ProbeVue QuickSheet

Version: 1.0.0 – [6.1 TL7, 7.1 TL1]
Date: 3/2/12

Note: The majority of the contents of this document are based on the original 6.1 release. Some items may not have been functional until 6.1 TL4 or TL6 (7.1). At least one item was introduced in TL7 (7.1 TL1).

Variable Data Types
- The data types available within the Vue language are generally the same as those within the C language.
- Vue also supports a string, list, associative array, timestamp, and stacktrace data types.
- float and double data types are supported for capture only. Floating point math is not supported within the ProbeVue environment.

Lists
- Lists are always global, and therefore must be declared in global scope.
- List must be initialized with the list() function only in @@BEGIN
- No truncate, or re-initialize function exists for a list. Use a total, min, max, and count variable to manually replicate with a reset option.
- Lists are abstract data types consisting of (returning) long longs.

Associative Arrays
- Auto typed, consist of primitive data types.
- Printed with print(), quantize(), or quantizate(). Reset with clear().
- Keys can be strings or numeric types.

Strings
- The String data type cannot be declared thread local.
- Strings can be concatenated using the << or + operators. mystring = "a" + "b";
- mystring += "c";
- Strings are declared using the following syntax: String mystring[length];

Variable Classes
- The "class" of a variable generally refers to its scope and its provider.
- Not all classes are available from every section of a Vue script.
- Variables can be explicitly declared as global or thread local.
- Variables declared in an action block are local to that block.
- Exit and Entry variables are relevant only in function probes.

Global
- Declared global and available only within Vue script
- Thread Local
- Local to the probed thread but global to the Vue script
- Declared within and local to the action block
- Exit
- Provided by syscall(x) / uft entry probes, local to action block
- Entry
- Provided by syscall(x) / uft entry probes, local to action block
- Kernel
- Provided externally, global to Vue script
- Built-In
- Provided externally, global to Vue script

Built-in Variables
The following variables are available in the predicate or probe action block are the relevant values for the process firing the probe.

