



---

## Trace-cmd and kernelshark trace viewer



---

## Contents

---

1. Trace-cmd and kernelshark trace viewer .....	3
2. Category:Tracing tools .....	13
3. Ftrace .....	15



A quality version of this page, approved on 9 October 2019, was based off this revision.

## Contents

1 Article purpose .....	4
2 Introduction .....	5
3 Installing the trace and debug tool on your target board .....	7
3.1 Using the STM32MPU Embedded Software distribution .....	7
3.1.1 Distribution Package .....	7
4 Getting started .....	9
4.1 Making a trace using function tracer mode .....	9
4.2 Making a trace using graph function tracer mode .....	9
4.3 Reading the trace .....	10
5 To go further .....	11
5.1 Visualizing trace using kernelshark .....	11
5.1.1 Host installation (Linux environment) .....	11
5.1.2 Opening and reading trace-cmd trace .....	11
6 References .....	13



---

## 1 Article purpose

---

This article provides the basic information needed to start using the Linux tool: **trace-cmd**<sup>[1]</sup> and host trace viewer **kernelshark**<sup>[2]</sup>.



## 2 Introduction

The following table provides a brief description of the tool, as well as its availability depending on the software packages:

✔: this tool is either present (ready to use or to be activated), or can be integrated and activated on the software package.

✘: this tool is not present and cannot be integrated, or it is present but cannot be activated on the software package.

Tool			STM32MPU Embedded Software distribution			STM32MPU Embedded Software distribution for Android™		
Name	Category	Purpose	Starter Package	Developer Package	Distribution Package	Starter Package	Developer Package	Distribution Package
		trace-cmd <sup>[1]</sup> command interacts with the <a href="#">Ftrace</a> tracer that is built inside the Linux kernel. It interfaces with the <a href="#">Ftrace</a> specific files found in the debugfs file system						



Tool			STM32MPU Embedded Software distribution			STM32MPU Embedded Software distribution for Android™		
Name	Category	Purpose	Starter Package	Developer Package	Distribution Package	Starter Package	Developer Package	Distribution Package
trace-cmd	Tracing tools	<p>under the tracing directory.</p> <p><b>kernelshark<sup>[2]</sup></b> is a front-end reader of trace-cmd output. "trace-cmd record" and "trace-cmd extract" create a trace.dat (trace-cmd.dat) file. kernelshark can read this file, and produce a graph and list view of the corresponding data.</p>	✘	✘	✔	✘	✘	✘

Note: in case you want to trace at boot time, it is recommended to use *ftrace*.



## 3 Installing the trace and debug tool on your target board

### 3.1 Using the STM32MPU Embedded Software distribution

#### 3.1.1 Distribution Package

Since Yocto Thud version, **trace-cmd** recipe is no more present in openembedded-core meta package and so is not installed by default in all STM32MPU Software Packages.

Following recipes are proposed to integrate trace-cmd in STM32MPU Embedded Software distribution. It has to be added in *layers/meta-st/meta-st-stm32mp/recipes-kernel*.

- Create *trace-cmd* directory:

```
PC $> mkdir layers/meta-st/meta-st-stm32mp/recipes-kernel/trace-cmd
PC $> cd layers/meta-st/meta-st-stm32mp/recipes-kernel/trace-cmd
```

- Create *trace-cmd.inc* file with the following content:

```
SRCREV = "57371aaa2f469d0ba15fd85276deca7bfdd7ce36"
PV = "2.6.2"

inherit pkgconfig

FILESEXTRAPATHS = . "${FILE_DIRNAME}/trace-cmd:"

SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git;
branch=trace-cmd-stable-v2.6"

S = "${WORKDIR}/git"
```

