# 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 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).

## Vue Structure

The following is an example of a Vue script. The interpreter "magic" allows this script to be set executable and called directly from the command line. This script will run for 10 seconds and count every successful read() system call that happens on the system during that time.

```
#!/bin/probevue                          Interpreter String

/* Successful read() counter */          Comment

int read(int, char *, int);              C Function Prototype

int x;                                   Variable Declaration

@@BEGIN                                  Probe Point
{
  x = 0;                                 Action Block
}

@@syscall:*:read:exit                    Probe Point
when(__rv != -1)                         Predicate
{
  x++;                                   Action Block
}

@@interval:*:clock:10000                 Probe Point
{
  printf("%d reads.", x);                Action Block
  exit();
}
```

## 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 data is paged out, probevue cannot cause a pagefault 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 probed function is available using the __arg*n*, __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.

## 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 lquantize(). 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

**Automatic** (action block local)
Declared within and local to the action block

**Exit** (__rv)
Provided by syscall(x) / uft exit probes, local to action block

**Entry** (__arg*X*) ⇐ Where $X \geq 1$
Provided by syscall(x) / uft entry probes, local to action block

**Kernel**
Provided externally, global to Vue script

**Built-In** (__pid, __pname, __uid, etc...)
Provided externally, values dependent upon probe type

**Shell** ($1, $PATH, etc...)
Provided externally, global to Vue script

## Built-in Variables

The following variables are availible in the predicate or probe action block are are the relevant values for the process firing the probe.

| | |
|---|---|
| __tid | Thread ID |
| __pid | Process ID |
| __ppid | Parent Process ID |
| __pgid | Process Group ID |
| __pname | (String) Process Name |
| __uid | User ID |
| __euid | Effective User ID |
| __trcid | PID of the probevue environment |
| __errno | Current errno value (exit probes) |
| __kernelmode | (Boolean) Process in Kernel-mode |
| __arg*X* | *X*th arg to probed function where $X \geq 1$ (entry) |
| __rv | returned value of probed function (exit) |

## Shell Variables

- Exported environmental variables are available within a Vue script much like they are in a shell script.
- The script command line positional paramaters are $1 ... $*n*
- The `$__CPID` variable is availible when using `probevue -X <command>`
- Parameters must be passed wrapped in \" to be recognized as a string
  `./myvuescript.e \"string\"`

## Other Variable Types

- `stktrace_t` ⇐ result of `get_stktrace(level)`
- probev_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 `__kernel`
  `__kernel long time` ⇐ Seconds since epoch

## 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. probevue probes that fire at BEGIN and END of Vue session
  2. User Function Entry probes (uft,uftjava,uftxlc++)
  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>`
  ○ Second, third, or last tuples can be globbed
- Interval probe:
  `@@interval:*:clock:`*X*`00`
  ○ Second tuple (optionally) specifies a PID context for the interval
  ↪ PID intervals only fire when the PID is on a CPU at the interval
  ○ The final section is in milliseconds and must be divisible by 100
  ↪ This can be tuned to 10 milliseconds with probevctrl
  ○ A value of 1000 fires every second
- User Function probe:
  `@@uft:<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`

## `probevue` Command-Line Options

- Most command line options will not be processed properly if they are passed as an option to the interpreter in the `#!/bin/probevue` magic.

| | |
|---|---|
| `-I <FILE>` | Use FILE as include file |
| `-o <FILE>` | Use FILE as output destination |
| `-X <PROG>` | Start PROG as watched process (`$__CPID`) |
| `-A <ARGS>` | Arguments to -X PROG |
| `-K` | Enable RAS functions |
| `-c timestamp=0` | Timestamp all output |

- If a script name is specified, it and its arguments should be the final arguments to `probevue`

```
probevue -I header.i script.e scriptparam1 scriptparam2
```

## @@uft probes

- uft supports entry and exit probes
- The PID must be specified in the probe description

## @@syscall probes

- Not all syscalls have probe definitions (in syscall provider).
- The function name portion of the probe definition cannot be globbed
- The function *must* be prototyped if __argX or __rv are to be accessed.

| | | | |
|---|---|---|---|
| absinterval | accept | bind | close |
| creat | execve | exit | fork |
| getgidx | getgroups | getinterval | getpeername |
| getpid | getppid | getpri | getpriority |
| getsockname | getsockopt | getuidx | incinterval |
| kill | listen | lseek | mknod |
| mmap | mq_close | mq_getattr | mq_notify |
| mq_open | mq_receive | mq_send | mq_setattr |
| mq_unlink | msgctl | msgget | msgrcv |
| msgsnd | nsleep | open | pause |
| pipe | plock | poll | read |
| reboot | recv | recvfrom | recvmsg |
| select | sem_close | sem_destroy | sem_getvalue |
| sem_init | sem_open | sem_post | sem_unlink |
| sem_wait | semctl | semget | semop |
| semtimedop | send | sendmsg | sendto |
| setpri | setpriority | setsockopt | setuidx |
| shmat | shmctl | shmdt | shmget |
| shutdown | sigaction | sigpending | sigprocmask |
| sigsuspend | socket | socketpair | stat |
| waitpid | write | | |

