

# **Linux Kernel GDB tracepoint module (KGTP)**

Update in 2014-06-24

## TOC

About this document.....	6
What is KGTP.....	7
Quick config and start KGTP.....	8
Get help or report issues.....	10
Table of different between GDB debug normal program and KGTP.....	11
Howto use GDB control KGTP trace and debug Linux kernel.....	14
Direct access the current value in normal mode.....	14
The memory of Linux kernel.....	14
the trace state variables.....	16
GDB tracepoint.....	17
set tracepoint.....	18
Howto handle the function is there but set tracepoint on it got fail....	19
How to set tracepoint condition.....	20
How to handle error "Unsupported operator (null) (52) in expression."	21
actions [num].....	22
collect expr1, expr2, .....	23
teval expr1, expr2, .....	24
while-stepping n.....	25
Start and stop the tracepoint.....	26
Enable and disable the tracepoint.....	27
Use tfind select the entry inside the trace frame info.....	28
How to handle error "No such file or directory.".....	29
Save the trace frame info to a file.....	30
Show and save the tracepoint.....	31
Delete tracepoint.....	32
Use tracepoint get register info from a point of kernel.....	33
Use tracepoint get the value of variable from a point of kernel.....	35
Show all the traced data of current frame.....	36
Get status of tracepoint.....	37
Set the trace buffer into a circular buffer.....	38
Do not stop tracepoint when the GDB disconnects.....	39
kprobes-optimization and the execution speed of tracepoint.....	39
How to use trace state variables.....	40
Simple trace state variables.....	41
Per_cpu trace state variables.....	43
How to define.....	44
Local CPU variables.....	44
CPU id variables.....	44
Example 1.....	45
Example 2.....	46
Special trace state variables \$current_task, \$current_task_pid, \$current_thread_info, \$cpu_id, \$dump_stack, \$printk_level, \$printk_format, \$printk_tmp, \$clock, \$hardirq_count, \$softirq_count and \$irq_count.....	47
Special trace state variable \$self_trace.....	50

Trace the function return with \$kret.....	51
Use \$ignore_error and \$last_errno to ignore the error of tstart.....	52
Use \$cooked_clock and \$cooked_rdtsc the time without KGTP used.....	53
Use \$xtime_sec and \$xtime_nsec get the timespec.....	54
Howto backtrace (stack dump).....	55
Collect stack with \$bt and use GDB command "backtrace".....	56
Collect stack of current function's caller with \$_ret.....	59
Use \$dump_stack to output stack dump through printk.....	61
Howto let tracepoint output value directly.....	63
Switch collect to output the value directly.....	63
Howto use watch tracepoint control hardware breakpoints to record memory access.....	65
Trace state variables of watch tracepoint.....	65
Static watch tracepoint.....	68
Dynamic watch tracepoint.....	69
Use while-stepping let Linux kernel do single step.....	70
Howto use while-stepping.....	70
Read the traceframe of while-stepping.....	71
Howto show a variable whose value has been optimized away.....	73
Update your GCC.....	73
Get the way that access the variable that has been out through parse ASM code.....	74
How to get the function pointer point to.....	77
If the debug info of the function pointer is not optimized out.....	77
If the debug info of the function pointer is optimized out.....	78
/sys/kernel/debug/gtpframe and offline debug.....	79
How to use /sys/kernel/debug/gtpframe_pipe.....	81
Get the frame info with GDB.....	82
Get the frame info with cat.....	83
Get the frame info with getframe.....	84
Use \$pipe_trace.....	85
Use KGTP with user applications.....	86
Let GDB connect KGTP for user applications.....	86
Read memory of user applications directly.....	87
Trace user applications.....	88
collect stack (for backtrace) of system from Linux kernel to user applications in tracepoint.....	90
How to use add-ons/hotcode.py.....	93
How to add plugin in C.....	94
API.....	94
Example.....	96
How to use.....	97
How to use performance counters.....	98
Define a perf event trace state variable.....	99
Define a per_cpu perf event trace state variable.....	100
The perf event type and config.....	101
Enable and disable all the perf event in a CPU with \$p_pe_en.....	103
GDB scripts to help with set and get the perf event trace state variables	

.....	104
Appendix A Preparatory work before use KGTP.....	107
Linux kernel.....	107
If your system use the Linux kernel that is built by yourself.....	107
If use with Linux kernel of Android.....	108
If your system use the Linux kernel from distribution.....	109
Ubuntu.....	109
The standard method of install the Linux kernel debug image.....	109
The second method of install the Linux kernel debug image.....	110
Install the Linux kernel headers.....	110
Install the Linux kernel source.....	110
New way.....	110
Old way.....	110
Fedora.....	111
Install the Linux kernel debug image.....	111
Install the Linux kernel devel package.....	111
Make sure current Linux kernel debug image is right.....	112
Where is the current Linux kernel debug image.....	112
Use /proc/kallsyms.....	113
Use linux_banner.....	114
Handle the issue that Linux kernel debug image's address info is not same with Linux kernel when it running.....	115
Get KGTP.....	116
Get KGTP through http.....	116
Get KGTP through git.....	117
Mirrors.....	118
Config KGTP.....	119
Compile KGTP.....	120
Normal compile.....	120
Compile KGTP with some special config.....	121
Install and uninstall KGTP.....	122
Use KGTP with DKMS.....	123
Use KGTP patch for Linux kernel.....	124
Install GDB for KGTP.....	125
Appendix B How to let GDB connect to KGTP.....	126
Normal Linux.....	126
Insmod the KGTP module.....	126
Handle the issue that cannot find "/sys/kernel/debug/gtp".....	127
Make GDB connect to gtp.....	128
Load Linux kernel debug image to GDB.....	128
GDB on the current machine.....	129
GDB on remote machine.....	130
Android.....	131
Insmod the KGTP module.....	131
Handle the issue that cannot find "/sys/kernel/debug/gtp".....	132
GDB connect to the KGTP.....	133
Appendix C Add module symbols to GDB.....	134
How to use getmod.....	134

How to use getmod.py.....	136
---------------------------	-----

# About this document

<http://teawater.github.io/kgtp/kgtp.html> is the the last version of this document in HTML format.

<https://raw.githubusercontent.com/teawater/kgtp/master/kgtp.pdf> is the the last version of this document in PDF format.

<https://raw.githubusercontent.com/teawater/kgtp/release/kgtp.pdf> is the the last release version of this document in PDF format.

# What is KGTP

**KGTP** is a comprehensive dynamic tracer for analysing Linux kernel and application (including Android) problems on production systems in real time.

To use it, you don't need patch or rebuild the Linux kernel. Just build KGTP module and insmod it is OK.

It makes Linux Kernel supply a GDB remote debug interface. Then GDB in current machine or remote machine can debug and trace Linux kernel and user space program through GDB tracepoint and some other functions without stopping the Linux Kernel.

And even if the board doesn't have GDB on it and doesn't have interface for remote debug. It can debug the Linux Kernel using offline debug (See [/sys/kernel/debug/gtpframe and offline debug](#)).

<http://www.youtube.com/watch?v=7nfGAbNsEZY> or [http://www.tudou.com/programs/view/fPu\\_koiKo38/](http://www.tudou.com/programs/view/fPu_koiKo38/) is the video that introduced KGTP in **English**.

<http://www.infoq.com/cn/presentations/gdb-sharp-knife-kgtp-linux-kernel> is the video that introduced KGTP in **Chinese**.

KGTP supports **X86-32, X86-64, MIPS** and **ARM**.

KGTP supports most versions of Linux kernel (from **2.6.18** to **upstream**).

Please go to [UPDATE](#) to get more info about KGTP update.

# Quick config and start KGTP

```
#kgtp.py will auto setup and start KGTP and GDB in current
machine.
#The first time you use this script needs to wait for a while because
there are some packages to download.
wget
https://raw.githubusercontent.com/teawater/kgtp/master/kgtp.py
sudo python kgtp.py
#Access memory of Linux kernel.
(gdb) p jiffies_64
$2 = 5081634360
#Set tracepoint in function vfs_read to collect its backtrace.
(gdb) trace vfs_read
Tracepoint 1 at 0xffffffff811b8c70: file fs/read_write.c, line 382.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $bt
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0 vfs_read (file=file@entry=0xffff88022017b000,
  buf=buf@entry=0x7fff0fdd80f0 <Address 0x7fff0fdd80f0 out of
  bounds>,
  count=count@entry=16, pos=pos@entry=0xffff8800626aff50) at
  fs/read_write.c:382
382 {
(gdb) bt
#0 vfs_read (file=file@entry=0xffff88022017b000,
  buf=buf@entry=0x7fff0fdd80f0 <Address 0x7fff0fdd80f0 out of
  bounds>,
  count=count@entry=16, pos=pos@entry=0xffff8800626aff50) at
  fs/read_write.c:382
#1 0xffffffff811b9819 in SYSC_read (count=16,
  buf=0x7fff0fdd80f0 <Address 0x7fff0fdd80f0 out of bounds>,
  fd=<optimized out>)
  at fs/read_write.c:506
```

Please read [Appendix A Preparatory work before use KGTP](#) and [Appendix B](#)



[How to let GDB connect to KGTP](#) if you want to use KGTP in remore machine or android.

# Get help or report issues

Please post issues to to <https://github.com/teawater/kgtp/issues>.

Or mail them to <mailto:teawater@gmail.com?Subject=Report%20an%20issue%20of%20KGTP>.

Or report it to QQ group **317654748**.

The KGTP team will try our best to help you.

Please goto <https://code.google.com/p/kgtp/issues/list> access the old issues list.

# Table of different between GDB debug normal program and KGTP

This table is for the people that have experience using GDB debug normal program. It will help you understand and remember the function of KGTP.

Function	GDB debug normal program	GDB control KGTP debug Linux kernel
Preparatory work	Have a GDB installed in your system. Program built with "-g".	<a href="#">Quick config and start KGTP</a>
Attach	Use command "gdb -p pid" or GDB command "attach pid" can attach a program that running in the system.	
Breakpoints	GDB command "b place_will_stop", let program execute after this command. Then program will stop in the place that setup a breakpoint.	KGTP doesn't support breakpoints but it support tracepoints. Tracepoints can be considered as a special kind of breakpoints. It can be setup in some place of Linux kernel and define some commands that you want to do in its actions. When tracepoints start, they will execute these commands when Linux kernel execute to these place. When tracepoint stop, you can use some GDB commands parse the data that get by tracepoints like what you do when program stop by breakpoints. <b>Difference</b> is breakpoints will stop the program But the

		<p>tracepoints of KGTP not. Please goto <a href="#">GDB tracepoint</a> get howto use it.</p>
Memory read	<p>After GDB stop the program(maybe doesn't need), it can read memory of program with GDB command "print", "x" and so on.</p>	<p>You can set special actions to collect memory to traceframe in tracepoints, and get the its value when tracepoint stop.<a href="#">collect expr1, expr2, ... Use tfind select the entry inside the trace frame info</a></p> <p>Or you can read memory directly when Linux kernel or program is running.<a href="#">Direct access the current value in normal mode</a></p>
Step and continue	<p>GDB can continue program execution with command "continue" and stop it with CTRL-C.</p>	<p>KGTP never stop the Linux kernel. But tracepoint can be start and stop.<a href="#">Start and stop the tracepoint</a></p> <p>Or use while-stepping tracepoint record Linux kernel with some times single step and Let KGTP switch to replay mode. Then it support execution commands (continue, step) and reverse-execute commands (reverse-continue, reverse-step). <a href="#">Use while-stepping let Linux kernel do single step</a></p>
Backtrace	<p>GDB can print backtrace of all stack frames with command "backtrace".</p>	<p>KGTP can do it too.<a href="#">Howto backtrace (stack dump)</a></p>
Watchpoint	<p>GDB can let programe stop when some memory access happen with watchpoint.</p>	<p>KGTP can record the memory access with watch tracepoint. <a href="#">Howto use watch tracepoint control hardware</a></p>

		<a href="#"><u>breakpoints to record memory access</u></a>
Call function	GDB can call function of program with command "call function(xx,xx)".	KGTP can call function of Linux kernel with plugin. <a href="#"><u>How to add plugin in C</u></a>

# Howto use GDB control KGTP trace and debug Linux kernel

## Direct access the current value in normal mode

After GDB connect to KGTP, if it doesn't select any a entry of trace frame buffer with GDB command "tfind", GDB in the normal mode. Then you can direct access the current value of memory (Linux kernel or the user space program) and the trace state variables without stop anything.

If you have selected a trace frame entry, use GDB command "tfind -1" to return to normal mode. Please goto [Use tfind select the entry inside the trace frame info](#) info about GDB command "tfind".

## The memory of Linux kernel

For example, you can access to "jiffies\_64" with following command:

```
(gdb) p jiffies_64
```

Or you can access to the first entry of "static LIST\_HEAD(modules)" with following command:

```
(gdb) p *((struct module *)((char *)modules->next - ((size_t) &((struct module *)0)->list)))
```

Or you can access to the CPU0 memory info of "DEFINE\_PER\_CPU(struct device \*, mce\_device);":

```
p *(struct device *)(__per_cpu_offset[0]+(uint64_t)(&mce_device))
```

If you want show more than one variables with one GDB command, please use following example:

```
(gdb) printf "%4d %4d %4d %4d %4d %4d %18d %lu\n", this_rq->cpu, this_rq->nr_running, this_rq->nr_uninterruptible, nr_active, calc_load_tasks->counter, this_rq->calc_load_active, delta, this_rq->
```

```
>calc_load_update  
2 1 0 0 0 0 673538312 717077240
```

## **the trace state variables**

You can access value of TSV with the command that same with access memory.

Please goto [How to use trace state variables](#) get more info about TSV.



# GDB tracepoint

Tracepoint is that GDB define some addresses and some actions and put them to the target (KGTP). After tracepoint start, , KGTP will do these actions (Some of them will collect data and save them to tracepoint frame buffer) when Linux kernel execution to there addresses. After that, Linux kernel will keep execution.