- Create *trace-cmd\_git.bb* file with the following content:

```
SUMMARY = "User interface to Ftrace"
HOMEPAGE = "http://git.kernel.org/"
LICENSE = "GPLv2 & LGPLv2.1"

require trace-cmd.inc

LIC_FILES_CHKSUM = "file://COPYING;md5=751419260aa954499f7abaabaa882bbe \
file://trace-cmd.c;beginline=6;endline=8;
md5=2c22c965a649ddd7973d7913c5634a5e \
file://COPYING.LIB;md5=edb195fe538e4552c1f6ca0fd7bf4f0a \
file://trace-input.c;beginline=5;endline=8;
md5=3ec82f43bbe0cfb5951ff414ef4d44d0 \
"

EXTRA_OEMAKE = "\
'prefix=${prefix}' \
'bindir=${bindir}' \
'man_dir=${mandir}' \
'html_install=${datadir}/kernelshark/html' \
'img_install=${datadir}/kernelshark/html/images' \
\
'bindir_relative=${@oe.path.relative(prefix, bindir)}' \
'libdir=${libdir}' \
```



```

\
NO_PYTHON=1 \
"
do_compile_prepend() {
# Make sure the recompile is OK
rm -f ${B}/.*.d
}
do_install() {
oe_runmake DESTDIR="${D}" install
}

```

- Add *trace-cmd* package install to the *st-image-weston* image:

```

PC $> echo 'IMAGE_INSTALL_append += "trace-cmd"' >> layers/meta-st/meta-st-openstlinux
/recipes-st/images/st-image-weston.bbappend

```

- Rebuilt the *st-image-weston* target:

```

PC $> cd <build directory>
PC $> bitbake st-image-weston

```

When testing the new image on the target board, *trace-cmd* can be found in the following path:

```

Board $> which trace-cmd
/usr/bin/trace-cmd

```

**However, an additional configuration of the Linux kernel must be performed to obtain the trace.**

This is the configuration used for **ftrace**. Please refer to [Ftrace install](#) to complete the configuration of the Linux kernel.

**This is the reason why you can not use this tool in Starter and Developer Packages context.**





## 4 Getting started

trace-cmd inherits all available configurations from Ftrace.

It is possible to get filtered trace with both function and graph function mode.

It is also possible to trace specific prints done in the source file.

You first need to mount the tracefs:

```
Board $> mount -t tracefs nodev /sys/kernel/tracing
```

To find out which tracers are available, simply cat the available\_tracers file in the tracing directory:

```
Board $> cat /sys/kernel/tracing/available_tracers
function_graph function nop
```

**Note that you must be placed on the target board within a directory path where you have access rights for writing the generated trace file.**

### 4.1 Making a trace using function tracer mode

The example below shows how to filter on a set of functions that contains uart:

- To get a trace on UART, you can connect to the target board with the new console (i.e. through ssh), and then type a simple command, like "/s"
- Then press Ctrl-C to stop the trace

```
Board $> trace-cmd record -p function -l *uart*
  plugin 'function'
Hit Ctrl^C to stop recording
^CCPU0 data recorded at offset=0x15d000
  4096 bytes in size
CPU1 data recorded at offset=0x15e000
  4096 bytes in size
```

See below for [reading the trace](#).

### 4.2 Making a trace using graph function tracer mode

The example below shows how to filter on a set of graph functions that contains uart:

- To get trace on UART, you can connect to the target board with the new console (i.e. through ssh), and then type a simple command, like "/s"
- Then press Ctrl-C to stop the trace



```
Board $> trace-cmd record -p function_graph -l *uart*
plugin 'function_graph'
Hit Ctrl^C to stop recording
^CCPU0 data recorded at offset=0x15d000
4096 bytes in size
CPU1 data recorded at offset=0x15e000
4096 bytes in size
```

See below for [reading the trace](#).

### 4.3 Reading the trace

Command to read the trace:

```
Board $> trace-cmd report
```

By default, it reads the previously created file *trace.dat* at the position where the command is entered.

