack	24	
getguestphys	24	
getsystemtime	24	
exit	25	
Offset-At Built-In Functions	25	
offatret	25	
offatseg	25	
offatstrcpy	26	
Non-Built-In Functions	26	
curpid	26	
curprocname	26	
3	Configuration and Static Probes Reference	27
VProbes Versioning	27	
VMX Configuration	27	
Supported Static Probes	27	
Limitations	29	
4	Code Samples	31
Sample Implementation of curprocname and curpid	31	
Script Example for vptop	32	
Guest Stack During Page Fault Handling	32	
Index	33	

About This Book

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

The VProbes facility runs on VMware Workstation and VMware Fusion®.

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
2011-06-13	Review draft for Workstation 8.0 and VMware Fusion 4.0.
2010-05-21	Published version for Workstation 7.1 and VMware Fusion 3.1.
2009-10-26	Published version for Workstation 7.0 and VMware Fusion 3.0.
2009-01-05	Corrected typos in <code>curprocname</code> and <code>curpid</code> example on page 31, republished.
2008-08-15	Added bags to the third draft for delivery with the Workstation 6.5 RC release.
2008-06-20	Second draft of this book for delivery with the Workstation 6.5 Beta 2 release.
2007-12-13	First draft of this Tech Note for the Workstation 6.5 Friends and Family release.

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.

VMware Technical Publications Glossary

VMware Technical Publications provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation go to <http://www.vmware.com/support/pubs>.

Document 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 resources available to you. To access the current versions of other VMware books, go to <http://www.vmware.com/support/pubs>.

Online and Telephone Support

To use online support to submit technical support requests, view your product and contract information, and register your products, go to <http://www.vmware.com/support>.

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>.

Introduction to VProbes

This chapter includes the following topics:

- [“What VProbes Can Do”](#) on page 7
- [“Enabling and Running VProbes”](#) on page 8
- [“The Emmett Language”](#) on page 9

What VProbes Can Do

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. You do not need to recompile or power down virtual machines before investigation. VProbes dynamically inserts scripts into running virtual machines, removes the scripts, then examines the output, repeating this sequence as needed. You can also set static probes at periodic intervals or when important events occur in a virtual machine. Because the VP language has a limited stack size and lacks loop constructs, scripts complete in a finite amount of time, avoiding denial of service impact.



CAUTION VProbes was a supported feature in Workstation 7, but experimental in earlier Workstation releases. VProbes is an experimental feature in VMware Fusion. VProbes is unsupported on ESXi 5.0.

Advantages of VProbes

VProbes had the following design goals:

- **Safe** – VProbes can inspect, record, and compute state of the guest, the virtual machine monitor (VMM), the virtual machine executable (VMX), and the virtual devices, without modifying the current state.
- **Dynamic** – Downtime or recompilation is not necessary for any virtual machine component under investigation. To use VProbes you write a simple script, which is dynamically inserted into and removed from a running virtual machine. After examining the output, you can change the script and repeat.
- **Static** – Predefined probes are available for execution at periodic intervals or when critical events occur.
- **Independent of operating system** – Differences between guest operating systems are made transparent when possible. For example, most operating systems assign names and numeric identifiers to processes. VProbes offers the `curprocname` and `curpid` functions to access this information.
- **Free when disabled** – You can use VProbes in production systems, because when instrumentation is not enabled by a script, VProbes has no cost in memory space or CPU time.

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

Enabling and Running VProbes

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

Enabling VProbes

To enable VProbes on a specific virtual machine, add the following line to its `.vmx` configuration file in the virtual machine directory:

```
vprobe.enable = TRUE
```

Running a VP Script

You use the `vmrun` program to execute VP scripts from the command line. The `vmrun` program is included with Workstation, VMware Fusion, VMware Server, and the VIX API. [Table 1-1](#) shows `vmrun` VP commands.

Table 1-1. VProbes Commands in `vmrun`

Command	Description
<code>vprobeLoad</code>	Load a VP script from the command line.
<code>vprobeLoadFile</code>	Load a VP script from the specified file.
<code>vprobeReset</code>	Disable all running VProbes.
<code>vprobeListProbes</code>	List available probe points.
<code>vprobeListGlobals</code>	List global variables.
<code>vprobeVersion</code>	Display the current VProbes version.