KGTP supply some interfaces that GDB or other programe can take the data of tracepoint frame buffer out to parse.

About these interfaces, this doc have introduced `"/sys/kernel/debug/gtp"`. And will introduce `"/sys/kernel/debug/gtpframe"` and `"/sys/kernel/debug/gtpframe_pipe"` later.

Doc of GDB tracepoint in

<http://sourceware.org/gdb/current/onlinedocs/gdb/Tracepoints.html>.

## set tracepoint

The trace command is very similar to the break command. Its argument location can be a source line, a function name, or an address in the target program. The trace command defines a tracepoint, which is a address or some addresses that KGTP do some actions in it.

Here are some examples of using the trace command:

*(gdb) trace foo.c:121 // a source file and line number*

*(gdb) trace +2 // 2 lines forward*

*(gdb) trace my\_function // first source line of function*

*(gdb) trace \*my\_function // EXACT start address of function*

*(gdb) trace \*0x2117c4 // an address*

## **Howto handle the function is there but set tracepoint on it got fail**

GCC will inline some static function to increase the performance. You cannot set tracepoint on the function name because object file doesn't have symbol of inline function.

You can use "trace filename:line" to set tracepoint on it.

## How to set tracepoint condition

<http://sourceware.org/gdb/current/onlinedocs/gdb/Tracepoint-Conditions.html>

Like breakpoints, we can set conditions on tracepoints. The speed of tracepoints is faster than breakpoints because KGTP can do all the condition checks.

For example:

```
(gdb) trace handle_irq if (irq == 47)
```

This action of tracepoint 1 will work only when irq number is 47.

And you can use GDB command "condition" to specify the condition of a tracepoint. GDB command "condition N COND" will specify tracepoint number N to trace only if COND is true.

For example:

```
(gdb) trace handle_irq  
(gdb) condition 1 (irq == 47)
```

GDB command "info tracepoint" will show the ID of the tracepoint.

Value of \$bpnum is the last ID of GDB tracepoint. Then you can use GDB command "condtion" set the condition of last tracepoint without get its ID. For example:

```
(gdb) trace handle_irq  
(gdb) condition $bpnum (irq == 47)
```

## How to handle error "Unsupported operator (null) (52) in expression."

If you use condition about string, you will get this error when you call "tstart".

To handle it, you can convert the char to int to handle this issue, for example:

```
(gdb) p/x 'A'
```

```
$4 = 0x41
```

```
(gdb) condition 1 (buff[0] == 0x41)
```

## **actions [num]**

This command will prompt for a list of actions to be taken when the tracepoint is hit. If the tracepoint number `num` is not specified, this command sets the actions for the one that was most recently defined (so that you can define a tracepoint and then say actions without bothering about its number). You specify the actions themselves on the following lines, one action at a time, and terminate the actions list with a line containing just `end`. So far, the only defined actions are `collect`, `teval`, and `while-stepping`.

## **collect expr1, expr2, ...**

Collect values of the given expressions when the tracepoint is hit. This command accepts a comma-separated list of any valid expressions. In addition to global, static, or local variables, the following special arguments are supported:

*\$regs Collect all registers.*  
*\$args Collect all function arguments.*  
*\$locals Collect all local variables.*

Please **note** that collect an pointer (collect ptr) will just collect the address of this pointer. Add a \* before ptr will make action collect the data that pointer point to (collect \*ptr).

## **teval expr1, expr2, ...**

Evaluate the given expressions when the tracepoint is hit. This command accepts a comma-separated list of expressions. The results are discarded, so this is mainly useful for assigning values to trace state variables (see [Simple trace state variables](#)) without adding those values to the trace buffer, as would be the case if the collect action were used.



## **while-stepping n**

Please goto [Use while-stepping let Linux kernel do single step](#) see howto use it.

## Start and stop the tracepoint

Tracepoint will exec actions only when it is starting use this GDB command:

*(gdb) tstart*

It will stop by this GDB command:

*(gdb) tstop*

## Enable and disable the tracepoint

Like breakpoint, tracepoint can be control by GDB commands "enable" and "disable". But please **note** that it only useful when tracepoint stop.

## Use tfind select the entry inside the trace frame info

GDB command "tfind" is used to select a entry of trace frame bufffer when tracepoint stop.

When GDB inside "tfind" mode, it will just show the values of this entry that the tracepoint action collect. So it will output some error when print some values that action doesn't collect for example the argument of function. That is not a bug, please don't worry about it.

Use "tfind" again will select next entry. "tfind id" will select entry id.

To return to normal mode([Direct access the current value in normal mode](#)), please use GDB command "tfind -1". Please goto <http://sourceware.org/gdb/current/onlinedocs/gdb/tfind.html> get more info about it.

## How to handle error "No such file or directory."

When GDB cannot find the source code of Linux kernel, it will show this error message. For example:

```
(gdb) tfind
Found trace frame 1, tracepoint 1
#0  vfs_read (file=0xffff8801c36e6400, buf=0x7fff51a8f110
<Address 0x7fff51a8f110 out of bounds>, count=16,
    pos=0xffff8801761dff48) at /build/builddd/linux-
3.2.0/fs/read_write.c:365
365  /build/builddd/linux-3.2.0/fs/read_write.c: No such file or
directory.
```

You can use GDB command "set substitute-path" to handle it. The previous example, the Linux kernel source is in "/build/builddd/test/linux-3.2.0/". But vmlinux let GDB find Linux kernel source in "/build/builddd/linux-3.2.0/". You can handle it with:

```
(gdb) set substitute-path /build/builddd/linux-3.2.0/
/build/builddd/test/linux-3.2.0/
(gdb) tfind
Found trace frame 1, tracepoint 1
#0  vfs_read (file=0xffff8801c36e6400, buf=0x7fff51a8f110
<Address 0x7fff51a8f110 out of bounds>, count=16,
    pos=0xffff8801761dff48) at /build/builddd/linux-
3.2.0/fs/read_write.c:365
365  {
```

GDB has some other commands to handle the source code issue. Please go to <http://sourceware.org/gdb/current/onlinedocs/gdb/Source-Path.html> get the introduction about them.

## Save the trace frame info to a file

/sys/kernel/debug/gtpframe supplies trace frame in tfile format (GDB can parse it) when KGTP is stop.

Please **note** that some "cp" cannot handle it very well, please use "cat /sys/kernel/debug/gtpframe > ./gtpframe" to copy it.

You can open file gtpframe when you want:

```
(gdb) target tfile ./gtpframe
Tracepoint 1 at 0xffffffff8114f3dc: file /home/teawater/kernel/linux-
2.6/fs/readdir.c, line 24.
Created tracepoint 1 for target's tracepoint 1 at 0xffffffff8114f3c0.
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_readdir (file=0xffff880036e8f300, filler=0xffffffff8114f240
<filldir>, buf=0xffff880001e5bf38)
    at /home/teawater/kernel/linux-2.6/fs/readdir.c:24
24    {
```

## **Show and save the tracepoint**

You can use GDB command "info tracepoints" to show all the tracepoints.

You can use GDB command "save tracepoints filename" to save the commands that setup the tracepoints and actions into file filename. Then you use use GDB commands "source filename" to setup this tracepints again.

## **Delete tracepoint**

GDB command "delete id" will delete tracepoint id. If "delete" without argument, it will delete all the tracepoint.



## Use tracepoint get register info from a point of kernel

The following is an example that records the value of all registers when "vfs\_readdir" is called.

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_readdir
Tracepoint 1 at 0xc01a1ac0: file
/home/teawater/kernel/linux-2.6/fs/readdir.c, line 23.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $reg
>end
(gdb) tstart
(gdb) shell ls
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0 0xc01a1ac1 in vfs_readdir (file=0xc5528d00, filler=0xc01a1900
<filldir64>,
    buf=0xc0d09f90) at /home/teawater/kernel/linux-
2.6/fs/readdir.c:23
23    /home/teawater/kernel/linux-2.6/fs/readdir.c: No such file or
directory.
    in /home/teawater/kernel/linux-2.6/fs/readdir.c
(gdb) info reg
eax      0xc5528d00      -984445696
ecx      0xc0d09f90      -1060069488
edx      0xc01a1900      -1072031488
ebx      0xffffffff7      -9
esp      0xc0d09f8c      0xc0d09f8c
ebp      0x0      0x0
esi      0x8061480      134616192
edi      0xc5528d00      -984445696
eip      0xc01a1ac1      0xc01a1ac1 <vfs_readdir+1>
eflags   0x286 [ PF SF IF ]
cs       0x60      96
ss       0x8061480      134616192
ds       0x7b      123
es       0x7b      123
fs       0x0      0
gs       0x0      0
(gdb) tfind
```

Found trace frame 1, tracepoint 1

0xc01a1ac1 23 in /home/teawater/kernel/linux-2.6/fs/readdir.c

(gdb) info reg

eax	0xc5528d00	-984445696
ecx	0xc0d09f90	-1060069488
edx	0xc01a1900	-1072031488
ebx	0xffffffff7	-9
esp	0xc0d09f8c	0xc0d09f8c
ebp	0x0	0x0
esi	0x8061480	134616192
edi	0xc5528d00	-984445696
eip	0xc01a1ac1	0xc01a1ac1 <vfs_readdir+1>
eflags	0x286	[ PF SF IF ]
cs	0x60	96
ss	0x8061480	134616192
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x0	0

## Use tracepoint get the value of variable from a point of kernel

The following is an example that records the value of "jiffies\_64" when the function "vfs\_readdir" is called:

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_readdir
Tracepoint 1 at 0xc01ed740: file /home/teawater/kernel/linux-2.6/fs/readdir.c, line 24.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect jiffies_64
>collect file->f_path.dentry->d_iname
>end
(gdb) tstart
(gdb) shell ls
arch  drivers  include  kernel  mm          Module.symvers
security System.map virt
block  firmware init    lib      modules.builtin net          sound    t
vmlinux
crypto fs      ipc    Makefile modules.order scripts    source
usr      vmlinux.o
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  0xc01ed741 in vfs_readdir (file=0xf4063000, filler=0xc01ed580
<filldir64>, buf=0xd6dfdf90)
    at /home/teawater/kernel/linux-2.6/fs/readdir.c:24
24    {
(gdb) p jiffies_64
$1 = 4297248706
(gdb) p file->f_path.dentry->d_iname
$1 = "b26", '\000' <repeats 28 times>
```

## Show all the traced data of current frame

After use "tfind" select an entry, you can use "tdump" to do it.

```
(gdb) tdump  
Data collected at tracepoint 1, trace frame 0:  
$cr = void  
file->f_path.dentry->d_iname =  
"gtp\000.google.chrome.g05ZYO\000\235\337\000\000\000\000\200\  
067k\364\200\067", <incomplete sequence \364>  
jiffies_64 = 4319751455
```

## **Get status of tracepoint**

Please use GDB command "tstatus".

## Set the trace buffer into a circular buffer

<http://sourceware.org/gdb/current/onlinedocs/gdb/Starting-and-Stopping-Trace-Experiments.html>

The frame buffer is not a circular buffer by default. When the buffer is full, the tracepoint will stop.

Following command will set frame buffer to a circular buffer. When the buffer is full, it will auto discard traceframes (oldest first) and keep trace.

*(gdb) set circular-trace-buffer on*

## Do not stop tracepoint when the GDB disconnects

<http://sourceware.org/gdb/current/onlinedocs/gdb/Starting-and-Stopping-Trace-Experiments.html>

KGTP will stop tracepoint and delete the trace frame when GDB disconnects with it by default.

Following command will open the KGTP disconnect-trace. After that, when GDB disconnects with KGTP, KGTP will not stop tracepoint. And after GDB reconnects to KGTP, it can keep control of KGTP like nothing happened.

*(gdb) set disconnected-tracing on*

## kprobes-optimization and the execution speed of tracepoint

The tracepoint is execution together with Linux kernel. So its speed will affect the speed of the system.

The KGTP tracepoint is based on Linux kernel kprobe. Because the normal kprobe is based on breakpoint instruction, so it is not very fast.

But if the arch of kernel is X86\_64 or X86\_32 and kernel config didn't open "Preemptible Kernel" (PREEMPT), the kprobe is speeded up by kprobes-optimization (CONFIG\_OPTPROBES) that makes kprobe very fast.

To make sure about that, you can use the following command in terminal:

*sysctl -A | grep kprobe*  
*debug.kprobes-optimization = 1*

That means that your kernel supports kprobes-optimization.

Please **note** that some KGTP functions will make this tracepoint use simple kprobe even if this kernel supports kprobes-optimization. This doc will add a note when introducing these functions. Please avoid using them when you really care about the tracepoint speed.

# How to use trace state variables

<http://sourceware.org/gdb/current/onlinedocs/gdb/Trace-State-Variables.html>

Trace state variable is referred to as the TSV.

TSV can be accessed in tracepoint action and condition or direct access by GDB command.

Please **note** that just GDB 7.2.1 and later versions support use trace state variables directly, the old version of GDB can show the value of trace state variables through command "info tvariables".



## Simple trace state variables

Define a trace state variable \$c.

```
(gdb) tvariable $c
```

Trace state variable \$c is created with initial value 0. The following action uses \$c to count how many irqs happened in the kernel.

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace handle_irq
(gdb) actions
Enter actions for tracepoint 3, one per line.
End with a line saying just "end".
>collect $c    #Save current value of $c to the trace frame buffer.
>teval $c=$c+1 #Increase the $c.
>end
```

Also, you can set a value of variable to trace state variable, but don't forget covert variable to "uint64\_t".

```
>teval $c=(uint64_t)a
```

You can get the current value of \$c while the trace is running or stopped.

```
(gdb) tstart
(gdb) info tvariables
$c          0          31554
(gdb) p $c
$5 = 33652
(gdb) tstop
(gdb) p $c
$9 = 105559
```

When using tfind, you can parse the trace frame buffer. If the value of a trace state variable is collected, you can parse it out.