If you want to specify an other file name:

```
Board $> trace-cmd report -i mytrace.dat
or
Board $> trace-cmd report mytrace.dat
```



## 5 To go further

### 5.1 Visualizing trace using kernelshark

#### 5.1.1 Host installation (Linux environment)

```
PC $> sudo apt-get install kernelshark
```

#### 5.1.2 Opening and reading trace-cmd trace

kernelshark can read trace-cmd report and produce a graph and list view of the data. Please refer to kernelshark home page<sup>[2]</sup> for detail.

- Get the trace from the board. It is possible to get the trace directly from the SDCard under use, or remotely from the target board:

```
PC $> scp root@<board_ip_address>/<path_to>/trace.dat <path_on_host>
```

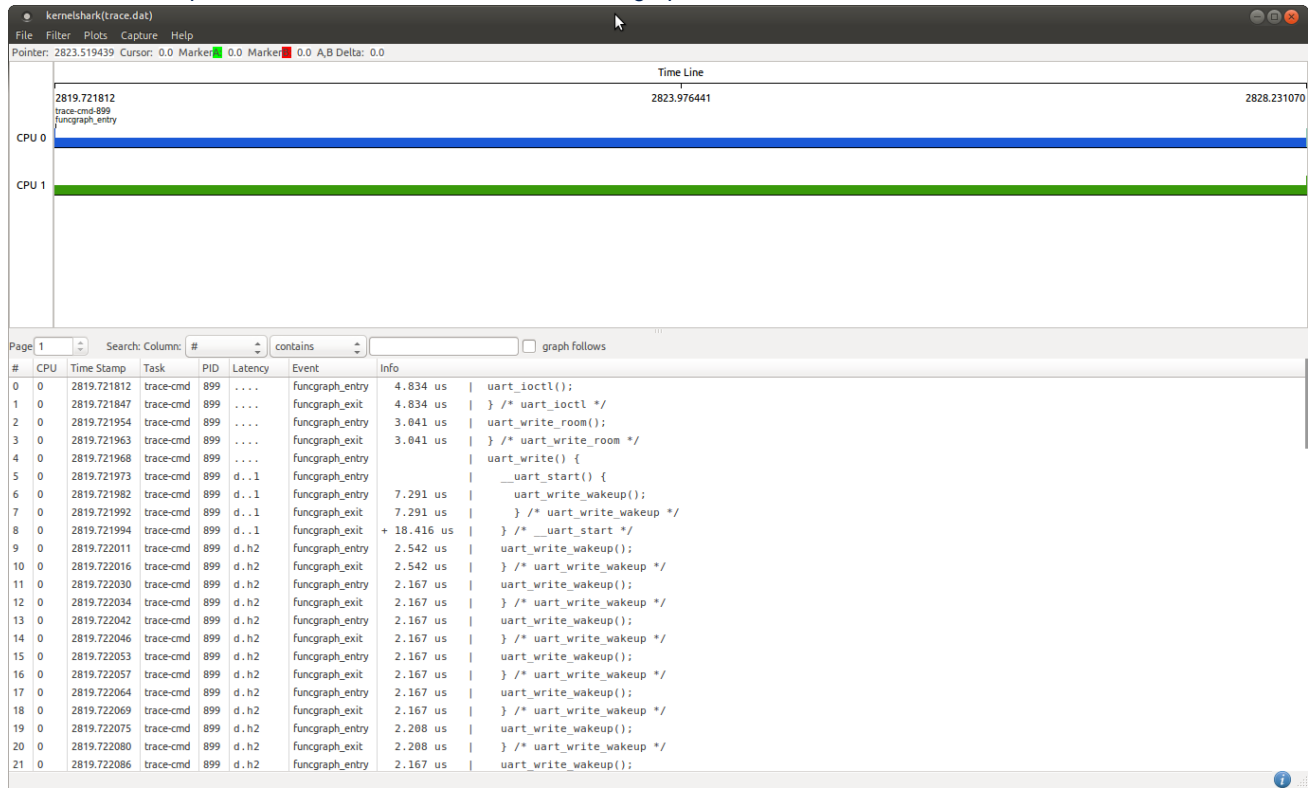
- Open trace with kernelshark

It can be done by opening kernelshark and then opening trace.dat file from the menu, or using the following command:

```
PC $> kernelshark <path_on_host>/trace.dat
```