For example, the following command loads the script between single quotes into the running virtual machine identified by its configuration file (`my.vmx`). The probe executes every second and prints the string "hello!" followed by a newline.

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

The `vmrun` utility redirects output from VP scripts to the `vprobe.out` file located in the virtual machine directory, appending each time a script runs. You can watch as this file grows by using the `tail -f` command. If you delete the `vprobe.out` file, `vmrun` will create a new one when it loads the next script.

Because the Windows command prompt (`cmd`) permits nesting of like-type quotes only – either single quotes or double quotes, but not both – the above `vmrun` command produces the error “unknown ident windows” and fails because it contains mixed quotes. To avoid this problem, you can install Cygwin and run VP scripts in a standard bash shell. Alternatively, you can use the `vprobeLoadFile` command to avoid quoting.

The following command loads the VP script from the file `test.vp` and runs it:

```
vmrun vprobeLoadFile my.vmx test.vp
```

Components of a VP Script

A VP script can contain one or more probes. VP scripts are not executed linearly. They are executed as a series of callbacks to individual probes when specific events occur. Each probe specifies an individual event and the instrumentation code to run when the event occurs.

You define probes using the `vprobe` statement. VP scripts use the file extension `.vp`, as shown in section “[Running a VP Script](#)” on page 8. 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 vm/my.vmx
```

You can refer to the same probe, such as `VMM1Hz`, multiple times in a script. All same-probe occurrences will execute at runtime, but there is no guarantee that they will execute in the order specified by the script.

Events intercepted by VProbes fall into two categories:

- **Static probe points** – Predefined virtual hardware events, such as the `VMM1Hz` event, or the delivery of a virtual hardware interrupt to the guest, `GuestIRQ`. For a list, see “Supported Static Probes” on page 27.
- **Dynamic probe points** – These probes run when the guest execution reaches a given address in the guest. Dynamic probes are specified using the `GUEST:` prefix, followed by an address. The example script below prints “reached 0xc0106ae0” if the guest reaches the location specified by that hexadecimal address.

```
(vprobe GUEST:0xc0106ae0
  (printf "reached 0xc0106ae0\n"))
```

Dynamic Probes and Guest Symbols

Guest symbol files provide a mapping from guest symbols to the hexadecimal addresses for those symbols. Use these symbol files to determine what addresses you want to probe.

On Linux guests, you can find guest symbols in the `/proc/kallsyms` file. This file contains exported kernel symbol definitions so that the Linux kernel modules facility can dynamically link and bind loadable modules.

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

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

The Emmett Language

VMware delivers VProbes in conjunction with the `vmrun` utility to support VP scripts. If you prefer a higher level language interface, you can program in Emmett.

Emmett is a C-like language that allows you to program in high-level constructs and compile your code into VP scripts. It is a small language that provides C-style types, expressions, and conditional operators. Emmett has syntactic support for aggregation and automatic inference of type for undeclared variables.

Finding the Emmett Toolkit

Emmett is available as open source, licensed with a variant of the flexible BSD legal agreement. You can download the software source code from the GIT repository at the following location:

<https://github.com/vmware/vprobe-toolkit>

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.

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

Deprecation Notice

The use of guest symbols in VP scripts has been deprecated. VP scripts must refer directly to hexadecimal addresses. If used, the `vprobe.guestSyms` option is ignored and a warning message is printed in the log file of the virtual machine.

Guest symbols are supported in the Emmett compiler, however. Emmett scripts can refer directly to guest symbols. When called with the `-s` option and a guest symbol file, the Emmett compiler automatically translates guest symbols from the Emmett script into their corresponding hexadecimal addresses.

Syntax and Functions Reference

This chapter includes the following topics:

- [“VP Syntax”](#) on page 11
- [“User-Defined Variables”](#) on page 12
- [“Built-in Global Variables”](#) on page 14
- [“Probes”](#) on page 17
- [“Conditional Expressions”](#) on page 18
- [“Do Expressions”](#) on page 19
- [“Functions”](#) on page 19

VP 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 function 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** – Unsigned numeric 64-bit quantities. Integers 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
0xffffffffc0
-65536
+54
```

Statements

Expressions and statements begin with opening parenthesis and end with closing 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, do not use them for user-defined functions or variables.

