

# ProbeVue QuickSheet

Version: 0.9.1  
Date: 3/11/9

Note: The contents of this document are based on AIX 6.1 TL2

## Overview

ProbeVue is a dynamic tracing environment introduced in AIX 6.1 that provides lightweight tracing of running applications. It is designed to have minimal impact on system performance and requires neither modifications to the kernel or applications being probed.

ProbeVue scripts are written as a series of instructions that should be processed whenever a probe point fires on the system. Probe points can be defined for entry or exit from a system call, regular intervals, or entry into user functions. The instructions for each probe event closely parallel the C programming language in both syntax and capabilities.

A Vue script is compiled at runtime by the probevue compiler and runs as a privileged user-land process. Because probevue runs in its own environment it does not have *direct* access to other processes such as the ability to modify data, but can *view* the results of events and read data from another process space.

## 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.

<pre>#!/bin/probevue</pre>	Interpreter String
<pre>/* Successful read() counter */</pre>	Comment
<pre>int read(int, char *, int size);</pre>	Function Prototype
<pre>int x;</pre>	Variable Declaration
<pre>@@BEGIN</pre>	Probe Point
<pre>{   x = 0; }</pre>	Action Block
<pre>@@syscall:*:read:exit</pre>	Probe Point
<pre>when(!_rv != -1)</pre>	Predicate
<pre>{   x++; }</pre>	Action Block
<pre>@@interval*:clock:10000</pre>	Probe Point
<pre>{   printf("%d reads.", x);   exit(); }</pre>	Action Block

## 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 (prematurely) return(); 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 `__argn`, `__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.

## 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 and a List data type. Both types are effectively arrays. A String is an array of chars, and a List is an array of long longs (although it is considered an "abstract" data type).
- Float data types (including doubles) are supported for capture only. Floating point math is not supported within the ProbeVue environment.
- Vue is highly sensitive to variable types. Mis-matching types and relying on casts or automatic type promotion can lead to unexpected results.

## Lists

- Lists are always global, and therefore must be declared in global scope.
- List must be initialized with the `list()` function. This function can only be called from a @@BEGIN action block.
- Currently no truncate, or re-initialize function exists for a list.
- Lists are considered abstract data types but are actually long longs.

## 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 exit probes, local to action block

### Entry ( \_\_argX )

Provided by syscall / 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 available in the predicate or probe action block 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
__kernelmode	(Boolean) Process in Kernel-mode
__argX	Xth argument to probed function
__rv	returned value of probed function

- \_\_argX variables are only available to entry probes. \_\_rv and \_\_errno are available only in syscall:::exit probes

## Shell Variables

- Exported environmental variables are available within a probevue 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  
./myvuecript.e \"string\"

## 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`

## 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

- ProbeVue has four different probe classes. They are:
  - User Function Entry probes (or uft probes)
  - System Call Entry/Exit probes (or syscall probes)
  - Probes that fire at specific time intervals (or interval probes)
  - probevue probes that fire at BEGIN and END of Vue session
- Probe specifications have the format:

```
@@<Probe_Class>:<Colon_Separated_Probe_Specifier>  
<Probe_Class> := syscall | uft | interval  
<Colon_Separated_Probe_Specifier> := (See following)
```
- Syscall probe:

```
@@syscall:<pid>:<syscall_name>:<entry | exit>
```

  - The <pid> is optional and can be globbed with a \*
- Interval probe:

```
@@interval*:clock:X00
```

  - The second section is reserved and *must* be a \*
  - The final section is in milliseconds and must be divisible by 100
  - 1000 milliseconds is 1 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
```

  - These do not have additional colon-separated specifications

## @@interval probes

- The final number of the probe is milliseconds between firings
- The following probe will fire every second

```
@@interval:*:clock:1000
```
- The interval *must* be evenly divisible by 100
- Many variables or functions are not relevant within interval probes.

## @@uft probes

- uft only supports the entry probe (not exit)
- The PID must be specified in the probe description

## @@syscall probes

- Not all syscalls have probe definitions.
- 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
sentimedop	send	sendmsg	sendto
setpri	setpriority	setsockopt	setuidx
shmat	shmctl	shmdt	shmget
shutdown	sigaction	sigpending	sigprocmask
sigsuspend	socket	socketpair	stat
waitpid	write		

## 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);
```

### Running average (of bytes read)

```
@@syscall:$1:read:exit
when ( __rv > 0 )
{
    count = count + 1;
    avg = avg + ( ( __rv - avg ) / count );
}
```

### 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);
```

## 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()
    - Dumps a stack trace
```

### Lists

- The List data type comes with a number of aggregation functions. to process data in the list. The names are self evident.

```
List list(void)
    - Initialize a list. Can only be called from @@BEGIN action block
void append(List, long long)
    - Append an item to a list
long long sum(List)
long long avg(List)
long long min(List)
long long max(List)
long long count(List)
```

### Probe point information

- While globbing is not allowed, multiple probes can be specified for an action block. Use the following functions to find what fired.

```
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
```

### Tentative 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)
    - Get the size of a data structure
String get_userstring(pointer, length)
    - Copy a string from userspace. Set length to -1 to copy to EOL.
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.
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
```

### 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` script header. There is also no “pragma” option to include these items in a script.

```
-I <FILE>   Use FILE as include file
-o <FILE>   Use FILE as output destination
-X <PROG>   Start PROG as watched process
-A <ARGS>   Arguments to -X PROG
-K          Enable RAS functions
```

- 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
```

## Shell Wrapper Example

- A shell wrapper is useful for
  - ⇒parameter validation
  - ⇒proper command line option parsing
  - ⇒simplified command packaging
- Note in the following example that  `${1}`  is a shell variable and is expanded by the shell before it is passed to probevue.
- The  `${1}`  is wrapped in quotes so that probevue will properly recognize it as a string.

```
#!/bin/ksh

if [[ -z ${1} ]]
then
    print "ERROR: Missing required parameter."
    exit
fi

/bin/probevue <<EOF
#!/bin/probevue
```

```
String filename[1024];
```

```
@@BEGIN
{
    filename = \"${1}\";
    printf("String is %s.\n", filename);
    exit();
}
```

```
EOF
```

## 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 yoursript.e
```

- or -

```
probevue -I header1.i,header2.i yoursript.e
```
- There is no `#include` or `#pragma` option from *within* a script

## Additional Information

- <http://www.ibm.com/developerworks/aix/library/au-probevue>

## 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.  $\LaTeX$ source not distributed at this time.