Here is the example of a trace filtered with UART function\_graph mode:





## 6 References

- 1.01.1 <https://lwn.net/Articles/410200/>
- 2.02.12.2 <http://rostedt.homelinux.com/kernelshark>

- Useful external links

Document link	Document Type	Description
<a href="#">Using KernelShark</a>	User Guide	<a href="https://lwn.net">https://lwn.net</a>
<a href="#">Trace-cmd presentation</a>	User Guide	<a href="https://lwn.net">https://lwn.net</a>
<a href="#">KernelShark quick tutorial</a>	Presentation	<a href="https://elinux.org">https://elinux.org</a>

### Universal Asynchronous Receiver/Transmitter

Stable: 17.06.2020 - 15:27 / Revision: 16.01.2020 - 13:36

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles and subcategories related to the **tracing tools and methods** that allow to get dynamic logs about the Linux® frameworks, the U-Boot or the TF-A.

Das U-Boot -- the Universal Boot Loader (see [U-Boot\\_overview](#))



---

## Subcategories

---

This category has the following 2 subcategories, out of 2 total.

- Linux tracing tools (7 P)
- Android tracing tools (1 P)



## Pages in category "Tracing tools"

The following 4 pages are in this category, out of 4 total.

- [How to debug OP-TEE](#)
- [How to debug TF-A BL2](#)
- [How to debug TF-A SP-MIN](#)
- [U-Boot - How to debug](#)

Stable: 30.03.2021 - 10:18 / Revision: 16.02.2021 - 15:49

A quality version of this page, approved on 30 March 2021, was based off this revision.

### Contents

1 Article purpose .....	16
2 Introduction .....	17
3 Installing the trace and debug tool on your target board .....	19
3.1 Using the STM32MPU Embedded Software distribution .....	19
3.1.1 Developer Package .....	19
3.1.2 Distribution Package .....	19
3.2 Using the STM32MPU Embedded Software distribution for Android™ .....	20
3.2.1 Distribution Package .....	20
4 Getting started .....	22
4.1 Using ftrace at runtime .....	22
4.2 Filter option .....	22
4.2.1 Function tracer mode .....	23
4.2.2 Graph function tracer mode .....	24
4.3 Buffer size .....	26
4.4 Using ftrace at boot time .....	26
4.4.1 With STM32MPU Embedded Software package .....	26
4.4.2 With STM32MPU Embedded Software package for Android .....	27
4.4.3 Checking for trace .....	28
4.4.4 Capturing an oops (from startup) to the serial console .....	28
4.5 Erasing trace .....	29
5 To go further .....	30
5.1 Adding print information for ftrace .....	30
5.2 Stack Trace .....	30
5.3 More tracers .....	32
5.4 Complementary tools .....	33
6 References .....	34



---

## 1 Article purpose

---

This article provides the basic information needed to start using the Linux<sup>®</sup> kernel tool: **ftrace**<sup>[1]</sup>.





## 2 Introduction

The following table provides a brief description of the tool, as well as its availability depending on the software packages:

✔: this tool is either present (ready to use or to be activated), or can be integrated and activated on the software package.

✘: this tool is not present and cannot be integrated, or it is present but cannot be activated on the software package.