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

Integer Variables (scope = thread local)

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 (scope = thread local)

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 with `setstr`:

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

Aggregate and Bag Variables (scope = global)

So far, `printf` is the only method introduced to record data from a probe. However, for frequently executed probes, `printf` is unsuitable. 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. In addition, frequent output of data is usually not necessary.

In most commonly executed probes, a small set of recorded values accounts for most 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.

A bag is a temporary scratch location for storing integer key-value pairs.

Aggregate Variables (scope = global)

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 when you use the `logaggr` statement. The `logaggr` statement saves the entire contents of the aggregate to the log file, `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[0xffffffff9800bf8e000] == avg 1 count 1 min 1 max 1 latest 1
  pf[0xffffffff9800408a01c] == avg 1 count 1 min 1 max 1 latest 1
  pf[0xffffffff980044ebded] == avg 1 count 1 min 1 max 1 latest 1
  pf[0xffffffff9800984c410] == avg 1 count 1 min 1 max 1 latest 1
```

This shows that 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 (scope = global)

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 with 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 the 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. 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.

The built-in global variables listed in [Table 2-2](#) and [Table 2-3](#) (but not the current probe state variables listed in [Table 2-1](#)) correspond to the 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*

Current Probe State

[Table 2-1](#) lists the built-in variables for the current probe state.

Table 2-1. Current Probe State

Name	Type	Scope	Description
ARG0-ARG9	Depends on probe	The currently executing probe	Arguments to the current probe. See Table 3-3, "Static Probes," on page 28 for details.
PROBENAME	String	The currently executing probe	Name of the currently executing probe.

Variables for Virtual CPU Registers

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

Table 2-2. Virtual CPU Registers

Name	Type	Scope	Description
RIP	Integer	Per-VCPU	Current 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	Current value of the running virtual CPU's corresponding general purpose register.
RFLAGS	Integer	Per-VCPU	Current value of the running virtual CPU's flags register.
CS, CSAR, CSBASE, CSLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's CS register.
SS, SSAR, SSBASE, SSLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's SS register.
DS, DSAR, DSBASE, DSLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's DS register.
ES, ESAR, ESBASE, ESLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's ES register.
FS, FSAR, FSBASE, FSLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's FS register.
GS, GSAR, GSBASE, GSLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's GS register.
TR, TRAR, TRBASE, TRLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's task register.
LDTR, LDTRAR, LDTRBASE, LDTRLIMIT	Integer	Per-VCPU	Current value, access rights, base, and limit of the virtual CPU's local descriptor table register.
GDTRBASE, GDTRLIMIT	Integer	Per-VCPU	Base and limit of the virtual CPU's current global descriptor table from the GDT register.
IDTRBASE, IDTRLIMIT	Integer	Per-VCPU	Base and limit of the virtual CPU's current interrupt descriptor table from the IDT register.
CR0, CR2, CR3, CR4, CR8	Integer	Per-VCPU	Current value of the running virtual CPU's control registers.
DR6, DR7	Integer	Per-VCPU	Current value of the running virtual CPU's debug registers.
EFER	Integer	Per-VCPU	Current value of the running virtual CPU's EFER registers.
KERNELGSBASE	Integer	Per-VCPU	Current value of the running virtual CPU's KERNELGSBASE MSR.
FSBASE, GSBASE	Integer	Per-VCPU	Current 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 2-3 lists the built-in variables for hardware data.

Table 2-3. Hardware Data

Name	Type	Scope	Description
SMM	Integer	Per-VCPU	1 if the VCPU is running in system management mode, 0 otherwise.
TSC	Integer	Global	Value of the pseudo time stamp counter (pseudo TSC) abstraction. The pseudo TSC behaves like the hardware TSC, except the pseudo TSC is guaranteed to increase monotonically over time and appears the same on all physical CPUs in the system. By contrast, the hardware TSC is sensitive to power-management related changes in clock speed, and each CPU in the system can contain a different hardware TSC value.

Table 2-3. Hardware Data (Continued)

TSC_HZ	Integer	Per-PCPU	Frequency of the TSC on the physical CPU where the probe fires.
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.
BRCNT	Integer	Global	Branch count during logging or replay.

Hardware Virtualization State

When hardware virtualization (HV) is in use, a variety of information about the 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 2-4](#) 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 2-4. Built-in Variables for Intel Processors

Name	Type
VMCS_EXIT_REASON	Integer
VMCS_INSTR_INFO	Integer
VMCS_EXIT_INTR_INFO	Integer
VMCS_EXIT_INTR_ERR	Integer
VMCS_IDTVEC_INFO	Integer
VMCS_IDTVEC_ERR	Integer
VMCS_INSTRLEN	Integer
VMCS_VMENTRY_INTR_INFO	Integer
VMCS_VMENTRY_XCP_ERR	Integer
VMCS_EXIT_QUAL	Integer

AMD Virtual Machine Control Block (VMCB) State Save Area

The variables in [Table 2-5](#) 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 2-5. 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. Each VP script must contain one or more probes. You can insert a probe multiple times. Duplicate probes are executed one after the other.

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

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"))
```