```
(gdb) tstop
(gdb) tfind
(gdb) info tvariables
$c          0          0
(gdb) p $c
$6 = 0
(gdb) tfind 100
(gdb) p $c
$7 = 100
```

If need, the tracepoint action that access the simple trace state variables will auto lock a spin lock for trace state variables. So it can handle race condition issue about trace state variables.

The following example is OK even if it running a machine that have more than one CPU.

```
>teval $c=$c+1
```

## **Per\_cpu trace state variables**

Per\_cpu trace state variables are special simple trace state variables.

When tracepoint action access to it, it will access to this CPU special trace state variables.

It have 2 advantages:

1. The tracepoint actions that access to per\_cpu trace state variables don't have the race conditon issue. So it don't need lock the spin lock for trace state variables. It is faster than simple trace state variables on multi-core machine.
2. Write the action that count some CPU special thing with it is easier than simple trace state variables.

## How to define

Per\_cpu trace state variables have two types:

### Local CPU variables

*"per\_cpu\_" + string*

or

*"p\_" + string*

For example:

*(gdb) tvariable \$p\_count*

When access this trace state variable in tracepoint actions, it will return the variable's value of CPU that this tracepoint actions running on.

### CPU id variables

*"per\_cpu\_" + string + CPU\_id*

or

*"p\_" + string + CPU\_id*

For example:

*(gdb) tvariable \$p\_count0*

*(gdb) tvariable \$p\_count1*

*(gdb) tvariable \$p\_count2*

*(gdb) tvariable \$p\_count3*

When access this trace state variable in tracepoint actions or GDB command line, it will return the variable's value of CPU CPU\_id.

Follow example can auto define a CPU id variables for each CPU of this machine. (Please note that need let GDB connect to KGTP before use these commands.)

*(gdb) set \$tmp=0*

*(gdb) while \$tmp < \$cpu\_number*

*>eval "tvariable \$p\_count%d", \$tmp*

*>set \$tmp=\$tmp+1*

*>end*

## Example 1

This example define a tracepoint that count the times that call `vfs_read` of each CPU.

```
tvariable $p_count
set $tmp=0
while $tmp<$cpu_number
    eval "tvariable $p_count%d", $tmp
    set $tmp=$tmp+1
end
trace vfs_read
actions
    teval $p_count=$p_count+1
end
```

Then you can show how many `vfs_read` in each CPU after "tstart":

```
(gdb) p $p_count0
$3 = 44802
(gdb) p $p_count1
$4 = 55272
(gdb) p $p_count2
$5 = 102085
(gdb) p $p_count3
```

## Example 2

This example record stack dump of the function that close IRQ longest time of each CPU.

```
set pagination off
```

```
tvariable $bt=1024  
tvariable $p_count  
tvariable $p_cc  
set $tmp=0  
while $tmp<$cpu_number  
eval "tvariable $p_cc%d",$tmp  
set $tmp=$tmp+1  
end
```

```
tvariable $ignore_error=1
```

```
trace arch_local_irq_disable  
commands  
teval $p_count=$clock  
end
```

```
trace arch_local_irq_enable if ($p_count && $p_cc < $clock -  
$p_count)  
commands  
teval $p_cc = $clock - $p_count  
collect $bt  
collect $p_cc  
teval $p_count=0  
end
```

```
enable
```

```
set pagination on
```

## **Special trace state variables \$current\_task, \$current\_task\_pid, \$current\_thread\_info, \$cpu\_id, \$dump\_stack, \$printk\_level, \$printk\_format, \$printk\_tmp, \$clock, \$hardirq\_count, \$softirq\_count and \$irq\_count**

KGTP special trace state variables \$current\_task, \$current\_thread\_info, \$cpu\_id and \$clock can very easy to access to some special value. You can see them when GDB connects to the KGTP. You can use them in tracepoint conditions or actions.

Access \$current\_task in tracepoint condition and action will get that returns of get\_current().

Access \$current\_task\_pid in tracepoint condition and action will get that returns of get\_current()->pid.

Access \$current\_thread\_info in tracepoint condition and action will get that returns of current\_thread\_info().

Access \$cpu\_id in tracepoint condition and action will get that returns of smp\_processor\_id().

Access \$clock in tracepoint condition and action will get that returns of local\_clock() that return the timestamp in nanoseconds.

\$rdtsc is only available on X86 and X86\_64 architecture. Access it in anytime will get current value of TSC with instruction RDTSC.

Access \$hardirq\_count in tracepoint condition and action will get that returns of hardirq\_count().

Access \$softirq\_count in tracepoint condition and action will get that returns of softirq\_count().

Access \$irq\_count in tracepoint condition and action will get that returns of irq\_count().

And KGTP has other special trace state variables \$dump\_stack, \$printk\_level, \$printk\_format and \$printk\_tmp. All of them output their values directly, as can be seen in [Howto let tracepoint output value directly](#).

The following example counts in \$c how many vfs\_read calls that process 16663 does and collects the struct thread\_info of current task:

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_read if (((struct task_struct *)$current_task)->pid == 16663)
(gdb) tvariable $c
```

```

(gdb) actions
Enter actions for tracepoint 4, one per line.
End with a line saying just "end".
>teval $c=$c+1
>collect (*(struct thread_info *)$current_thread_info)
>end
(gdb) tstart
(gdb) info tvariables
Name          Initial    Current
$c             0         184
$current_task  0         <unknown>
$current_thread_info 0         <unknown>
$cpu_id        0         <unknown>
(gdb) tstop
(gdb) tfind
(gdb) p *(struct thread_info *)$current_thread_info
$10 = {task = 0xf0ac6580, exec_domain = 0xc07b1400, flags = 0,
status = 0, cpu = 1, preempt_count = 2, addr_limit = {
  seg = 4294967295}, restart_block = {fn = 0xc0159fb0
<do_no_restart_syscall>, {{arg0 = 138300720, arg1 = 11,
  arg2 = 1, arg3 = 78}, futex = {uaddr = 0x83e4d30, val = 11,
flags = 1, bitset = 78, time = 977063750,
  uaddr2 = 0x0}, nanosleep = {index = 138300720, rmtp = 0xb,
expires = 335007449089}, poll = {
  ufds = 0x83e4d30, nfds = 11, has_timeout = 1, tv_sec = 78,
tv_nsec = 977063750}}}},
  sysenter_return = 0xb77ce424, previous_esp = 0, supervisor_stack
= 0xef340044 "", uaccess_err = 0}

```

Another example shows how much `sys_read()` executes in each CPU.

```

(gdb) tvariable $c0
(gdb) tvariable $c1
(gdb) trace sys_read
(gdb) condition $bpnum ($cpu_id == 0)
(gdb) actions
>teval $c0=$c0+1
>end
(gdb) trace sys_read
(gdb) condition $bpnum ($cpu_id == 1)
(gdb) actions
>teval $c1=$c1+1
>end
(gdb) info tvariables
Name          Initial    Current
$current_task  0         <unknown>
$cpu_id        0         <unknown>
$c0            0         3255

```



*\$c1*            *0*            *1904*

sys\_read() execute 3255 times in cpu0 and 1904 times in cpu1. Please note that this example just to howto use \$cpu\_id. Actially, this example use per\_cpu trace state variables is better.

## Special trace state variable `$self_trace`

`$self_trace` is different with the special trace state variables in the previous section. It is used to control the behavior of tracepoint.

In default, when tracepoint is triggered, the actions will not execute if the `current_task` is the a KGTP self process (GDB, netcat, getframe or some others process that access to the interface of KGTP).

If you want tracepoint actions execute with any task, please include a command access to the `$self_trace` in the actions i.e. add following command to the actions:

```
>teval $self_trace=0
```

## Trace the function return with \$kret

Sometime, set the tracepoint to the end of function is hard because the Kernel is compiled with optimization. At this time, you can get help from \$kret.

\$kret is a special trace state variable like \$self\_trace. When you set value of it inside the action of tracepoint, this tracepoint be set with kretprobe instead of kprobe. Then it can trace the end of this function.

Please note that this tracepoint must set in the first address of the function in format "**function\_name**".

Following part is an example:

```
#"(function_name)" format can make certain that GDB send the  
first address of function to KGTP.  
(gdb) trace *vfs_read  
(gdb) actions  
>teval $kret=0  
#Following part you can set commands that you want.
```

## Use `$ignore_error` and `$last_errno` to ignore the error of `tstart`

If KGTP got any error of `tstart`, this command will get fail.

But sometime we need ignore this error and let KGTP keep work. For example: If you set tracepoint on the inline function `spin_lock`. This tracepoint will be set to a lot of addresses that some of them cannot be set `kprobe`. It will make `tstart` get fail. You can use "`$ignore_error`" ignore this error.

And the last error number will available in "`$last_errno`".

*(gdb) tvariable \$ignore\_error=1*

This command will open ignore.

*(gdb) tvariable \$ignore\_error=0*

This command will close ignore.

## **Use \$cooked\_clock and \$cooked\_rdtsc the time without KGTP used**

Access these two trace state variables can get the time without KGTP used. Then we can get more close to really time that a part of code used even if the actions of tracepoint is very complex. They will be introduce in Cookbook (coming soon).

## **Use `$xtime_sec` and `$xtime_nsec` get the `timespec`**

Access these two trace state variables will return the time of day in a `timespec` that use `getnstimeofday`.

`$xtime_sec` will access to the second part of a `timespec`.

`$xtime_nsec` will access to the nanosecond part of a `timespec`.

# **Howto backtrace (stack dump)**

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a stack frame. The stack frames are allocated in a region of memory called the call stack.

## Collect stack with \$bt and use GDB command "backtrace"

Because this way is faster (just collect the stack when trace) and parse out most of info inside the call stack (it can show all the stack info that I introduce). So I suggest you use this way to do the stack dump.

First we need add the collect the stack command to the tracepoint action.

~~The general collect the stack command in GDB tracepoint is: In x86\_32, following command will collect 512 bytes of stack.~~

~~>collect \*(unsigned char \*)\$esp@512~~

~~In x86\_64, following command will collect 512 bytes of stack.~~

~~>collect \*(unsigned char \*)\$rsp@512~~

~~In MIPS or ARM, following command will collect 512 bytes of stack.~~

~~>collect \*(unsigned char \*)\$sp@512~~

~~These commands is so hard to remember, and the different arch need different command.~~

KGTP have an special tracepoint trace state variable \$bt. If tracepoint action access it, KGTP will auto collect the \$bt size (default value is 512) stack. For example, this command will collect 512 bytes stack memory:

*>collect \$bt*

If you want to change size of \$bt, you can use following GDB command before "tstart":

*(gdb) tvariable \$bt=1024*

Following part is an example about howto collect stack and howto use GDB parse it:

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_readdir
Tracepoint 1 at 0xffffffff8118c300: file
/home/teawater/kernel2/linux/fs/readdir.c, line 24.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $bt
>end
(gdb) tstart
(gdb) shell ls
1  crypto  fs      include kernel  mm
Module.symvers security System.map vmlinux
arch  drivers  hotcode.html  init  lib  modules.builtin  net
```



```

sound    usr      vmlinux.o
block    firmware hotcode.html~ ipc    Makefile modules.order
scripts  source    virt
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_readdir (file=0xffff8800c5556d00, filler=0xffffffff8118c4b0
<filldir>, buf=0xffff880108709f40)
    at /home/teawater/kernel2/linux/fs/readdir.c:24
24      {
(gdb) bt
#0  vfs_readdir (file=0xffff8800c5556d00, filler=0xffffffff8118c4b0
<filldir>, buf=0xffff880108709f40)
    at /home/teawater/kernel2/linux/fs/readdir.c:24
#1  0xffffffff8118c689 in sys_getdents (fd=<optimized out>,
dirent=0x1398c58, count=32768) at
/home/teawater/kernel2/linux/fs/readdir.c:214
#2  <signal handler called>
#3  0x00007f00253848a5 in ?? ()
#4  0x00003efd32cddfc9 in ?? ()
#5  0x00002c15b7d04101 in ?? ()
#6  0x000019c0c5704bf1 in ?? ()
#7  0x0000000900000000 in ?? ()
#8  0x000009988cc8d269 in ?? ()
#9  0x000009988cc9b8d1 in ?? ()
#10 0x0000000000000000 in ?? ()
(gdb) up
#1  0xffffffff8118c689 in sys_getdents (fd=<optimized out>,
dirent=0x1398c58, count=32768) at
/home/teawater/kernel2/linux/fs/readdir.c:214
214          error = vfs_readdir(file, filldir, &buf);
(gdb) p buf
$1 = {current_dir = 0x1398c58, previous = 0x0, count = 32768,
error = 0}
(gdb) p error
$3 = -9
(gdb) frame 0
#0  vfs_readdir (file=0xffff8800c5556d00, filler=0xffffffff8118c4b0
<filldir>, buf=0xffff880108709f40)
    at /home/teawater/kernel2/linux/fs/readdir.c:24
24      {

```

From this example, we can see some GDB commands that parse the the call stack:

- **bt** is the alias of GDB commands backtrace that print a backtrace of the entire stack: one line per frame for all frames in the stack.
- **up n** is move n frames up the stack. For positive numbers n, this

advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. `n` defaults to one.

- **down `n`** is move `n` frames down the stack. For positive numbers `n`, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. `n` defaults to one. You may abbreviate down as `do`.
- **frame `n`** is select frame number `n`. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is the one for main.

You can see that when you use `up`, `down` or `frame` to the different call stack frame, you can output the value of the arguments and local variables of different call stack frame.

To get the more info about howto use GDB parse the call stack, please see <http://sourceware.org/gdb/current/onlinedocs/gdb/Stack.html>

## Collect stack of current function's caller with `$_ret`

If you just want to collect stack of current function's caller, please use `$_ret`.

Please **note** that set the tracepoint that collect `$_ret` cannot in the first address of function.

For example:

```
(gdb) list vfs_read
360     }
361
362     EXPORT_SYMBOL(do_sync_read);
363
364     ssize_t vfs_read(struct file *file, char __user *buf, size_t count,
loff_t *pos)
365     {
366         ssize_t ret;
367
368         if (!(file->f_mode & FMODE_READ))
369             return -EBADF;
(gdb) trace 368
Tracepoint 2 at 0xffffffff8117a244: file
/home/teawater/kernel2/linux/fs/read_write.c, line 368.
(gdb) actions
Enter actions for tracepoint 2, one per line.
End with a line saying just "end".
>collect $_ret
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 2
#0  vfs_read (file=0xffff880141c46000, buf=0x359bda0 <Address
0x359bda0 out of bounds>, count=8192, pos=0xffff88012fa49f48)
    at /home/teawater/kernel2/linux/fs/read_write.c:368
368         if (!(file->f_mode & FMODE_READ))
(gdb) bt
#0  vfs_read (file=0xffff880141c46000, buf=0x359bda0 <Address
0x359bda0 out of bounds>, count=8192, pos=0xffff88012fa49f48)
    at /home/teawater/kernel2/linux/fs/read_write.c:368
#1  0xffffffff8117a3ea in sys_read (fd=<optimized out>,
buf=<unavailable>, count=<unavailable>)
    at /home/teawater/kernel2/linux/fs/read_write.c:469
Backtrace stopped: not enough registers or memory available to
unwind further
(gdb) up
```

```
#1 0xffffffff8117a3ea in sys_read (fd=<optimized out>,
buf=<unavailable>, count=<unavailable>)
    at /home/teawater/kernel2/linux/fs/read_write.c:469
469          ret = vfs_read(file, buf, count, &pos);
(gdb) p ret
$2 = -9
```

You see that the caller of function `vfs_read` is `sys_read`. And the local variable `ret` of `sys_read` is -9.

## Use \$dump\_stack to output stack dump through printk

Because this way need parse the stack when tracing and call printk inside, so it will be slow, unsafe, unclear and cannot access a lot of info of call stack. So I suggest you use the prev way to do stack dump.

KGTP has special trace state variable \$dump\_stack, "collect" it will let Linux Kernel output stack dump through printk.

Following example lets Linux Kernel show the stack dump of vfs\_readdir:

```
target remote /sys/kernel/debug/gtp
trace vfs_readdir
commands
    collect $dump_stack
end
```

Then your kernel will printk like:

```
[22779.208064] gtp 1:Pid: 441, comm: python Not tainted 2.6.39-rc3+ #46
[22779.208068] Call Trace:
[22779.208072] [<fe653cca>] gtp_get_var+0x4a/0xa0 [gtp]
[22779.208076] [<fe653d79>] gtp_collect_var+0x59/0xa0 [gtp]
[22779.208080] [<fe655974>] gtp_action_x+0x1bb4/0x1dc0 [gtp]
[22779.208084] [<c05b6408>] ? _raw_spin_unlock+0x18/0x40
[22779.208088] [<c023f152>] ? __find_get_block_slow+0xd2/0x160
[22779.208091] [<c01a8c56>] ? delayacct_end+0x96/0xb0
[22779.208100] [<c023f404>] ? __find_get_block+0x84/0x1d0
[22779.208103] [<c05b6408>] ? _raw_spin_unlock+0x18/0x40
[22779.208106] [<c02e0838>] ? find_revoke_record+0xa8/0xc0
[22779.208109] [<c02e0c45>] ?
jbd2_journal_cancel_revoke+0xd5/0xe0
[22779.208112] [<c02db51f>] ?
__jbd2_journal_temp_unlink_buffer+0x2f/0x110
[22779.208115] [<fe655c4c>] gtp_kp_pre_handler+0xcc/0x1c0
[gtp]
[22779.208118] [<c05b8a88>]
kprobe_exceptions_notify+0x3d8/0x440
[22779.208121] [<c05b7d54>] ?
hw_breakpoint_exceptions_notify+0x14/0x180
[22779.208124] [<c05b95eb>] ? sub_preempt_count+0x7b/0xb0
[22779.208126] [<c0227ac5>] ? vfs_readdir+0x15/0xb0
[22779.208128] [<c0227ac4>] ? vfs_readdir+0x14/0xb0
[22779.208131] [<c05b9743>] notifier_call_chain+0x43/0x60
[22779.208134] [<c05b9798>]
__atomic_notifier_call_chain+0x38/0x50
[22779.208137] [<c05b97cf>] atomic_notifier_call_chain+0x1f/0x30
```

[22779.208140] [<c05b980d>] notify\_die+0x2d/0x30  
[22779.208142] [<c05b71c5>] do\_int3+0x35/0xa0

# Howto let tracepoint output value directly

In the previous parts, you may understand that to get a value from Linux kernel, you need to use a tracepoint "collect" action to save the value to the tracepoint frame and use the GDB command "tfind" to parse the value from the frame data.

But we want get the value directly sometimes, so KGTP supports two ways to output values directly.

## Switch collect to output the value directly

KGTP has special trace state variables \$printk\_level, \$printk\_format and \$printk\_tmp to support this function.

\$printk\_level: if its value is 8 (this is the default value), "collect" action will save value to the tracepoint frame in the simple behavior.

If its value is 0-7, "collect" will output the value through "printk" directly, and value will be the level of printk. The level is:

0	<i>KERN_EMERG</i>	<i>system is unusable</i>
1	<i>KERN_ALERT</i>	<i>action must be taken immediately</i>
2	<i>KERN_CRIT</i>	<i>critical conditions</i>
3	<i>KERN_ERR</i>	<i>error conditions</i>
4	<i>KERN_WARNING</i>	<i>warning conditions</i>
5	<i>KERN_NOTICE</i>	<i>normal but significant condition</i>
6	<i>KERN_INFO</i>	<i>informational</i>
7	<i>KERN_DEBUG</i>	<i>debug-level messages</i>

\$printk\_format, collect printk will output value in the format that is set by it. The format is:

0	<i>This is the default value.</i> <i>If the size of collect value is 1, 2, 4 or 8, it will be output as an unsigned decimal.</i> <i>If not, it will be output as a hexadecimal string.</i>
1	<i>Output value in signed decimal.</i>
2	<i>Output value in unsigned decimal.</i>
3	<i>Output value in unsigned hexadecimal.</i>
4	<i>Output value as a string.</i>
5	<i>Output value as a hexadecimal string.</i>

\$printk\_tmp, to output the value of global variable need set to it first.

Following example shows a count number, pid, jiffies\_64 and the file name that

call `vfs_readdir`:

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) tvariable $c
(gdb) trace vfs_readdir
(gdb) actions
>teval $printk_level=0
>collect $c=$c+1
>collect ((struct task_struct *)$current_task)->pid
>collect $printk_tmp=jiffies_64
>teval $printk_format=4
>collect file->f_path.dentry->d_iname
>end
```

Then your kernel will printk like:

```
gtp 1:$c=$c+1=41
gtp 1:((struct task_struct *)$current_task)->pid=12085
gtp 1:$printk_tmp=jiffies_64=4322021438
gtp 1:file->f_path.dentry->d_iname=b26
gtp 1:$c=$c+1=42
gtp 1:((struct task_struct *)$current_task)->pid=12085
gtp 1:$printk_tmp=jiffies_64=4322021438
gtp 1:file->f_path.dentry->d_iname=b26
```

"gtp 1" means that it was output by tracepoint 1.



# Howto use watch tracepoint control hardware breakpoints to record memory access

**Watch tracepoint** can control hardware breakpoints to record the memory access through set some special trace state variables in its action.

Please **note** that watch tracepoint is just support by X86 and X86\_64 now. And dynamic watch tracepoint just can work OK in Linux 2.6.27 and newer version because Linux 2.6.26 and older version have some IPI issues on smp support.

## Trace state variables of watch tracepoint

<b>Name</b>	<b>Written by normal tracepoint</b>	<b>Read by normal tracepoint</b>	<b>Written by static watch tracepoint</b>	<b>Read by static watch tracepoint</b>	<b>Written by dynamic watch tracepoint</b>	<b>Read by dynamic watch tracepoint</b>
\$watch_static	Not support	Not support	If "teval \$watch_static=1", then this tracepoint is static watch tracepoint.	Not support	If "teval \$watch_static=1", then this tracepoint is static watch tracepoint.	Not support
\$watch_set_id	When this tracepoint want to setup a dynamic watch tracepoint, set a id of a dynamic watch tracepoint to \$watch_set_id to point out which dynamic watch tracepoint you wan to setup.	Not support	Not support	Not support	Not support	Not support
\$watch_set_addr	When this tracepoint want to setup a dynamic watch tracepoint, set the address of a dynamic watch tracepoint to \$watch_set_addr to point out which dynamic watch tracepoint you wan to setup.	Not support	Not support	Not support	Not support	Not support
\$watch_type	When this tracepoint want to setup a dynamic watch tracepoint, set the watch type of this dynamic watch tracepoint to \$watch_type. 0 is exec. 1 is write. 2 is read or write.	Get the value that this tracepoint set to \$watch_type	Set the type of this watch tracepoint.	Get the type of this watch tracepoint.	Set the default type of this watch tracepoint.	Get the type of this watch tracepoint when it really exec.
\$watch_size	When this tracepoint want to setup a dynamic watch tracepoint, set the watch size of this dynamic watch tracepoint to \$watch_size. The size should be 1, 2, 4, 8.	Get the value that this tracepoint set to \$watch_size.	Set the size of this watch tracepoint.	Get the size of this watch tracepoint.	Set the default size of this watch tracepoint.	Get the size of this watch tracepoint when it really exec.
\$watch_start	Set the address to a dynamic watch tracepoint(set by \$watch_set_addr or \$watch_set_id) and let it try to start	Get the result of this start. (It will fail becasue X86 just have 4 hardware	Not support	Not support	Not support	Not support

	work.	breakpoints.) Get 0 if success. If < 0 is the error id.				
\$watch_stop	Set a address to \$watch_stop will let a dynamic watch tracepoint that watch in this address stop.	Get the result of this stop.	Not support	Not support	Not support	Not support
\$watch_trace_num	Not support	Not support	Not support	Not support	Not support	The tracepoint number that setup this dynamic watch tracepoint.
\$watch_trace_addr	Not support	Not support	Not support	Not support	Not support	The tracepoint address that setup this dynamic watch tracepoint.
\$watch_addr	Not support	Not support	Not support	The address that this watch tracepoint is watching.	Not support	The address that this watch tracepoint is watching.
\$watch_val	Not support	Not support	Not support	The current value of the memory that this watch tracepoint is watching.	Not support	The current value of the memory that this watch tracepoint is watching.
\$watch_prev_val	Not support	Not support	Not support	The previous value of the memory that this watch tracepoint is watching.	Not support	The previous value of the memory that this watch tracepoint is watching.
\$watch_count	Not support	Not support	Not support	Not support	Not support	A special count for this watch tracepoint session.

## Static watch tracepoint

You can use static watch tracepoint when you want watch value of a global variable or some memory that you can get its address directly. Following example is watch jiffies\_64's write:

```
#Static watch tracepoint get watch address from tracepoint address.  
trace *&jiffies_64  
actions  
  #Set this watch tracepoint to static  
  teval $watch_static=1  
  #Watch memory write  
  teval $watch_type=1  
  teval $watch_size=8  
  collect $watch_val  
  collect $watch_prev_val  
  collect $bt  
end
```

## Dynamic watch tracepoint

If you want to watch value of a local variable or some memory that you just get its address inside the function, you can use dynamic watch tracepoint. Following example is watch write of `f->f_pos` and `f->f_op` inside function `get_empty_filp`:

```
trace *1
commands
  teval $watch_static=0
  teval $watch_type=1
  teval $watch_size=8
  collect $bt
  collect $watch_addr
  collect $watch_val
  collect $watch_prev_val
end
```

Define a dynamic watch tracepoint. The address "1" of it is not the address of memory that it will watch. It just help tracepoint that setup this dynamic watch tracepoint can find it.

```
list get_empty_filp
trace 133
commands
  teval $watch_set_addr=1
  teval $watch_size=4
  teval $watch_start=&(f->f_pos)
  teval $watch_size=8
  teval $watch_start=&(f->f_op)
end
```

Define a normal tracepoint that start to watch `f->f_pos` and `f->f_op` inside function `get_empty_filp`.

```
trace file_sb_list_del
commands
  teval $watch_stop=&(file->f_pos)
  teval $watch_stop=&(file->f_op)
end
```

Define a normal tracepoint that stop the tracepoint that watch `file->f_pos` and `file->f_op`.

# Use while-stepping let Linux kernel do single step

Please **note** that while-stepping is just support by X86 and X86\_64 now.

Video about howto use while-stepping <http://www.codepark.us/a/13>.

## Howto use while-stepping

while-stepping is a special tracepoint action that include some actions with it.

When tracepoints that its actions include "while-stepping n" execute, it will do n times single steps and executes the actions of while-stepping. For example:

```
trace vfs_read
#Because single step will make system slow, so use passcount or
condition to limit the execution times of tracepoint is better.
passcount 1
commands
collect $bt
collect $step_count
#do 2000 times single steps.
while-stepping 2000
#Following part is actions of "while-stepping 2000".
#Because step maybe execute to other functions, so does not
access local variables is better.
collect $bt
collect $step_count
end
end
```

Please **note** that tracepoint will disable the interrupt of current CPU when it do single step. Access **\$step\_count** in actions will get the count of this step that begin with 1.

## Read the traceframe of while-stepping

The data of different step that is recorded by while-stepping actions will be saved in different traceframe that you can use `tfind` ([Use `tfind` select the entry inside the trace frame info](#)) to select them.