Tool			STM32MPU Embedded Software distribution			STM32MPU Embedded Software distribution for Android™		
Name	Category	Purpose	Starter Package	Developer Package	Distribution Package	Starter Package	Developer Package	Distribution Package
ftrace	Tracing tools	ftrace <sup>[1]</sup> (Function Tracer) is a powerful kernel tracing utility that is able, for instance, to trace every kernel function calls and kernel events without adding any	✘	✘	✔	✘	✘	✔



Tool			STM32MPU Embedded Software distribution			STM32MPU Embedded Software distribution for Android™		
Name	Category	Purpose	Starter Package	Developer Package	Distribution Package	Starter Package	Developer Package	Distribution Package
		extra code in your kernel source code						

**Note:** Before Linux kernel 4.1, all the ftrace tracing control files were within the debugfs file system, which is typically located at `/sys/kernel/debug/tracing`. Now, it is located in `/sys/kernel/tracing`, and independent from debugfs.

For backward compatibility, when mounting the debugfs file system, the tracefs file system is automatically mounted at: `/sys/kernel/debug/tracing`.

All files located in the tracefs file system are located in that debugfs file system directory as well.

**Please note that all functions present in the symbol table are available for ftrace. To know if a function is available in the symbol list, you can use the command `"nm vmlinux | grep <function_name>"`**



## 3 Installing the trace and debug tool on your target board

**ftrace** is a kernel feature which is not activated by default in the OpenSTLinux distributions as there is an impact on the Linux kernel size (around 1.5% increase of vmlinux), and also an impact on the overall performance, because of an additional treatment done to the trace kernel events and function calls.

In order to use 'Kernel Function Tracer' required for **ftrace**, the Linux kernel configuration must activate `CONFIG_FUNCTION_TRACER` and `CONFIG_FUNCTION_GRAPH_TRACER` using the Linux kernel menuconfig tool:

```
Symbol: FUNCTION_TRACER
Location:
  Kernel Hacking --->
    Tracers -->
      [*] Kernel Function Tracer

Symbol: FUNCTION_GRAPH_TRACER
Location:
  Kernel Hacking --->
    Tracers -->
      [*] Kernel Function Tracer
      [*] Kernel Function Graph Tracer
```

### 3.1 Using the STM32MPU Embedded Software distribution

#### 3.1.1 Developer Package

It is not recommended to enable the `ftrace` kernel configuration by using the Developer Package, as all external modules should be also recompiled (e.g. *gcnano driver for GPU STM32MP1*), and this is not possible with the Developer Package, which does not necessary provide all the sources.

That is the reason why this is set as not supported for Developer Package.

#### 3.1.2 Distribution Package

- Enable the required Linux kernel configuration  
To enable `CONFIG_FUNCTION_TRACER` and `CONFIG_FUNCTION_GRAPH_TRACER` in the Linux kernel configuration, please refer to [Menuconfig](#) or [how to configure kernel](#) article to get instructions for modifying the configuration and recompiling the Linux kernel image in the Distribution Package context.
- You must also recompile the external Linux kernel module(s) (if existing) being not part of the Linux kernel source tree.  
Example for *gcnano driver of GPU STM32MP1*:

```
PC $> bitbake gcnano-driver-stm32mp
```

- Re-build the full OpenSTLinux image, in order to recompile all dependencies and have correct rootfs including the external Linux kernel modules

```
PC $> bitbake st-image-weston
```



### Warning

As explained before, the size of the uncompressed Linux kernel image increases when enabling the ftrace configuration.

Depending of the memory configuration of your target board (defined in the device tree), an increase of your kernel image can overlap some reserved regions placed after.

In that case you have a compilation error highlighted.

If this overlap has a minor impact (meaning that some features are no more functional but being not critical), this is possible to bypass the compilation error by activating the Linux kernel configuration

`CONFIG_SECTION_MISMATCH_WARN_ONLY` using the Linux kernel Menuconfig tool ([Menuconfig](#) or [how to configure kernel](#))

```
Symbol: SECTION_MISMATCH_WARN_ONLY
Location:
  Kernel Hacking --->
    Compile-time checks and compiler options -->
      [*] Make section mismatch errors non-fatal
```