## @@syscallx (Extended syscall) probes

- syscallx is similar to syscall except it is not limited to a small list
- The extended syscall provider allows for globbing of the syscall name
- Functions must be forward declared / prototyped for argument access

## Vue Snippets

**Multiple probe definitions on one line**

```
@@syscall:*:read:entry, @@syscall:*:write:entry
```

**Printing time elapsed**

```
totalt = diff_time(begint, endt, MILLISECONDS) / 1000;
printf("Time elapsed:  %ld h %ld m %ld s\n",
        totalt / 3600,
        (totalt % 3600) / 60,
        totalt % 60);
```

**Have probevue exit when watched PID exits**

```
@@syscall:$1:exit:entry
{
   exit();
}
```

**Calculating a "floating point" percentage of a integer number**

```
printf("%lu.%0.2lu%%",
       (intn * 100) / intd,
       ((intn * 10000) / intd) % 100);
```

**Capturing count of each syscall for a specific PID**

```
@@syscallx:$1:*:entry
{
   syscount[get_function()]++;
}
```

**Print top-level function for PID every 1/10th sec** (when on CPU)

```
@@interval:$1:clock:100
{
   stktrace(PRINT_SYMBOLS | GET_USER_TRACE, 1);
}
```

## 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 symbol 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 print(myArray)
↪ Simply dumps array data
void quantize(myArray)
↪ Prints array data with relative "bars"
void lquantize(myArray)
↪ Prints array data with adjusted relative "bars"

### Lists

List list(void) ⇐ Initialize a list.(@@BEGIN)
void append(List, long long) ⇐ Append an item

- The List data type utilizes a number of aggregation functions.

sum(List) avg(List) count(List) min(List) max(List)

### Probe point information

String get_probe(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

### Tenative Tracing

- Tentative tracing allows trace data to be captured and selectively used. All tentative 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 __argX)
↪ Get the size of a data type
String get_userstring(pointer, length)
↪ Copy a string from userspace. Set length to -1 to copy to EOL.
stktrace_t get_stktrace(depth) ⇐ printf(%t...) to print
↪ Return a stktrace_t item with depth levels
probev_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). format_flag is either MILLISECONDS or MICROSECONDS
void exit(void)
↪ Exit the probevue session.
void return(void)
↪ Exit the action block.
int atoi(String)
↪ Converts a string representation of a number to an int
String strstr(String_1, String_2)
↪ Return a *new* String containing first instance of S2 in S1
void copy_userdata(__argX, destination)
↪ Copies probed userland memory structure to probevue memory

## Builtin Structs

- All values here are long long except cwd (cwd is of type String)

__curthread {
tid ⇐ Thread ID
pid ⇐ Process ID
policy ⇐ Scheduling policy
pri ⇐ Priority
cpuusage ⇐ CPU usage
cpuid ⇐ Processor to which the current thread is bound to
sigmask ⇐ Signal blocked on the thread
lockcount } ⇐ Number of kernel lock taken by the thread
__curproc {
pid ⇐ Process ID
ppid ⇐ Parent process ID
pgid ⇐ Process group ID
uid ⇐ Real user ID
suid ⇐ Saved user ID
pri ⇐ Priority
nice ⇐ Nice value
cpu ⇐ Processor usage
adspace ⇐ Process address space
majflt ⇐ I/O page fault
minflt ⇐ Non I/O page fault
size ⇐ Size of image in pages
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)
__ublock {
text ⇐ Start of text
tsize ⇐ Text size (bytes)
data ⇐ Start of data
sdata ⇐ Current data size (bytes)
mdata ⇐ Maximum data size (bytes)
stack ⇐ Start of stack
stkmax ⇐ Stack max (bytes)
euid ⇐ Effective user ID
uid ⇐ Real user ID
egid ⇐ Effective group ID
gid ⇐ Real group ID
utime_sec ⇐ Process user resource usage time in seconds
stime_sec ⇐ Process system resource usage time in seconds
maxfd } ⇐ Max fd value in user

## Headers

- typedefed types are not valid (unless the typedef is included).
- Headers have .i extensions by convention. It is not safe to assume that .h files will parse correctly.
- Header files can be included using one of the following methods:

```
probevue -I header1.i -I header2.i yourscript.e
  - or -
probevue -I header1.i,header2.i yourscript.e
```

- There is no #include or #pragma option from *within* a script

## About this QuickSheet

**Created by:** William Favorite (wfavorite@tablespace.net)
**Updates at:** http://www.tablespace.net
**Disclaimer:** This document is a guide and it includes no express warranties 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 is free to redistribute as long as credit to the author and tablespace.net is retained in the printed and viewable versions. LaTeXsource not distributed at this time.