There are four types of probes: 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. [Table 3-3, "Static Probes,"](#) on page 28 contains the argument list, if any, of each static probe. The following sample script uses 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))
```

The script produces the following output:

```
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
```

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))
```

The script produces the following output:

```
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 a guest probe point is `GUEST_WRITE:<guest linear address>`. The following script uses 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"))
```

The script produces the following output:

```
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 `TIMER:<microsecond interval>`. Periodic probes are useful for monitoring changes over time in the state of the guest. 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 TIMER:333333
  (setint tickCount (+ tickCount 333333))
  (printf "%d seconds have elapsed.\n" (/ tickCount 1000000)))
```

The script produces the following output:

```
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

The VP equivalent of `if` and `switch` statements in other languages is `cond`. The `cond` expression takes one or more lists. Each list contains two expressions. The first is treated as a condition. The `cond` expression iterates sequentially through the list pairs, evaluating the condition in the first. If the first condition evaluates to true (nonzero), `cond` evaluates the related expression. Then `cond` skips the remainder of list pairs in the expression, no matter how many subsequent list pairs evaluate to true.

The `cond` expression has the following syntax:

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

The following expression prints "B" to the log because `(== 1 1)` is the first expression that evaluates to TRUE.

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

The following expression does not print anything to the log because nothing evaluates to TRUE:

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

The following expression prints "B" to the log because 1 evaluates to TRUE.

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

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. A function has the following syntax:

```
(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. A user-defined function uses the following syntax:

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

Here are three examples of user-defined functions:

```
; 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)
```

Both the arguments exp1 and exp2 can be an integer literal, an integer variable, or an expression that evaluates to an integer.

The following tables list the various types of built-in operators.

- [Table 2-6, “Arithmetic Operations,”](#) on page 20
- [Table 2-7, “Bit Operations,”](#) on page 20
- [Table 2-8, “Logical Operations,”](#) on page 20
- [Table 2-9, “Comparison Operations,”](#) on page 20

Table 2-6. Arithmetic Operations

Operator	Operation	Example
+	addition	(+ a 23)
-	subtraction	(- 53 0xa)
*	multiplication	(* 6 7)
/	division	(/ 10 3) ; result is 3
%	modulo	(% RAX 2)

Table 2-7. Bit Operations

Operator	Operation	Example
<<	left shift	(<< 1 31)
>>	right shift	(>> CR0 31)
^	exclusive or	(^ var1 var2)
~	bitwise not	(~ 0x5e5e) ; Unary operator
	bitwise or	(pte 0xc00)
&	bitwise and	(& CR0 0x80000000)

Table 2-8. Logical Operations

Operator	Operation	Example
&&	logical and	(&& flag-a flag-b)
	logical or	(flag-a flag-b)
!	logical not	(! now) ; Unary operator

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 to false (zero). The second argument to `||` is not evaluated if the first argument evaluates to true (nonzero).

Table 2-9. Comparison Operations

Operator	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)

Built-In Functions

Built-in functions can be of type `STRING` or `INTEGER`, depending on whether they evaluate to a string or an integer. In the following sections, the function definitions note the return type and the arguments that the function takes. If a function takes different argument lists, multiple argument lists are documented. If the type of a function parameter is specified, the parameter can 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` – The return value indicates success (1) or failure (0).
- Argument: `STRING` – The string to print to the log.

Sample script:

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

Sample output:

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

logint

Print an integer to the log.

- Return type: `INTEGER` – The return value indicates success (1) or failure (0).
- Argument: `INTEGER` – The integer to print to the log.