## 3.2 Using the STM32MPU Embedded Software distribution for Android™

### 3.2.1 Distribution Package

- Enable the required Linux kernel configuration  
To enable `CONFIG_FUNCTION_TRACER` and `CONFIG_FUNCTION_GRAPH_TRACER` in the Linux kernel configuration, please refer to [How to customize kernel for Android](#) article to get instructions for modifying the configuration
- Recompile the Linux kernel image and modules in the Distribution Package for Android context.

```
PC $> build_kernel vmlinux -i
PC $> build_kernel modules -i
```

- You must also recompile the external Linux kernel module(s) (if existing) being not part of the Linux kernel source tree.  
Example for gcnano driver of GPU STM32MP1:

```
PC $> build_kernel gpu -i
```

- Rebuild the full Android images, in order to recompile all dependencies and take into account new prebuilt images for Linux kernel image and modules:

```
PC $> make -j
```

### Warning

As explained before, the size of the uncompressed Linux kernel image increases when enabling the ftrace configuration.



Depending of the memory configuration of your board target (defined in the device tree), an increase of your kernel image can overlap some reserved regions placed after.

In that case you have a compilation error highlighted.

If this overlap has a minor impact (meaning that some features are no more functional but being not critical), this is possible to bypass the compilation error by activating the Linux kernel configuration

CONFIG\_SECTION\_MISMATCH\_WARN\_ONLY using the Linux kernel Menuconfig tool ([How to customize kernel for Android](#))

```
Symbol: SECTION_MISMATCH_WARN_ONLY
```

```
Location:
```

```
Kernel Hacking --->
```

```
Compile-time checks and compiler options -->
```

```
[*] Make section mismatch errors non-fatal
```



## 4 Getting started

### 4.1 Using ftrace at runtime

First of all, you need to enable/activate the ftrace feature from target.


- Once target booted and logged, mount tracefs:

```
Board $> mount -t tracefs nodev /sys/kernel/tracing
```

#### Below information is related to the Android™ distribution

Need to enable root access rights

- Using ADB shell is ADB link available:



```
PC $> adb root
PC $> adb shell
Board $> ...
```

- Using UART console shell:

```
Board $> su
Board $> ...
```

- At that step, all the 'ftrace' features take place in the file system directory path `/sys/kernel/tracing`.

To find out which tracers are available, simply cat the `available_tracers` file in the tracing directory:

```
Board $> cat /sys/kernel/tracing/available_tracers
function_graph function nop
```

More tracers can be added by kernel build configurations. Please refer to [To go further](#) section.

### 4.2 Filter option

#### Warning

ftrace uses a function / graph\_function filter, not a driver filter. As a consequence, tracing *\*myDriver\** functions will not trace the *myHelper* function from *myDriver*

You can get the list of available filter function with the following command:

```
Board $> cat /sys/kernel/tracing/available_filter_functions
```



#### 4.2.1 Function tracer mode

- Start a tracing session

```
Board $> echo 1 > /sys/kernel/tracing/tracing_on
```

- To enable the function tracer, just write **function** to the current\_tracer file. You can then verify the current value:

```
Board $> echo function > /sys/kernel/tracing/current_tracer
Board $> cat /sys/kernel/tracing/current_tracer
function
```

```
Board $> cat /sys/kernel/tracing/trace | head -20
# tracer: function
#
# entries-in-buffer/entries-written: 144045/33695515   #P:2
#
#
#          -----=> irqs-off
#          /-----=> need-resched
#          /-----=> hardirq/softirq
#          /-----=> preempt-depth
#          /-----=> delay
#
# TASK-PID   CPU#  | | | | |  TIMESTAMP  FUNCTION
# | | | | |  | | | | |  | | | | |  |
date-3591   [001]  | | | | |  3278.796042: memblock_is_map_memory <-pfn_valid
date-3591   [001]  | | | | |  3278.796046: unlock_page <-filemap_map_pages
date-3591   [001]  | | | | |  3278.796051: alloc_set_pte <-filemap_map_pages
date-3591   [001]  | | | | |  3278.796053: add_mm_counter_fast <-alloc_set_pte
date-3591   [001]  | | | | |  3278.796055: page_add_file_rmap <-alloc_set_pte
date-3591   [001]  | | | | |  3278.796057: __sync_icache_dcache <-alloc_set_pte
date-3591   [001]  | | | | |  3278.796059: pfn_valid <-__sync_icache_dcache
date-3591   [001]  | | | | |  3278.796061: memblock_is_map_memory <-pfn_valid
date-3591   [001]  | | | | |  3278.796064: unlock_page <-filemap_map_pages
```