Or you can switch KGTP to replay mode to select all the traceframe of a while-stepping tracepoint with GDB execution and reverse-execution commands. For example:

Use `tfind` select one the traceframe of a while-stepping tracepoint.

```
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_read (file=0xffff8801f7bd4c00, buf=0x7fff74e4edb0
<Address 0x7fff74e4edb0 out of bounds>, count=16,
    pos=0xffff8801f4b45f48) at /build/builddd/linux-
3.2.0/fs/read_write.c:365
365  {
```

Following commands will switch KGTP to replay mode.

```
(gdb) monitor replay
(gdb) tfind -1
No longer looking at any trace frame
#0  vfs_read (file=0xffff8801f7bd4c00, buf=0x7fff74e4edb0
<Address 0x7fff74e4edb0 out of bounds>, count=16,
    pos=0xffff8801f4b45f48) at /build/builddd/linux-
3.2.0/fs/read_write.c:365
365  {
```

Then you can use execution commands.

```
(gdb) n
368      if (!(file->f_mode & FMODE_READ))
(gdb) p file->f_mode
$5 = 3
```

Set breakpoints (Just valid in replay mode, will not affect Linux kernel execution).

```
(gdb) b 375
Breakpoint 2 at 0xffffffff81179b75: file /build/builddd/linux-
3.2.0/fs/read_write.c, line 375.
(gdb) c
Continuing.
```

```
Breakpoint 2, vfs_read (file=0xffff8801f7bd4c00,
buf=0x7fff74e4edb0 <Address 0x7fff74e4edb0 out of bounds>,
count=16,
    pos=0xffff8801f4b45f48) at /build/builddd/linux-
3.2.0/fs/read_write.c:375
```

```

375         ret = rw_verify_area(READ, file, pos, count);
(gdb) s
rw_verify_area (read_write=0, file=0xffff8801f7bd4c00,
ppos=0xffff8801f4b45f48, count=16)
    at /build/buildd/linux-3.2.0/fs/read_write.c:300
300         inode = file->f_path.dentry->d_inode;

```

Use reverse-execution commands.

```
(gdb) rs
```

```

Breakpoint 2, vfs_read (file=0xffff8801f7bd4c00,
buf=0x7fff74e4edb0 <Address 0x7fff74e4edb0 out of bounds>,
count=16,
    pos=0xffff8801f4b45f48) at /build/buildd/linux-
3.2.0/fs/read_write.c:375
375         ret = rw_verify_area(READ, file, pos, count);
(gdb) rn
372         if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))

```

GDB commands tstart, tfind or quit can auto close the replay mode.



# Howto show a variable whose value has been optimized away

Sometimes, GDB will output some value like:

*inode has been optimized out of existence.*

*res has been optimized out of existence.*

That is because value of inode and res is optimized. Linux Kernel is built with -O2 so you will get this trouble sometimes.

There are 2 ways to handle it:

## Update your GCC

The VTA branch [http://gcc.gnu.org/wiki/Var\\_Tracking\\_Assignments](http://gcc.gnu.org/wiki/Var_Tracking_Assignments) was merged for GCC 4.5. This helps a lot with generating dwarf for previously "optimized out" values.

## Get the way that access the variable that has been out through parse ASM code

Even if update the GCC to the newer version, you will still meet the issue. The main reason is the data is inside the registers but GCC doesn't put it to debug info. Then GDB just can output this variable has been optimized away.

But you can get where is the variable from ASM code and access it inside the tracepoint actions.

Following is a example that find variable "f" of function get\_empty\_filp and use it in tracepoint actions:

We want collect the value of "f" but looks it has been optimized away.

```
(gdb) list get_empty_filp
...
...
...
137      INIT_LIST_HEAD(&f->f_u.fu_list);
138      atomic_long_set(&f->f_count, 1);
139      rwlock_init(&f->f_owner.lock);
140      spin_lock_init(&f->f_lock);
141      eventpoll_init_file(f);
(gdb)
142      /* f->f_version: 0 */
143      return f;
(gdb) trace 143
Tracepoint 1 at 0xffffffff8119b30e: file fs/file_table.c, line 143.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect f
`f` is optimized away and cannot be collected.
```

Now use "disassemble /m" command get the ASM code and source line that have relation with "f" and parse them.

```
(gdb) disassemble /m get_empty_filp
...
...
...
125      f = kmem_cache_zalloc(filp_cachep, GFP_KERNEL);
126      if (unlikely(!f))
    0xffffffff8119b28c <+92>: test %rax,%rax
    0xffffffff8119b292 <+98>: je 0xffffffff8119b362
    <get_empty_filp+306>

127      return ERR_PTR(-ENOMEM);
```

```

0xffffffff8119b362 <+306>: mov    $0xffffffffffff4,%rax
0xffffffff8119b369 <+313>: jmp    0xffffffff8119b311
<get_empty_filp+225>

```

Code from "+98" to "+132" is not show in this part because they belong to other inline function. But you can get them with GDB command "disassemble get\_empty\_filp".

```

0xffffffff8119b287 <+87>: callq 0xffffffff81181cb0
<kmem_cache_alloc>
0xffffffff8119b28c <+92>: test   %rax,%rax
0xffffffff8119b28f <+95>: mov    %rax,%rbx
0xffffffff8119b292 <+98>: je     0xffffffff8119b362
<get_empty_filp+306>
0xffffffff8119b298 <+104>: mov    0xb4d406(%rip),%edx    #
0xffffffff81ce86a4 <percpu_counter_batch>
0xffffffff8119b29e <+110>: mov    $0x1,%esi
0xffffffff8119b2a3 <+115>: mov    $0xffffffff81c05340,%rdi
---Type <return> to continue, or q <return> to quit---
0xffffffff8119b2aa <+122>: callq 0xffffffff8130dd20
<__percpu_counter_add>

```

According to the ASM code you can see that return value of kmem\_cache\_alloc is inside \$rax and its value is set to \$rbx.

Looks \$rbx has the value of "f". Let's check other ASM code.

```

128
129         percpu_counter_inc(&nr_files);
130         f->f_cred = get_cred(cred);
0xffffffff8119b2b4 <+132>: mov    %r12,0x70(%rbx)

```

Set a value to element of f, the ASM code is set value of \$r12 to a address that base address is \$rbx. It also looks like \$rbx is "f".

```

131         error = security_file_alloc(f);
0xffffffff8119b2b8 <+136>: mov    %rbx,%rdi
0xffffffff8119b2bb <+139>: callq 0xffffffff8128ee30
<security_file_alloc>

132         if (unlikely(error)) {
0xffffffff8119b2c0 <+144>: test   %eax,%eax
0xffffffff8119b2c2 <+146>: jne    0xffffffff8119b36b
<get_empty_filp+315>
---Type <return> to continue, or q <return> to quit---

133         file_free(f);
134         return ERR_PTR(error);

```

```

0xffffffff8119b393 <+355>: movslq -0x14(%rbp),%rax
0xffffffff8119b397 <+359>: jmpq 0xffffffff8119b311
<get_empty_filp+225>

135      }
136
137      INIT_LIST_HEAD(&f->f_u.fu_list);
138      atomic_long_set(&f->f_count, 1);
139      rwlock_init(&f->f_owner.lock);
0xffffffff8119b2e4 <+180>: movl $0x100000,0x50(%rbx)

140      spin_lock_init(&f->f_lock);
0xffffffff8119b2c8 <+152>: xor  %eax,%eax
0xffffffff8119b2d1 <+161>: mov  %ax,0x30(%rbx)

141      eventpoll_init file(f);
142      /* f->f_version: 0 */
143      return f;
0xffffffff8119b30e <+222>: mov  %rbx,%rax

```

And after check other ASM code. You can make sure that \$rbx is "f".

Then you can access "f" through access \$rbx in tracepoint actions, for example:

```

(gdb) trace 143
Tracepoint 1 at 0xffffffff8119b30e: file fs/file_table.c, line 143.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
#collect f
>collect $rbx
#collect *f
>collect *((struct file *)$rbx)
#collect f->f_op
>collect ((struct file *)$rbx)->f_op
>end

```

# How to get the function pointer point to

## If the debug info of the function pointer is not optimized out

You can collect it directly and print what it point to. For example:

```
377          count = ret;
378          if (file->f_op->read)
379              ret = file->f_op->read(file, buf, count, pos);
(gdb)
(gdb) trace 379
Tracepoint 1 at 0xffffffff81173ba5: file
/home/teawater/kernel/linux/fs/read_write.c, line 379.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect file->f_op->read
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind
(gdb) p file->f_op->read
$5 = (ssize_t (*)(struct file *, char *, size_t, loff_t *))
0xffffffff81173190 <do_sync_read>
#Then you know file->f_op->read point to do_sync_read.
```

## If the debug info of the function pointer is optimized out

You can use tracepoint step to handle it. For example:

```
#Find out which instruction that it is called.
(gdb) disassemble /rm vfs_read
379          ret = file->f_op->read(file, buf, count, pos);
0xffffffff81173ba5 <+181>: 48 89 da    mov    %rbx,%rdx
0xffffffff81173ba8 <+184>: 4c 89 e9    mov    %r13,%rcx
0xffffffff81173bab <+187>: 4c 89 e6    mov    %r12,%rsi
0xffffffff81173bae <+190>: 4c 89 f7    mov    %r14,%rdi
0xffffffff81173bb1 <+193>: ff d0      callq  *%rax
0xffffffff81173bb3 <+195>: 48 89 c3    mov    %rax,%rbx
(gdb) trace *0xffffffff81173bb1
Tracepoint 1 at 0xffffffff81173bb1: file
/home/teawater/kernel/linux/fs/read_write.c, line 379.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>while-stepping 1
>collect $reg
>end
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind
#0  tty_read (file=0xffff88006ca74900, buf=0xb6b7dc <Address
0xb6b7dc out of bounds>, count=8176,
      ppos=0xffff88006e197f48) at
/home/teawater/kernel/linux/drivers/tty/tty_io.c:960
960  {
#Then you know file->f_op->read point to tty_read.
```

Please **note** that while-stepping will make tracepoint cannot use kprobes-optimization.

# **/sys/kernel/debug/gtpframe and offline debug**

/sys/kernel/debug/gtpframe supplies trace frame in tfile format (GDB can parse it) when KGTP is stop.

In the PC that can run the GDB:

Change the "target remote XXXX" to

```
(gdb) target remote | perl ./getgtpersp.pl
```

After that, set tracepoint and start it as usual:

```
(gdb) trace vfs_readdir  
Tracepoint 1 at 0xffffffff8114f3c0: file /home/teawater/kernel/linux-  
2.6/fs/readdir.c, line 24.  
(gdb) actions  
Enter actions for tracepoint 1, one per line.  
End with a line saying just "end".  
#If your GDB support tracepoint "printf" (see "Howto use tracepoint  
printf"), use it to show the value directly is better.  
>collect $reg  
>end  
(gdb) tstart  
(gdb) stop  
(gdb) quit
```

Then you can find files gtpstart and gtpstop in current directory. Copy it to the machine that you want to debug.

In the debugged machine, copy the program "putgtpersp" and "gtp.ko" in the KGTP directory to this machine first. After insmod the gtp.ko:

Start the tracepoint:

```
./putgtpersp ./gtpstart
```

Stop the tracepoint:

```
./putgtpersp ./gtpstop
```

You can let Linux Kernel show the value directly, please see [Howto let tracepoint output value directly](#).

If you want to save the value to the trace frame and parse later, you can use file "/sys/kernel/debug/gtpframe" that has the trace frame. Copy it to the PC

that has GDB.

Please **note** that some "cp" cannot handle it very well, please use "cat /sys/kernel/debug/gtpframe > ./gtpframe" to copy it.

In the PC that can run the GDB:

```
(gdb) target tfile ./gtpframe
Tracepoint 1 at 0xffffffff8114f3dc: file /home/teawater/kernel/linux-2.6/fs/readdir.c, line 24.
Created tracepoint 1 for target's tracepoint 1 at 0xffffffff8114f3c0.
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_readdir (file=0xffff880036e8f300, filler=0xffffffff8114f240
<filldir>, buf=0xffff880001e5bf38)
    at /home/teawater/kernel/linux-2.6/fs/readdir.c:24
24    {
```

Please **note** that if you want connect KGTP from GDB in remote machine after use offline debug, you need "rmmod gtp" and "insmod gtp.ko" before call "nc".



## How to use

### **/sys/kernel/debug/gtpframe\_pipe**

This interface supplies same format trace frame with "gtpframe". But it can work when KGTP is running. After data is read, it will auto deleted from trace frame like "trace\_pipe" of ftrace.

## Get the frame info with GDB

```
#connect to the interface  
(gdb) target tfile /sys/kernel/debug/gtpframe_pipe  
#Get one trace frame entry  
(gdb) tfind 0  
Found trace frame 0, tracepoint 1  
#Get the next one  
(gdb) tfind  
Target failed to find requested trace frame.  
(gdb) tfind 0  
Found trace frame 0, tracepoint 1
```

This way is better to work with python to parse Kernel. add-ons/hotcode.py is an example of python script.

## Get the frame info with cat

*sudo cat /sys/kernel/debug/gtpframe\_pipe > g*

Then all the trace frame will be saved in file "g".

## Get the frame info with getframe

KGTP package include a program "getframe" can help you save the trace frame to files.

Following part is the help of it:

*getframe -h*

*Get the trace frame of KGTP and save them in current directory with tfile format.*

*Usage: ./getframe [option]*

- g n    Set the minimum free size limit to n G.  
When free size of current disk is smaller than n G,  
./getframe will exit (-q) or wait some seconds (-w).  
The default value of it is 2 G.*
- q      Quit when current disk is smaller than  
minimum free size limit (-g).*
- w n    Wait n seconds when current disk is smaller  
than minimum free size limit (-g).*
- e n    Set the entry number of each tfile to n.  
The default value of it is 1000.*
- h      Display this information.*

## Use \$pipe\_trace

For the lock safe, KGTP will ignore the task that read the /sys/kernel/debug/gtpframe\_pipe in default.

If you really need trace this task, and be sure that is safe. You can use following command before call "tstart":

*(gdb) tvariable \$pipe\_trace=1*

Then KGTP will not ignore the task that read /sys/kernel/debug/gtpframe\_pipe.