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
399241216
```

sprintf

Store formatted data in a string variable.

- Return type: `INTEGER` – The return value indicates success (1) or failure (0).
- Arguments:
 - `STRING VARIABLE`: Destination variable.
 - `STRING`: Format string.
 - Zero or more arguments, depending on the contents of the format string.

The `sprintf` function works in the same manner as the C library's `sprintf`.

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 0xffffffff8000189c0f1
Current RIP is 0xffffffff80001ce9133
Current RIP is 0xffffffff80001857670
```

printf

Print formatted data to the log.

- Return type: INTEGER – The return value indicates success (1) or failure (0).
- Arguments:
 - STRING VARIABLE: Destination variable.
 - STRING: Format string.
 - Zero or more arguments, depending on the contents of the format string.

The `printf` function works in the same manner as the C library's `printf`, except that its output is directed to the log instead of to `stdout`.

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. The value returned is zero if and only if the strings are equal.

- 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 – The return value is the 8 bytes of memory as an integer.
- Argument: INTEGER – Linear address to access.

Sample script:

```
; Print the speed of the Time Stamp Counter (TSC) if it is enabled.
; The numeric addresses for "tsc_disabled" and "tsc_khz" must be
; retrieved from a symbol file like /proc/kallsyms. These symbols
; are available for 32-bit Ubuntu 7.04. Look for similar symbols
; in other Linux guests.

(definteger tsc_disabled_addr 0xc042d2b8)
(definteger tsc_khz_addr      0xc042d2bc)
(definteger tsc_disabled)
(definteger tsc_khz)

(vprobe VMM1Hz
  (setint tsc_disabled (& 0xffffffff (getguest tsc_disabled_addr)))
  (cond (tsc_disabled (printf "TSC is disabled\n"))
        (1 (do (setint tsc_khz (& 0xffffffff (getguest tsc_khz_addr)))
                (printf "TSC kHz: %d\n" tsc_khz))))))
```

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. The length of the resulting string is limited to 255 bytes. If the address is invalid (that is, not currently mapped in), the current probe is cancelled.

- Return type: INTEGER
- Arguments:
 - STRING VARIABLE: Destination string.
 - INTEGER: Number of bytes to copy.
 - INTEGER: Starting linear address.

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

Sample script:

```
; Print the saved Linux boot command line.
; The numeric address for "saved_command_line" must be retrieved
; from a symbol file like /proc/kallsyms. This symbol is available
; for 32-bit Ubuntu 7.04. Look for a similar symbol in other Linux guests.

(definteger saved_command_line_addr 0xc042b020)
(defstring command_line_str)

(vprobe VMM1Hz
  (getgueststr command_line_str 255 saved_command_line_addr)
  (printf "Linux command line (at %#x):\n%s\n"
         saved_command_line_addr command_line_str))
```

Sample output:

```
Linux command line (at 0xc042b020):
root=UUID=64123f18-e6fd-4b7b-ae63-d1b995cd4046 ro 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 – The return value indicates success (1) or failure (0).
- Argument: STRING VARIABLE – Destination for stack information.

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_0xfec97b
    fe_0xfec86bed_0xfec863f8_0xfec834c3_0fee6f1ff_0x809f258_0x809f6b1_0x809895d_0x816
    52ed_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
- Argument: 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: None

Sample script:

```
; Approximate the TSC frequency.
(definteger lastTsc 0)
(definteger tsc 0)
(definteger lastSystemTime 0)
(definteger systemTime 0)
(definteger tsc Freq 0)
(defun readTime ()
 (setint lastSystemTime systemTime)
 (setint lastTsc tsc)
 (setint systemTime (getsystemtime))
 (setint tsc TSC))
(defun calculateFreq (tscDiff systemTimeDiff)
 (cond ((!= 0 systemTimeDiff)
 (/ (* 1000000 tscDiff) systemTimeDiff))
 (1
 0)))
```

```
(vprobe VMX1Hz
  (readTime)
  (cond ((!= 0 last)
    (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)
```

exit

End the current probe and immediately return control to the caller of the VP script.

- Return type: INTEGER
- Arguments: None

Sample script:

```
(definteger i)
(vprobe VMM1Hz
  (setint i (+ i 1))
  (printf "v-%d i: %d\n" VCPUID i)
  (cond
    ((>= i 4)
      (do
        (printf "v-%d i: %d exiting\n" VCPUID i)
        (exit))))))
```

Sample output:

```
v-0 i: 1
v-0 i: 2
v-0 i: 3
v-0 i: 4
v-0 i: 4 exiting
```

Offset-At Built-In Functions

Three functions provide support for searching several code patterns. These may be deprecated in the future.

offatret

Search for the first return instruction from a given linear address, and return the value returned by that instruction.

- Return type: INTEGER
- Argument: INTEGER – Starting linear address.

offatseg

Search for the first FS or GS prefixed instruction from a given linear address, and return the displacement or immediate argument of that instruction.

- Return type: INTEGER
- Argument: INTEGER – Starting linear address.

offatstrcpy

Search for the first call to `strcpy` from a given linear address, and return the value of the RSI register at that instruction.

- Return type: INTEGER
- Arguments:
 - INTEGER: Starting linear address.
 - INTEGER: Address of `strcpy`.

Non-Built-In Functions

Several functions are not built into the language but can be implemented as user-defined, guest-specific functions. See [“Sample Implementation of `curprocname` and `curpid`”](#) on page 31 for an example of how to implement `curprocname` and `curpid` for 64-bit Linux 2.6 kernels.

curpid

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

- Return type: INTEGER – The return value is the current process ID.
- Arguments: None

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 the following topics:

- [“VProbes Versioning”](#) on page 27
- [“VMX Configuration”](#) on page 27
- [“Supported Static Probes”](#) on page 27
- [“Limitations”](#) on page 29

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	Issues warning about version mismatch, and attempts to run.
Same, but specified minor version is higher	Issues warning about possible incompatibilities, and 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 1 and minor version 0:
(`version "1.0"`)

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
<code>vpobe.enable</code>	Enables VProbes for this virtual machine.
<code>vpobe.vpFile</code>	Alternative method for specifying a VP source file.
<code>vpobe.outFile</code>	Specifies a different file for VProbes output.

Supported Static Probes

During probe fire, the name of the executing probe is available in the global string variable `PROBENAME`. [Table 3-3, “Static Probes,”](#) on page 28 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 10 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 CR3 register.	None
Guest_IRQ	Run before a hardware interrupt is delivered to 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; ARG1: Operand size; ARG2: Rep count
Out	Run when the guest executes an Out instruction.	ARG0: Port; ARG1: Operand size; 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_IOSStart	Run at the start of disk I/O.	ARG0: Adapter type; ARG1: Controller ID; ARG2: Drive ID; ARG3: 1 for read, 0 for write; ARG4: Unique I/O identifier; ARG5: Start sector; ARG6: Number of sectors; ARG7: Number of scatter/gather entries
Disk_IOFinish	Run upon completion of disk I/O.	ARG0: Adapter type; ARG1: Controller ID; ARG2: Drive ID; ARG3: 1 for read, 0 for write; ARG4: Unique I/O identifier; ARG5: Start sector; ARG6: Number of sectors; ARG7: Number of scatter/gather entries
MAC_SendPacket	Run when sending a network packet.	ARG0: Network adapter
MAC_ReceivePacket	Run when receiving a network packet.	ARG0: Network adapter
VMMLoad	Run in VMM context once when a VP script is loaded.	None
VMMUnLoad	Run in VMM context once when a VP script is unloaded.	None
VMXLoad	Run in VMX context once when a VP script is loaded.	None
VMXUnLoad	Run in VMX context once when a VP script is unloaded.	None

Limitations

The VProbes facility imposes the following 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, which limits the depth of recursion. 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, built-in functions such as `printf` and `scanf`, and aggregates.

- Writing aggregates to the log is expensive.

Writing aggregates to the log using `logaggr` involves some resource-intensive steps, making it a very expensive function. VMware recommends that you not call `logaggr` 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.

- Some static probes might fire outside of the virtual CPU (VCPUs) context.

Due to the architecture of VMware software, it is possible for some probes, such as `MAC_SendPacket` and `MAC_ReceivePacket`, to fire in a non-VCPU context. If this happens and the running script references VCPU state, a runtime error message appears complaining about lack of access to the VCPU state on “non-vcpu vmx threads.”

Code Samples

This chapter includes 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

The following VP script defines functions for both the current process name and the current process ID, for 64-bit Linux 2.6 kernels. The addresses of the `sys_getpid`, `get_task_comm`, and `strncpy` symbols must be extracted from a symbol file such as `/proc/kallsyms`.