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_tid</td>
<td>Thread ID</td>
</tr>
<tr>
<td>_pid</td>
<td>Process ID</td>
</tr>
<tr>
<td>_ppid</td>
<td>Parent Process ID</td>
</tr>
<tr>
<td>_pgid</td>
<td>Process Group ID</td>
</tr>
<tr>
<td>_phame</td>
<td>(String) Process Name</td>
</tr>
<tr>
<td>_uid</td>
<td>User ID</td>
</tr>
<tr>
<td>_euid</td>
<td>Effective User ID</td>
</tr>
<tr>
<td>_rcid</td>
<td>PID of the probe environment</td>
</tr>
<tr>
<td>_error</td>
<td>Current errno value (exit probes)</td>
</tr>
<tr>
<td>_kernelemode</td>
<td>(Boolean) Process in Kernel-mode</td>
</tr>
<tr>
<td>_argv</td>
<td>Xth arg to probed function where X &gt; 1 (entry)</td>
</tr>
<tr>
<td>_rv</td>
<td>returned value of probed function (exit)</td>
</tr>
</tbody>
</table>
```

Shell Variables
- Exported environmental variables are available within a Vue script much like they are in a shell script.
- The script command line positional parameters are $1 ... $n
- The $CPID variable is available when using probevue -X <command>
- Parameters must be passed wrapped in " to be recognized as a string

Other Variable Types
- syscall_x_t = result of get_syscall(name)
- probevue_timestamp_t = result of timestamp()

Declaring Thread, Global, & Kernel Variables
- Variables declared at the top of a Vue script are global.
- Specifically declare a variable global using:_global int myglobal; (Explicitly declared)
- global:myglobal = 0; (Implicitly declared on first use)
- Thread variables are declared using:_thread or thread:
- Kernel variables are declared using _global

Predicates
- Predicates are optional filtering clauses for probes definitions
- For example, to limit read probes to only stdin for a single PID: @syscall::read:entry
- When (((_pid == $PID) && (_arg1 == 0))

Probe Types & Formats
- ProbeVue has five general probe classes. They are:
  1. Conventional trace probes (systrace)
  2. User Function Entry probes (uf,ufjava,utfxc++)
  3. System Call Entry/Exit probes (syscall,syscallx)
  4. Probes that fire at specific time intervals (interval)
  5. Conventional trace probes (systrace)

Syscall probe:
- @syscall::<pid>::syscall_name::<entry | exit>
- The <pid> is optional and can be globbed with a *
- Syscallx probe:
- @syscallx::<pid>::syscall_name::<entry | exit>
- A value of 1000 fires every second
- User Function probe:
- @uf::<pid>::<func_name>::entry
- The <pid> and <func_name> sections are required (no globs)
- The third section is reserved and must be a *
- ProbeVue probe:
- @@BEGIN and @@END

Concepts
- Basic data types from other processes can be accessed directly while structured data such as a struct or an array must be copied into the probevue environment with get_userstring() or copy_userdata() before it can be used within the probevue process environment.
- If a data is paged out, probevue cannot cause a page fault to bring the data back in. probevue will return 0 / NULL for this data.
- Looping and complex flow is not supported in Vue, but if-then-else conditional flow control is supported. Additionally you can return(); (prematurely) from an action block (but not return a value).
- While action blocks act internally like C functions in terms of scoping and syntax, they have no parameters or return values. Data from the C function is available using the _argv, _rv, and other "_" variables. These variables are globally available but are relevant to the firing probe. For this reason many of the built-in variables "_" have no relevance to interval probes that do not fire in a PID context.
Functions

Printing

void printf(format, ...)  
  → Works like the C stdio version of printf()
void trace(data)  
  → Dumps data in hex to the trace buffer (output)
stktrace(flags, depth)  
  → Dumps a stack trace of depth levels. Flags:
    PRINT_SYMBOLS = Use symbolic names
    GET_USER_TRACE = Show user-mode stack
    ptree(int depth)  
  → Print an ASCII-art tree of processes
print_args()  
  → Print function name and arguments to that function

Associative Array Printing

void printv(myArray)  
  → Simply dumps array data
void quantize(myArray)  
  → Prints array data with relative "bars"
void iquantize(myArray)  
  → Prints array data with adjusted relative "bars"

Probe point information

String get_probe_point(void)  
  → Get the name of the firing probe
get_function(void)  
  → Get the name of the firing function (minus "()")
int get_location_point(void)  
  → Returns either FUNCTION_ENTRY or FUNCTION_EXIT

Tentative Tracing

• Tentative tracing allows trace data to be captured and selectively used. All tentave tracing sessions are keyed with a string that is the single parameter to each of these functions.
void start_tentative(String)  
void end_tentative(String)  
void commit_tentative(String)  
void discard_tentative(String)

Other

int strlen(String)  
  → Get the length of a string (TL3 and later).
int sizeof(type) = May be unreliable on some types (like _argv)
  → Get the size of a data type
String get_user_string(pointer, length)  
  → Copy a string from userspace. Set length to -1 to copy to EOL.
stktrace_t getstktrace(depth) = print((...) to print
  → Return a stktrace_t item with depth levels
proverb_t Timestamp_t timestamp(void)  
  → Get a high resolution time stamp
long long diff_time(start_ts, end_ts, format_flag)  
  → Compare two time stamps (from timestamp() function)

void probe_t Gedmother is either MILLIESCONDS or MICROSECONDS
void exit(void)  
  → Exit the process session.
void return(void)  
  → Exit the action block.
int atoi(String)  
  → Converts a string representation of a number to an int
String strstr(String, String)  
  → Return a new String containing first instance of S2 in S1
void copy_userdata(_argx, destinaion)  
  → Copies probed userland memory structure to probevue memory

Builtin Structures

• All values here are long long excpet cwd (cwd is of type String)
current_thread {  
  tid = Thread ID
  pid = Process ID
  policy = Scheduling policy
  pri = Priority
cpuusage = CPU usage
cpuid = Processor to which the current thread is bound to
signmask = Signal blocked on the thread
lockcount = Number of kernel locks taken by the thread
...

currproc {  
  pid = Process ID
  ppid = Parent process ID
  pgid = Process group ID
  uid = Real user ID
  euid = Saved user ID
  pri = Priority
  nice = Nice value
  cpu = Processor usage
  adspace = Process address space
  majfil = Major file of process
  minfil = Minor file of process
  size = Size of image in bytes
  sigpend = Signals pending on the process
  sigignore = Signals ignored by the process
  sigcatch = Signals being caught by the process
  forktime = Creation time of the process
  threadcount = Number of threads in the process
  cwd } = CWD of process (7.1 TL1/6.1 TL7)
...

---

About this QuickSheet

Created by: William Favorite (wffavorite@tablespace.net)

Copyright: This document is a guide and it includes no express warrants to the suitability, relevance, or compatibility of its contents with any specific system. Research any and all commands that you inflict upon your command line.

Distribution: The PDF version of this guide is free to redistribute as long as credit to the author and tablespace.net is retained in the printed and viewable versions.