# Use KGTP with user applications

KGTP can access the memory and trace user applications without stop it.

## Let GDB connect KGTP for user applications

- 1) Open GDB without load any user applications.
- 2) If user applications is running on the current machine, use GDB command "target extended-remote /sys/kernel/debug/gtp" connect to KGTP. If user applications is running on the remote machine, use netcat like [GDB on remote machine](#) but replace "target remote" to "target extended-remote".
- 3) Load user applications (it must be built with GCC option "-g" to make it has debug information) with GDB command "file".
- 4) Use GDB command "attach pid" to attach the task's pid.

Then let GDB connect KGTP for user applications should be:

```
sudo gdb-release
(gdb) target extended-remote /sys/kernel/debug/gtp
Remote debugging using /sys/kernel/debug/gtp
0x00000000 in ?? ()
(gdb) file a.out
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from /home/teawater/kernel/kgtp/a.out...done.
(gdb) attach 15412
A program is being debugged already. Kill it? (y or n) y
Attaching to program: /home/teawater/kernel/kgtp/a.out, Remote
target
# Some version of GDB will output internal-error, please answer "n"
to ignore it.
```

## Read memory of user applications directly

After GDB attach the user applications success, you can access the memory of this task with GDB commands "p" and "x". You can get help of these commands with GDB commands "help p" and "help x". For example:

```
(gdb) p c
$19 = 4460
(gdb) p &c
$21 = (int *) 0x601048 <c>
(gdb) x 0x601048
0x601048 <c>: 0x00001181
```

## Trace user applications

KGTP use **uprobes** function of Linux kernel trace user applications, just Linux kernel 3.9 and later version support this function.

The build config of most Linux distributions's Linux kernel (3.9 and later) has opened **uprobes**.

For the Linux kernel that built by yourself:

```
Kernel hacking --->
  [*] Tracers --->
    [*] Enable uprobes-based dynamic events
```

If current Linux kernel **uprobes** is opened, you can set tracepoint according to [GDB tracepoint](#) after GDB attach the user applications success. For example:

```
(gdb) trace 14
Tracepoint 1 at 0x400662: file /home/teawater/kernel/kgtp-
misc/test.c, line 14.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $bt
>collect c
>end
(gdb) tstart
(gdb) tstatus
Trace is running on the target.
Collected 5 trace frames.
Trace buffer has 20824428 bytes of 20828160 bytes free (0% full).
Trace will stop if GDB disconnects.
Not looking at any trace frame.
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  main (argc=1, argv=0x7fff5e878368, envp=0x7fff5e878378)
    at /home/teawater/kernel/kgtp-misc/test.c:14
    14          c += 1;
(gdb) bt
#0  main (argc=1, argv=0x7fff5e878368, envp=0x7fff5e878378)
    at /home/teawater/kernel/kgtp-misc/test.c:14
(gdb) p c
$7 = 36
```

Please **note** that even if you just attach one of these tasks, user applications's tracepoint will be triggered by all tasks of a user application. (I think this is a very interesting feature of **uprobes**, so I didn't limit it in KGTP tracepoint.)



You can add `$current_task_pid` check to conditions of tracepoint to make tracepoint just be triggered by one of this task. Following example is set a tracepoint that just for task 985:

```
(gdb) trace 14
Tracepoint 1 at 0x400662: file /home/teawater/kernel/kgtp-
misc/test.c, line 14.
(gdb) condition $bpnum ($current_task_pid == 985)
```

And you can "collect `$current_task_pid`" in tracepoint actions to make sure which task triggers the tracepoint. For example:

```
(gdb) trace 14
Tracepoint 2 at 0x400662: file /home/teawater/kernel/kgtp-
misc/test.c, line 14.
(gdb) actions
Enter actions for tracepoint 2, one per line.
End with a line saying just "end".
>collect $current_task_pid
>collect c
>end
(gdb) tstart
(gdb) tstatus
Trace is running on the target.
Collected 6 trace frames.
Trace buffer has 20827776 bytes of 20828160 bytes free (0% full).
Trace will stop if GDB disconnects.
Not looking at any trace frame.
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 2
#0 main (argc=<unavailable>, argv=<unavailable>,
envp=<unavailable>) at /home/teawater/kernel/kgtp-misc/test.c:14
14             c += 1;
(gdb) p $current_task_pid
$2 = 9983
(gdb) tfind
Found trace frame 1, tracepoint 2
14             c += 1;
(gdb) p $current_task_pid
$3 = 9982
(gdb)
```

## collect stack (for backtrace) of system from Linux kernel to user applications in tracepoint

**\$current** is a special trace state variable that if the action of an tracepoint access it, this tracepoint will access the values of the registers and the memory of current task instead of Linux kernel.

In general, the tracepoint will get the registers value of current task from **task\_pt\_regs**. Then collect **\$current** in tracepoint actions will let this tracepoint access values of current task. For example:

```
(gdb) actions  
Enter actions for tracepoint 1, one per line.  
End with a line saying just "end".  
>collect $current  
>collect $bt  
>end
```

In addition, for some special function that its arguments include the pointer to the registers(for example: do\_IRQ function of X86), tracepoint need get the registers from the arguments of fuction. Then set the pointer to **\$current** will let this tracepoint get it. For example:

```
(gdb) actions  
Enter actions for tracepoint 1, one per line.  
End with a line saying just "end".  
>teval $current=(uint64_t)regs  
>collect $bt  
>end
```

**\$current\_task\_user** is a special trace state variable that it is value will be true when current task is in user mode.

With these two trace state variables, you can use KGTP collect the stack(backtrace) of current task.

Following example show how we do backtrace(stack dump) from user space to Linux kernel:

```
#Connect to KGTP(same with prev section)  
(gdb) target extended-remote /sys/kernel/debug/gtp  
#Setup an tracepoint that collect the user space stack of task 18776.  
(gdb) trace vfs_read  
Tracepoint 1 at 0xffffffff8117a3d0: file  
/home/teawater/kernel/linux/fs/read_write.c, line 365.  
(gdb) condition 1 ($current_task_user && $current_task_pid ==  
18776)  
(gdb) actions  
Enter actions for tracepoint 1, one per line.
```

End with a line saying just "end".

```

>collect $current
>collect $bt
>end
#Setup a tracepoint that collect kernel space stack of task 18776.
(gdb) trace vfs_read
Note: breakpoint 1 also set at pc 0xffffffff8117a3d0.
Tracepoint 2 at 0xffffffff8117a3d0: file
/home/teawater/kernel/linux/fs/read_write.c, line 365.
(gdb) condition 2 ($current_task_user && $current_task_pid ==
18776)
(gdb) actions
Enter actions for tracepoint 2, one per line.
End with a line saying just "end".
>collect $bt
>end
(gdb) tstart
(gdb) tstop
#Following part is same with prev section, add a new inferior to
parse info of the user space program.
(gdb) add-inferior
Added inferior 2
(gdb) inferior 2
[Switching to inferior 2 [<null>] (<noexec>)]
(gdb) file gdb
Reading symbols from /usr/local/bin/gdb...done.
(gdb) attach 18776
#tracepoint 1 collect the user space stack.
(gdb) tfind
Found trace frame 0, tracepoint 1
#0 0x00007f77331d7d0f in __read_nocancel () from /lib/x86_64-
linux-gnu/libpthread.so.0
#This is the user space backtrace of task 18776.
(gdb) bt
#0 0x00007f77331d7d0f in __read_nocancel () from /lib/x86_64-
linux-gnu/libpthread.so.0
#1 0x000000000078e145 in rl_callback_read_char () at
../src/readline/callback.c:201
#2 0x000000000069de79 in rl_callback_read_char_wrapper
(client_data=<optimized out>) at ../src/gdb/event-top.c:169
#3 0x000000000069ccf8 in process_event () at ../src/gdb/event-
loop.c:401
#4 process_event () at ../src/gdb/event-loop.c:351
#5 0x000000000069d448 in gdb_do_one_event () at
../src/gdb/event-loop.c:465
#6 0x000000000069d5d5 in start_event_loop () at
../src/gdb/event-loop.c:490
#7 0x0000000000697083 in captured_command_loop

```

```

(data=<optimized out>) at ../../src/gdb/main.c:226
#8 0x0000000000695d8b in catch_errors (func=0x697070
<captured_command_loop>, func_args=0x0, errstring=0x14df99e
""
    mask=6) at ../../src/gdb/exceptions.c:546
#9 0x00000000006979e6 in captured_main (data=<optimized
out>) at ../../src/gdb/main.c:1001
#10 0x0000000000695d8b in catch_errors (func=0x697360
<captured_main>,
    func@entry=<error reading variable: PC not available>,
    func_args=0x7fff08afd5b0,
    func_args@entry=<error reading variable: PC not available>,
    errstring=<unavailable>,
    errstring@entry=<error reading variable: PC not available>,
    mask=<unavailable>,
    mask@entry=<error reading variable: PC not available>) at
../../src/gdb/exceptions.c:546
#11 <unavailable> in ?? ()
Backtrace stopped: not enough registers or memory available to
unwind further
#The tracepoint 2 collect the kernel space stack. So switch to
inferior 1 that load the kernel debug info.
(gdb) tfind
Found trace frame 1, tracepoint 2
#0 0xffffffff8117a3d0 in ?? ()
(gdb) inferior 1
[Switching to inferior 1 [Remote target]
(/home/teawater/kernel/b/vmlinux)]
[Switching to thread 1 (Remote target)]
#0 vfs_read (file=0xffff88021a559500, buf=0x7fff08afd31f
<Address 0x7fff08afd31f out of bounds>, count=1,
    pos=0xffff8800c47e1f48) at
/home/teawater/kernel/linux/fs/read_write.c:365
365 {
#This is the backtrace of kernel stack.
(gdb) bt
#0 vfs_read (file=0xffff88021a559500, buf=0x7fff08afd31f
<Address 0x7fff08afd31f out of bounds>, count=1,
    pos=0xffff8800c47e1f48) at
/home/teawater/kernel/linux/fs/read_write.c:365
#1 0xffffffff8117a59a in sys_read (fd=<optimized out>,
    buf=0x7fff08afd31f <Address 0x7fff08afd31f out of bounds>,
    count=1) at /home/teawater/kernel/linux/fs/read_write.c:469
#2 <signal handler called>
#3 0x00007f77331d7d10 in ?? ()
#4 0x0000000000000000 in ?? ()

```

## **How to use add-ons/hotcode.py**

This script can show the hottest code line in the Linux kernel or user space program through parse and record the pc address in the interrupt handler.

Please goto <http://code.google.com/p/kgtp/wiki/hotcode> see howto use it.

# How to add plugin in C

KGTP support plugin that write in C. The plugin will be built as LKM

## API

*#include "gtp.h"*

This header file include the API that plugin need.

*extern int gtp\_plugin\_mod\_register(struct module \*mod);*  
*extern int gtp\_plugin\_mod\_unregister(struct module \*mod);*

These two functions register and unregister the plugin module. Then when KGTP will add module usage count when it access the resource of the plugin module.

*extern struct gtp\_var \*gtp\_plugin\_var\_add(char \*name, int64\_t val,*  
*struct gtp\_var\_hooks \*hooks);*

This function add special trace state variable to the KGTP.

- **name** is the name of special trace state variable.
- **val** is initialization value of special trace state variable.
- **hooks** is the function pointers. The function pointers can be set to NULL if this function doesn't support.
- **Return** the gtp\_var pointer if success. Get error will return error code that IS\_ERR and PTR\_ERR can handle.

```
struct gtp_var_hooks {  
    int (*gdb_set_val)(struct gtp_trace_s *unused, struct gtp_var  
*var,  
                        int64_t val);  
    int (*gdb_get_val)(struct gtp_trace_s *unused, struct gtp_var  
*var,  
                        int64_t *val);  
    int (*agent_set_val)(struct gtp_trace_s *gts, struct gtp_var  
*var,  
                        int64_t val);  
    int (*agent_get_val)(struct gtp_trace_s *gts, struct gtp_var  
*var,  
                        int64_t *val);  
};
```

- **gdb\_set\_val** will be called when GDB set the value of TSV. Please note that TSV just can be set by GDB command "tvariable \$xxx=1" and the

value just be sent to KGTP when GDB command "tstart".

- **unused** is unused. Just to make this pointer can share function with agent\_set\_val.
- **var** is the pointer that point to the gtp\_var pointer. Then function of plugin can use it to figure out which TSV is accessed when TSVs share the function.
- **val** is the value that GDB set.
- **Return** return -1 if error. return 0 if success.
- **gdb\_get\_val** will be called when GDB get the value of TSV. Please note that TSV get is different with TSV set. It can be gotten from KGTP anytime. And get its value just like get the value of GDB internal value. For example: "p \$xxx".
  - **unused** is same with gdb\_set\_val.
  - **var** is same with gdb\_set\_val.
  - **val** is the pointer that use to return value.
  - **Return** is same with gdb\_set\_val.
- **agent\_set\_val** will be called when tracepoint action([teval expr1, expr2, ...](#)) set the value of TSV.
  - **gts** is pointer to the tracepoint session struct.
  - **var** is same with gdb\_set\_val.
  - **val** is the value the action set.
  - **Return** is same with gdb\_set\_val.
- **agent\_get\_val** will be called when tracepoint action([collect expr1, expr2, ...](#) or [teval expr1, expr2, ...](#)) get the value of TSV.
  - **gts** is same with agent\_set\_val.
  - **var** is same with gdb\_set\_val.
  - **val** is same with gdb\_get\_val.
  - **Return** is same with gdb\_set\_val.

*[extern int gtp\\_plugin\\_var\\_del\(struct gtp\\_var \\*var\);](#)*

When rmmod the plugin module, use this function remove the TSV that gtp\_plugin\_var\_add add.

## Example

plugin\_example.c that in the KGTP directory is the example for KGTP plugin. You can use "make P=1" build it. It add 4 TSV to KGTP.