```
(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 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 is 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 0xffffffff80096ea9)) ; address of "sys_getpid"
      (setint _nameOffset (offatstrncpy 0xffffffff800537a8 ; address of "get_task_comm"
        0xffffffff80052aa1)))) ; address of "strncpy"

    (cond
      ((>= GSBASE 0x100000000) GSBASE)
      (1 KERNELGSBASE)))
  (defun curprocname ()
    (guestloadstr _procName (+ _nameOffset (guestload (curthrptr))))
    _procName)
  (defun curpid ()
    (guestload (+ _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 the 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, 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

The following 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, to determine 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))
```

Index

A

addition **20**
advantages of VProbes **7**
aggregate variables in VP scripts **13, 14**
AMD Virtual Machine Control Block **16**

B

bitwise and **20**
bitwise not **20**
bitwise or **20**
built-in functions **21**
built-in global variables **14, 15**
 current probe state **14**
 hardware data **15**
 hardware virtualization **16**
built-in operators **19**

C

comments in VP scripts **11**
components of a VP script **8**
conditional expressions, cond **18**
curpid sample implementation **26, 31**
curprocname sample implementation **26, 31**

D

data probes **18**
design goals of VProbes **7**
Disk_IOFinish static probe **28**
Disk_IOStart static probe **28**
division **20**
do expressions **19**
dynamic probes **17**
dynamic probes and guest symbols **9**

E

Emmett high-level language **9**
equal to **20**
exclusive or **20**
exit built-in function **25**
experimental status of VProbes **7**

F

functions, user-defined and built-in **19**
Fusion and VProbes **7**

G

getguest built-in function **23**

getguestphys built-in function **24**
getgueststr built-in function **23**
getsystemtime built-in function **24**
GitHub site with Emmett toolkit **9**
greater than **20**
greater than or equal to **20**
guest stack, print during page fault **32**
guest symbols and dynamic probes **9**
Guest_* static probe **28**
Guest_CR3Write static probe **28**
Guest_IRQ static probe **28**
Guest_TripleFault static probe **28**
gueststack built-in function **24**

H

HV_Exit static probe **28**
HV_NPF static probe **28**
HV_Resume static probe **28**

I

In static probe **28**
integer literals in VP scripts **11**
integer variables in VP scripts **12**
Intel Virtual Machine Control Structure **16**

L

left shift **20**
less than **20**
less than or equal to **20**
limitations of VProbes facility **29**
loading a VP script **8**
logical and **20**
logical not **20**
logical or **20**
logint built-in function **21**
logstr built-in function **21**

M

MAC_ReceivePacket static probe **28**
MAC_SendPacket static probe **28**
modulo **20**
multiplication **20**

N

not equal to **20**

O

offatret built-in function **25**
 offatseg built-in function **25**
 offatstrcpy built-in function **26**
 Out static probe **28**

P

page fault printing guest stack **32**
 periodic probes **18**
 printf built-in function **22**
 probes, static and dynamic **17**

R

right shift **20**
 running a VP script **8**

S

SMM_RSMPPost static probe **28**
 SMM_RSMPPre static probe **28**
 SMM_SMIPPost static probe **28**
 SMM_SMIPPre static probe **28**
 sprintf built-in function **21**
 static probes **17**
 static probes in PROBENAME **27**
 strcmp built-in function **22**
 string literals in VP scripts **11**
 string variables in VP scripts **12**
 subtraction **20**
 syntax of VP scripts **11**

T

technical support resources **6**
 TSC_HZ static probe **17**

U

user-defined functions **19**
 user-defined variables **12**

V

variables in VP scripts **12**
 versioning of VProbes **27**
 VIX API and vmrun **8**
 VMM10Hz static probe **21, 22, 28**
 VMM1Hz static probe **8, 13, 17, 28**
 VMMLoad static probe **28**
 VMMUnload static probe **28**
 vmrun utility **8**

- vprobeListGlobals command **8**
- vprobeListProbes command **8**
- vprobeLoad command **8**
- vprobeReset command **8**
- vprobeVersion command **8**

 VMware Fusion and VProbes **7**

VMX configuration file **27**

- vprobe.enable **27**
- vprobe.outFile **27**
- vprobe.vpFile **27**

VMXLoad static probe **28**

VMXUnload static probe **28**

vprobe-toolkit with Emmett language **9**

vptop sample script **32**

W

Workstation and VProbes **7**