- To apply function(s) filter you can set value(s) in `/sys/kernel/tracing/set_ftrace_filter`, then check for new content of the trace:

```
# Here we take the example with all uart functions
Board $> echo "*uart*" > /sys/kernel/tracing/set_ftrace_filter

# Clean the existing trace
Board $> echo > /sys/kernel/tracing/trace

# Display new trace content (in that case, please do some actions in the console to get
some traces)
Board $> cat /sys/kernel/tracing/trace | head -20
# tracer: function
#
#
#          -----=> irqs-off
#          /-----=> need-resched
#          /-----=> hardirq/softirq
#          /-----=> preempt-depth
#          /-----=> delay
#
# TASK-PID   CPU#  | | | | |  TIMESTAMP  FUNCTION
# | | | | |  | | | | |  | | | | |  |
sh-343      [000]  | | | | |  9313.041827: uart_ioctl <-tty_ioctl
sh-343      [000]  | | | | |  9313.041855: uart_ioctl <-tty_ioctl
sh-343      [000]  | | | | |  9313.041866: uart_chars_in_buffer <-tty_wait_until_sent
sh-343      [000]  | | | | |  9313.041870: uart_wait_until_sent <-tty_wait_until_sent
```



```
sh-343 [000] .... 9313.041875: uart_set_termios <-tty_set_termios
sh-343 [000] .... 9313.041968: uart_write_room <-tty_write_room
sh-343 [000] .... 9313.041974: uart_write <-n_tty_write
sh-343 [000] d..1 9313.041979: __uart_start <-uart_write
sh-343 [000] d..1 9313.041987: uart_write_wakeup <-stm32_transmit_chars
sh-343 [000] d.h2 9313.042007: uart_write_wakeup <-stm32_transmit_chars
sh-343 [000] d.h2 9313.042022: uart_write_wakeup <-stm32_transmit_chars
```

More information about filtering option and configuration is available in the Linux documentation for ftrace<sup>[2]</sup>.

- To empty the trace please refer to the paragraph Erasing trace
- To clear out the filter so that all functions are recorded again:

```
Board $> echo > /sys/kernel/tracing/set_ftrace_filter
```

## 4.2.2 Graph function tracer mode

Start a tracing session:

```
Board $> echo 1 > /sys/kernel/tracing/tracing_on
```

To enable the function tracer, just write **function\_graph** into the current\_tracer file. You can then verify the current value:

```
Board $> echo function_graph > /sys/kernel/tracing/current_tracer
Board $> cat /sys/kernel/tracing/current_tracer
function_graph
```

```
Board $> cat /sys/kernel/tracing/trace | head -20
# tracer: function_graph
#
# CPU  DURATION  |          FUNCTION CALLS
# |   |   |   |
1) | 1.015 | us | | _spin_lock_irqsave();
1) | 0.476 | us | | internal_add_timer();
1) | 0.423 | us | | wake_up_idle_cpu();
1) | 0.461 | us | | _spin_unlock_irqrestore();
1) | 4.770 | us | | }
1) | 5.725 | us | | }
1) | 0.450 | us | | mutex_unlock();
1) + 24.243 | us | | }
1) | 0.483 | us | | _spin_lock_irq();
1) | 0.517 | us | | _spin_unlock_irq();
1) |       |   | | prepare_to_wait() {
1) | 0.468 | us | |   _spin_lock_irqsave();
1) | 0.502 | us | |   _spin_unlock_irqrestore();
1) | 2.411 | us | | }
1) | 0.449 | us | | kthread_should_stop();
1) |       |   | | schedule() {
```