- **\$test1** support nothing.
- **\$test2** support be get and set by GDB or tracepoint action.
- **\$test3** just support tracepoint action set. When set a value to it, it will look up a kernel symbol of this value and print it. For example "teval \$test3=(int64\_t)\$rip".
- **\$test4** just support tracepoint action set. When set a value to it, it will look up a kernel symbol of current tracepoint address and print it.



## How to use

- insmod KGTP module according to [Insmod the KGTP module](#).
- insmod plugin\_example.ko
- Use GDB connect to KGTP and use it.
- Disconnect GDB. If option in [Do not stop tracepoint when the GDB disconnects](#) set to on, set it to off.
- rmmod plugin\_example.ko

Please **note** that KGTP support add more than one plugin.

# How to use performance counters

Performance counters are special hardware registers available on most modern CPUs. These registers count the number of certain types of hw events: such as instructions executed, cachemisses suffered, or branches mis-predicted - without slowing down the kernel or applications. These registers can also trigger interrupts when a threshold number of events have passed - and can thus be used to profile the code that runs on that CPU.

The Linux Performance Counter subsystem called perf event can get the value of performance counter. You can access it through KGTP perf event trace state variables.

Please goto read the file tools/perf/design.txt in Linux Kernel to get more info about perf event.

## Define a perf event trace state variable

Access an performance counter need define following trace state variable:

*"pe\_cpu\_" + tv\_name      Define the the CPU id of the performance counter.*

*"pe\_type\_" + tv\_name      Define the the type of the performance counter.*

*"pe\_config\_" + tv\_name    Define the the config of the performance counter.*

*"pe\_en\_" + tv\_name        This the switch to enable or disable the performance counter.*

*The performance counter is disable in default.*

*"pe\_val\_" + tv\_name      Access this variable can get the value of the performance counter.*

## Define a per\_cpu perf event trace state variable

Define a per\_cpu perf event trace state variable is same with define [Per\\_cpu trace state variables](#).

*"p\_pe\_" + perf\_event type + string + CPU\_id*

Please **note** that if you define a per\_cpu perf event trace state variable, you will not need define the cpu id("pe\_cpu") because KGTP already get it.

## The perf event type and config

The type of perf event can be:

- 0 *PERF\_TYPE\_HARDWARE*
- 1 *PERF\_TYPE\_SOFTWARE*
- 2 *PERF\_TYPE\_TRACEPOINT*
- 3 *PERF\_TYPE\_HW\_CACHE*
- 4 *PERF\_TYPE\_RAW*
- 5 *PERF\_TYPE\_BREAKPOINT*

If the type is 0(*PERF\_TYPE\_HARDWARE*), the config can be:

- 0 *PERF\_COUNT\_HW\_CPU\_CYCLES*
- 1 *PERF\_COUNT\_HW\_INSTRUCTIONS*
- 2 *PERF\_COUNT\_HW\_CACHE\_REFERENCES*
- 3 *PERF\_COUNT\_HW\_CACHE\_MISSES*
- 4 *PERF\_COUNT\_HW\_BRANCH\_INSTRUCTIONS*
- 5 *PERF\_COUNT\_HW\_BRANCH\_MISSES*
- 6 *PERF\_COUNT\_HW\_BUS\_CYCLES*
- 7 *PERF\_COUNT\_HW\_STALLED\_CYCLES\_FRONTEND*
- 8 *PERF\_COUNT\_HW\_STALLED\_CYCLES\_BACKEND*

If the type is 3(*PERF\_TYPE\_HW\_CACHE*), the config need to divide to 3 parts:  
First one is cache id, it need be  $\ll 0$  before set to config:

- 0 *PERF\_COUNT\_HW\_CACHE\_L1D*
- 1 *PERF\_COUNT\_HW\_CACHE\_L1I*
- 2 *PERF\_COUNT\_HW\_CACHE\_LL*
- 3 *PERF\_COUNT\_HW\_CACHE\_DTLB*
- 4 *PERF\_COUNT\_HW\_CACHE\_ITLB*
- 5 *PERF\_COUNT\_HW\_CACHE\_BPU*

Second one is cache op id, it need be  $\ll 8$  before set to config:

- 0 *PERF\_COUNT\_HW\_CACHE\_OP\_READ*
- 1 *PERF\_COUNT\_HW\_CACHE\_OP\_WRITE*
- 2 *PERF\_COUNT\_HW\_CACHE\_OP\_PREFETCH*

Last one is cache op result id, it need be  $\ll 16$  before set to config:

- 0 *PERF\_COUNT\_HW\_CACHE\_RESULT\_ACCESS*
- 1 *PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS*

If you want get the perf count of *PERF\_COUNT\_HW\_CACHE\_L1I(1)*, *PERF\_COUNT\_HW\_CACHE\_OP\_WRITE(1)* and *PERF\_COUNT\_HW\_CACHE\_RESULT\_MISS(1)*, you can use:

*(gdb) tvariable \$pe\_config\_cache=1 | (1 << 8) | (1 << 16)*  
tools/perf/design.txt in Linux Kernel have more info about type and config of perf event.

## Enable and disable all the perf event in a CPU with \$p\_pe\_en

I think the best way that count a part of code with performance counters is enable all the count in the begin of the code and disable all of them in the end. You can do it with "pe\_en". But if you have a lot of perf event trace state variables. That will make the tracepoint action very big. \$p\_pe\_en is for this issue. You can enable all the perf event trace state variables in current CPU with following action:

```
>teval $p_pe_en=1
```

Disable them with set \$p\_pe\_en to 0.

```
>teval $p_pe_en=0
```

## GDB scripts to help with set and get the perf event trace state variables

Following is a GDB script define two commands dpe and spe to help define and show the perf event trace state variables.

You can put it to the ~/.gdbinit or your tracepoint script. Then you can use this two commands in GDB directly.

```
define dpe
  if ($argc < 2)
    printf "Usage: dpe pe_type pe_config [enable]\n"
  end
  if ($argc >= 2)
    eval "tvariable $p_pe_val_%d%d_c", $arg0, $arg1
    eval "tvariable $p_pe_en_%d%d_c", $arg0, $arg1
    set $tmp=0
    while $tmp<$cpu_number
      eval "tvariable $p_pe_type_%d%d_c%d=%d", $arg0, $arg1, $tmp,
      $arg0
      eval "tvariable $p_pe_config_%d%d_c%d=%d", $arg0, $arg1,
      $tmp, $arg1
      eval "tvariable $p_pe_val_%d%d_c%d=0", $arg0, $arg1, $tmp
      if ($argc >= 3)
        eval "tvariable $p_pe_en_%d%d_c%d=%d", $arg0, $arg1, $tmp,
        $arg2
      end
      set $tmp=$tmp+1
    end
  end
end

document dpe
Usage: dpe pe_type pe_config [enable]
end

define spe
  if ($argc != 2 && $argc != 3)
    printf "Usage: spe pe_type pe_config [cpu_id]\n"
  end
  if ($argc == 2)
    set $tmp=0
    while $tmp<$cpu_number
      eval "printf \"\$p_pe_val_%%d%%d_c%%d=%%ld\\n\", $arg0,
      $arg1, $tmp, $p_pe_val_%d%d_c%d", $arg0, $arg1, $tmp
      set $tmp=$tmp+1
    end
  end
end
```



```

end
if ($argc == 3)
    eval "printf \"\$p_pe_val_%%d%%d_c%%d=%%ld\\n\",$arg0, $arg1,
$tmp, $p_pe_val_%%d%%d_c%%d", $arg0, $arg1, $arg2
end
end

document spe
Usage: spe pe_type pe_config [cpu_id]
end

```

Following is an example to use it get the performance counters of function tcp\_v4\_rcv:

```

#Connect to KGTP
(gdb) target remote /sys/kernel/debug/gtp
#Define 3 pe tvs for PERF_COUNT_HW_CPU_CYCLES,
PERF_COUNT_HW_CACHE_MISSES and
PERF_COUNT_HW_BRANCH_MISSES.
(gdb) dpe 0 0
(gdb) dpe 0 3
(gdb) dpe 0 5
#enable the performance counters of this CPU in the begin of this
function.
(gdb) trace tcp_v4_rcv
(gdb) action
>teval $p_pe_en=1
>end
#$$kret make this hanler the end of function tcp_v4_rcv.
(gdb) trace *(tcp_v4_rcv)
(gdb) action
>teval $kret=0
#disable all performance counters of this CPU
>teval $p_pe_en=0
#Access the per cpu perf event tv will access to the current cpu pe
tv.
>collect $p_pe_val_00_0
>collect $p_pe_val_03_0
>collect $p_pe_val_05_0
#Set all the pe tv to 0
>teval $p_pe_val_00_0=0
>teval $p_pe_val_03_0=0
>teval $p_pe_val_05_0=0
>end
tstart
#Wait some time that current pc receive some tcp package.
(gdb) tstop
(gdb) tfind

```

```
(gdb) spe 0 0 $cpu_id  
$p_pe_val_00_2=12676  
(gdb) spe 0 3 $cpu_id  
$p_pe_val_03_2=7  
(gdb) spe 0 5 $cpu_id  
$p_pe_val_05_2=97
```

# Appendix A Preparatory work before use KGTP

## Linux kernel

### If your system use the Linux kernel that is built by yourself

To use KGTP, your Linux kernel need open following options:

*General setup --->*

*[\*] Kprobes*

*[\*] Enable loadable module support --->*

*Kernel hacking --->*

*[\*] Debug Filesystem*

*[\*] Compile the kernel with debug info*

Please rebuild your Linux kernel if you change any options of the config.

## If use with Linux kernel of Android

The default Linux kernel config of Android should not support KGTP. To use KGTP, Linux kernel of Android need open following options:

*[\*] Enable loadable module support --->*

*General setup --->*

*[\*] Prompt for development and/or incomplete code/drivers*

*[\*] Kprobes*

*Kernel hacking --->*

*[\*] Debug Filesystem*

*[\*] Compile the kernel with debug info*

Please rebuild your Linux kernel if you change any options of the Linux kernel config.

# If your system use the Linux kernel from distribution

You need install some Linux kernel package.

## Ubuntu

### The standard method of install the Linux kernel debug image

#### 1) Add debug source to the sources list of Ubuntu.

Create an /etc/apt/sources.list.d/ddebs.list by running the following line at a terminal:

```
echo "deb http://ddebs.ubuntu.com $(lsb_release -cs) main restricted  
universe multiverse" | \  
sudo tee -a /etc/apt/sources.list.d/ddebs.list
```

Stable releases (not alphas and betas) require three more lines adding to the same file, which is done by the following terminal command:

```
echo "deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates main  
restricted universe multiverse  
deb http://ddebs.ubuntu.com $(lsb_release -cs)-security main  
restricted universe multiverse  
deb http://ddebs.ubuntu.com $(lsb_release -cs)-proposed main  
restricted universe multiverse" | \  
sudo tee -a /etc/apt/sources.list.d/ddebs.list
```

Import the debug symbol archive signing key:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys  
428D7C01
```

Then run:

```
sudo apt-get update
```

#### 2) Get Linux kernel debug image

```
sudo apt-get install linux-image-$(uname -r)-dbg
```

Then you can find Linux kernel debug image in "/usr/lib/debug/boot/vmlinux-\$(uname -r)".

Please **note** that this step **Get Linux kernel debug image** need do again when Linux kernel update.

## The second method of install the Linux kernel debug image

If you got some trouble with the standard method, please use following commands to install the Linux kernel debug image.

```
wget http://ddebs.ubuntu.com/pool/main/l/linux/linux-image-$(uname -r)-dbgsym_$(dpkg -s linux-image-$(uname -r) | grep ^Version: | sed 's/Version: //' )_$(uname -i | sed 's/x86_64/amd64/').ddeb
sudo dpkg -i linux-image-$(uname -r)-dbgsym_$(dpkg -s linux-image-$(uname -r) | grep ^Version: | sed 's/Version: //' )_$(uname -i | sed 's/x86_64/amd64/').ddeb
```

Please **note** that this method need do again when Linux kernel update.

## Install the Linux kernel headers

```
sudo apt-get install linux-headers-generic
```

## Install the Linux kernel source

### New way

Install package that we need:

```
sudo apt-get install dpkg-dev
```

Get the Linux kernel source:

```
apt-get source linux-image-$(uname -r)
```

Then you can find Linux kernel directory in current directory.

Move this directory to "/build/builddd/".

### Old way

Install the source package:

```
sudo apt-get install linux-source
```

Uncompress the source package:

```
sudo mkdir -p /build/builddd/
sudo tar vxjf /usr/src/linux-source-$(uname -r | sed 's/-.*//').tar.bz2 -C /build/builddd/
sudo rm -rf /build/builddd/linux-$(uname -r | sed 's/-.*//')
sudo mv /build/builddd/linux-source-$(uname -r | sed 's/-.*//') /build/builddd/linux-$(uname -r | sed 's/-.*//')
```

Please **note** that this step **Install the Linux kernel source** need do again when Linux kernel update.

## Fedora

### Install the Linux kernel debug image

Use following command:

```
sudo debuginfo-install kernel
```

Or:

```
sudo yum --enablerepo=fedora-debuginfo install kernel-debuginfo
```

Then you can find Linux kernel debug image in `"/usr/lib/debug/lib/modules/(uname -r)/vmlinux"`.

### Install the Linux kernel devel package

```
sudo yum install kernel-devel-$(uname -r)
```

Please **note** that after update the Linux kernel package, you may need to call this command.

## Make sure current Linux kernel debug image is right

GDB open the right Linux kernel debug image is an very important because GDB will get the debug info and address info from it. So before you use KGTP, please do the check to make sure about it.

There are 2 ways to do the check, what I suggest is do both of them to make sure Linux kernel debug image is right.

Please **note** that if you determine you use the right Linux kernel debug image, but cannot pass these ways. Please see [Handle the issue that Linux kernel debug image's address info is not same with Linux kernel when it running](#).

## Where is the current Linux kernel debug image

In UBUNTU, you can find it in `"/usr/lib/debug/boot/vmlinux-$(uname -r)"`.

In Fedora, you can find it in `"/usr/lib/debug/lib/modules/$(uname -r)/vmlinux"`.

If you build Linux kernel with yourself, file "vmlinux" in the Linux kernel build directory is the debug image.



## Use /proc/kallsyms

In the system that its Linux kernel is what you want to trace, use following command to get the address of sys\_read and sys\_write:

```
sudo cat /proc/kallsyms | grep sys_read  
ffffffff8117a520 T sys_read  
sudo cat /proc/kallsyms | grep sys_write  
ffffffff8117a5b0 T sys_write
```

Then we can get that the address of sys\_read is 0xffffffff8117a520 and the address of sys\_write is 0xffffffff8117a5b0.

After that use GDB get address of sys\_read and sys\_write from Linux kernel debug image:

```
gdb ./vmlinux  
(gdb) p sys_read  
$1 = {long int (unsigned int, char *, size_t)} 0xffffffff8117a520  
<sys_read>  
(gdb) p sys_write  
$2 = {long int (unsigned int, const char *, size_t)} 0xffffffff8117a5b0  
<sys_write>
```

The address of sys\_read and sys\_write is same, so the Linux kernel debug image is right.

## Use linux\_banner

```
sudo gdb ./vmlinux
(gdb) p linux_banner
$1 = "Linux version 3.4.0-rc4+ (teawater@teawater-
Precision-M4600) (gcc version 4.6.3 (GCC) ) #3 SMP Tue Apr
24 13:29:05 CST 2012\n"
```

This linux\_banner is the kernel info inside the Linux kernel debug image.

After that, connect to KGTP following the way in [Make GDB connect to gtp](#) connect to KGTP and print linux\_banner again.

```
(gdb) target remote /sys/kernel/debug/gtp
Remote debugging using /sys/kernel/debug/gtp
0x0000000000000000 in irq_stack_union ()
(gdb) p linux_banner
$2 = "Linux version 3.4.0-rc4+ (teawater@teawater-Precision-
M4600) (gcc version 4.6.3 (GCC) ) #3 SMP Tue Apr 24 13:29:05 CST
2012\n"
```

This linux\_banner is the kernel info that Linux kernel that KGTP is tracing. If it is same with the prev kernel info, the Linux kernel debug image is right.

## Handle the issue that Linux kernel debug image's address info is not same with Linux kernel when it running

In X86\_32, you will found that the Linux kernel debug image's address info is not same with Linux kernel when it running through the ways in [Make sure current Linux kernel debug image is right](#). And you determine the Linux kernel debug image is right.

This issue is because:

*Processor type and features --->*

*(0x1000000) Physical address where the kernel is loaded*

*(0x100000) Alignment value to which kernel should be aligned*

The values of these two options are different. Please **note** that the "Physical address where the kernel is loaded" is not showed in config sometimes. You can get its value through search "PHYSICAL\_START".

You can handle this issue through change "Alignment value to which kernel should be aligned" same with "Physical address where the kernel is loaded".

This issue doesn't affect X86\_64.

# Get KGTP

## Get KGTP through http

Please goto <https://github.com/teawater/kgtp/archive/master.zip> get the upstream version of KGTP.

Please goto <https://github.com/teawater/kgtp/archive/release.zip> get the last release of KGTP.

## Get KGTP through git

Following command will get the upstream version of KGTP:

```
git clone https://github.com/teawater/kgtp.git
```

Following command will get the last release version of KGTP:

```
git clone https://github.com/teawater/kgtp.git -b release
```

## **Mirrors**

<https://code.csdn.net/teawater/kgtp>

<https://www.gitshell.com/teawater/kgtp/>

<https://git.oschina.net/teawater/kgtp>

# Config KGTP

Following part is the default config of KGTP inside the Makefile. With this config, KGTP will build together with current kernel that running on this machine.

```
KERNELDIR := /lib/modules/`uname -r`/build  
CROSS_COMPILE :=
```

KERNELDIR is set to the directory which holds the kernel you want to build for. By default, it is set to the kernel that you are running.

Please note that this directory should be Linux kernel build directory or linux-headers directory but not the Linux kernel source directory. And the Linux kernel build directory should be used after build successful.

CROSS\_COMPILE is set to the prefix name of compiler that you want to build KGTP. Empty to compile with your default compiler.

ARCH is the architecture.

Or you can choose which kernel you want build with and which compiler you want use by change Makefile.

For example:

```
KERNELDIR := /home/teawater/kernel/bamd64  
CROSS_COMPILE :=x86_64-glibc_std-  
ARCH := x86_64
```

KERNELDIR is set to /home/teawater/kernel/bamd64. Compiler will use x86\_64-glibc\_std-gcc.

# Compile KGTP

## Normal compile

```
cd kgtp/  
make
```

In some build environment (for example Android) will get some error with user space program getmod or getframe. Please ignore this error and use the gtp.ko in this directory.

If you get error message "/usr/bin/ld: cannot find -lc" in Fedora, please use following command handle it.

```
sudo yum install glibc-static
```



## Compile KGTP with some special config

Most of time, KGTP can auto select right options to build with Various versions of Linux kernel.

But if you want config special options with yourself, you can read following part:

With this option, KGTP will not auto select any build options.

*make AUTO=0*

With this option, KGTP will use simple frame instead of KGTP ring buffer. The simple frame doesn't support gtpframe\_pipe. It just for debug KGTP.

*make AUTO=0 FRAME\_SIMPLE=1*

With this option, \$clock will return rdtsc value instead of local\_clock.

*make AUTO=0 CLOCK\_CYCLE=1*

With this option, KGTP will use procfs instead of debugfs.

*make AUTO=0 USE\_PROC=1*

The options can use together, for example:

*make AUTO=0 FRAME\_SIMPLE=1 CLOCK\_CYCLE=1*

# Install and uninstall KGTP

KGTP don't need to be install because it can insmod directly inside its directory (See [Insmode the KGTP module](#)). But if you need, you can install it to your system.

Install:

```
cd kgtp/  
sudo make install
```

Uninstall:

```
cd kgtp/  
sudo make uninstall
```

# Use KGTP with DKMS

You can use KGTP with DKMS if you want it.

Following commands will copy the files of KGTP to the directory that DKMS need.

```
cd kgtp/  
sudo make dkms
```

Then you can use DKMS commands to control KGTP. Please goto <http://linux.dell.com/dkms/manpage.html> to see how to use DKMS.

# Use KGTP patch for Linux kernel

Most of time, you don't need KGTP patch because KGTP can build as a LKM and very easy to use. But to help some people include KGTP to them special Linux Kernel tree, KGTP supply patches for Linux kernel.

In the KGTP directory:

- **gtp\_3.7\_to\_upstream.patch** is the patch for Linux kernel from 3.7 to upstream.
- **gtp\_3.0\_to\_3.6.patch is the patch** for Linux kernel from 3.0 to 3.6.
- **gtp\_2.6.39.patch is the patch** for Linux kernel 2.6.39.
- **gtp\_2.6.33\_to\_2.6.38.patch** is the patch for Linux kernel from 2.6.33 to 2.6.38.
- **gtp\_2.6.20\_to\_2.6.32.patch** is the patch for Linux kernel from 2.6.20 to 2.6.32.
- **gtp\_older\_to\_2.6.19.patch** is the patch for Linux kernel 2.6.19 and older version.

# Install GDB for KGTP

The GDB that older than 7.6 have some bugs of tracepoint. And some functions of GDB are not very well.

So if your GDB is older than 7.6 please go to <https://code.google.com/p/gdbt/> to get howto install GDB for KGTP. It supplies sources of UBUBTU, CentOS, Fedora, Mandriva, RHEL, SLE, openSUSE. Also have static binary for others.

If you have issue about GDB please get help according to [Get help or report issues about KGTP](#).

# Appendix B How to let GDB connect to KGTP

To use KGTP function need let GDB connect to KGTP first.

## Normal Linux

### Insmod the KGTP module

If you have installed KGTP in your system, you can:

```
sudo modprobe gtp
```

Or you can use the kgtp module in the directory.

```
cd kgtp/  
sudo insmod gtp.ko
```

## Handle the issue that cannot find "/sys/kernel/debug/gtp"

If you got this issue, please make sure "Debug Filesystem" is opened in your kernel config first. Please goto [If your system use the Linux kernel that is built by yourself](#) see how to open it.

If it is opened, please use following command mount sysfs.

```
sudo mount -t sysfs none /sys/
```

Maybe you will got some error for examle "sysfs is already mounted on /sys". Please ignore it.

please use following command mount debugfs.

```
mount -t debugfs none /sys/kernel/debug/
```

Then you can find "/sys/kernel/debug/gtp".

## Make GDB connect to gtp

### Load Linux kernel debug image to GDB

You can open GDB with following command to load image:

*[gdb kernel\\_debug\\_image\\_file](#)*

Or after you open GDB, use following GDB command to load image:

*[file kernel\\_debug\\_image\\_file](#)*

According to “[Where is the current Linux kernel debug image](#)”, You can find Linux kernel debug image (kernel\_debug\_image\_file).

Please **note** that let GDB open a right vmlinux file is very important for KGTP. Please goto “[Make sure current Linux kernel debug image is right](#)” get how to do it.



## GDB on the current machine

```
sudo gdb ./vmlinux  
(gdb) target remote /sys/kernel/debug/gtp  
Remote debugging using /sys/kernel/debug/gtp  
0x0000000000000000 in ?? ()
```

After that, you can begin to use GDB command trace and debug the Linux Kernel.

## GDB on remote machine

Use nc map the KGTP interface to port 1024.

```
sudo su  
nc -l 1234 </sys/kernel/debug/gtp >/sys/kernel/debug/gtp  
#(nc -l -p 1234 </sys/kernel/debug/gtp >/sys/kernel/debug/gtp for  
old version netcat.)
```

After that, nc will hang there to wait connection.

```
Let gdb connect to the port 1234.  
gdb-release ./vmlinux  
(gdb) target remote xxx.xxx.xxx.xxx:1234
```

After that, you can begin to use GDB command trace and debug the Linux Kernel.

# Android

This video introduces use GDB connect to the KGTP in the Android, Please goto [http://youtu.be/\\_UGN2j8Ctg0](http://youtu.be/_UGN2j8Ctg0) or <http://www.tudou.com/programs/view/FjkQ6HhPnfE/> to see it.

## Insmmod the KGTP module

**First**, make sure ADB has connected with Android.

**Second**, copy KGTP module to Android.

*`sudo adb push gtp.ko /`*

Directory "/" may be read-only. You can choice other directory or use command "sudo adb shell mount -o rw,remount /" remount the directory to can write.

**Third**, insmod the module.

*`adb shell insmod /gtp.ko`*

## Handle the issue that cannot find "/sys/kernel/debug/gtp"

If you got this issue, please make sure "Debug Filesystem" is opened in your kernel config first. Please goto

If\_your\_system\_use\_the\_Linux\_kernel\_that\_is\_built\_by\_yourself see howto "[If use with Linux kernel of Android](#)" see howto open it.

If it is opened, please use following command mount sysfs.

```
sudo adb shell mount -t sysfs none /sys/
```

Maybe you will got some error for examle "Device or resource busy". Please ignore it.

please use following command mount debugfs.

```
sudo adb shell mount -t debugfs none /sys/kernel/debug/
```

Then you can find "/sys/kernel/debug/gtp".

## GDB connect to the KGTP

Use nc map the KGTP interface to port 1024.

```
adb forward tcp:1234 tcp:1234  
adb shell "nc -l -p 1234 </sys/kernel/debug/gtp  
>/sys/kernel/debug/gtp"  
 #(adb shell "nc -l 1234 </sys/kernel/debug/gtp  
>/sys/kernel/debug/gtp" for new version netcat.)
```

After that, nc will hang there to wait connection.

Let gdb connect to the port 1234.

```
gdb-release ./vmlinux  
(gdb) target remote :1234
```

After that, you can begin to use GDB command trace and debug the Linux Kernel.

# Appendix C Add module symbols to GDB

Sometimes you need to add a Linux kernel module's symbols to GDB to debug it.

Add symbols with hand is not very easy, so KGTP package include an GDB python script "getmod.py" and a program "getmod" can help you.

## How to use getmod

"getmod" is written by C so you can use it anywhere even if in an embedded environment.

For example:

```
#Following command save Linux Kernel module info to the file  
/tmp/mi in GDB  
#command format.  
sudo getmod >/tmp/mi  
#in gdb part:  
(gdb) source /tmp/mi  
add symbol table from file "/lib/modules/2.6.39-  
rc5+/kernel/fs/nls/nls_iso8859-1.ko" at  
  .text_addr = 0xf80de000  
  .note.gnu.build-id_addr = 0xf80de088  
  .exit.text_addr = 0xf80de074  
  .init.text_addr = 0xf8118000  
  .rodata.str1.1_addr = 0xf80de0ac  
  .rodata_addr = 0xf80de0c0  
  _mcount_loc_addr = 0xf80de9c0  
  .data_addr = 0xf80de9e0  
  .gnu.linkonce.this_module_addr = 0xf80dea00  
#After this GDB command, all the Linux Kernel module info is loaded  
into GDB.
```

If you use remote debug or offline debug, maybe you need change the base directory. Following example is for it.

```
#/lib/modules/2.6.39-rc5+/kernel is replaced to sudo ./getmod -r  
/home/teawater/kernel/b26
```

```
sudo ./getmod -r /home/teawater/kernel/b26 >~/tmp/mi
```

# How to use getmod.py

Please **note** that static build GDB that download from <https://code.google.com/p/gdbt/> cannot use getmod.py.

Connect to KGTP before use the getmod.py.

*(gdb) source ~/kgtp/getmod.py*

Then this script will auto load the Linux kernel module's symbols to GDB.