To apply a graph function(s) filter, you can set value(s) in `/sys/kernel/tracing/set_graph_function`, then check for the new content of the trace:





```
# Here we take the example with all uart functions
Board $> echo "*uart*" > /sys/kernel/tracing/set_graph_function
```

```
# Clean the existing trace
Board $> echo > /sys/kernel/tracing/trace
```

```
# Display the new trace content (in that case, please do some action in the console to get some traces)
```

```
Board $> cat /sys/kernel/tracing/trace | head -20
# tracer: function_graph
#
# CPU DURATION FUNCTION CALLS
# | | | | |
1) | | | | |
1) 0.875 us | uart_ioctl() {
1) 0.792 us | mutex_lock();
1) + 15.542 us | mutex_unlock();
1) | }
1) 0.583 us | uart_ioctl() {
1) 0.584 us | mutex_lock();
1) 9.792 us | mutex_unlock();
1) | }
1) 0.667 us | uart_chars_in_buffer() {
1) 5.458 us | _raw_spin_lock_irqsave() {
1) | preempt_count_add();
1) 0.583 us | }
1) 5.000 us | _raw_spin_unlock_irqrestore() {
1) + 19.459 us | preempt_count_sub();
1) 1.541 us | }
1) |
1) 0.583 us | uart_wait_until_sent();
1) 0.583 us | uart_set_termios() {
1) + 10.291 us | mutex_lock();
1) | mutex_unlock();
1) | }
1) 0.666 us | uart_write_room() {
1) 5.333 us | _raw_spin_lock_irqsave() {
1) | preempt_count_add();
1) 0.583 us | }
1) 5.000 us | _raw_spin_unlock_irqrestore() {
1) + 19.625 us | preempt_count_sub();
1) | }
1) |
1) 0.625 us | uart_write() {
1) 5.209 us | _raw_spin_lock_irqsave() {
1) | preempt_count_add();
1) | }
1) |
1) | _uart_start() {
1) | stm32_start_tx() {
1) | stm32_transmit_chars() {
```

More information about filtering option and configuration on the Linux documentation for `ftrace`<sup>[2]</sup>.

- To empty the trace please refer to the paragraph [Erasing trace](#)
- To clear out this special filter so that all functions will be recorded again:

```
Board $> echo > /sys/kernel/tracing/set_graph_function
```



### 4.3 Buffer size

A buffer is allocated for each CPU. For making a trace analysis you can change this buffer size (increase or decrease).

- This is possible to read the given size value per CPU, or the total (value is given in kilobytes):

```
# Per CPU
Board $> cat /sys/kernel/tracing/buffer_size_kb
1411
or
Board $> cat /sys/kernel/tracing/per_cpu/cpuX/buffer_size_kb
1411
```

```
# Total for all CPUs: combined size of all the trace buffers
Board $> cat /sys/kernel/tracing/buffer_total_size_kb
2822
```

- To change the value (*note that the trace buffers are allocated in pages (blocks of memory that the kernel uses for allocation, usually 4 Kbytes in size)*)

```
# Same value for each CPU (here 1000*4096/1024=4000)
Board $> echo 4000 > /sys/kernel/tracing/buffer_size_kb
or
# Change buffer size value for a specific CPU X (here 1000*4096/1024=4000)
Board $> echo 4000 > /sys/kernel/tracing/per_cpu/cpuX/buffer_size_kb
```

### 4.4 Using ftrace at boot time

You can use ftrace from the kernel boot, which can be very useful to debug the boot issues.

For this, you have to use the kernel command-line parameters:

- **ftrace** and also **ftrace\_filter** or **ftrace\_graph\_filter** if you want to add filter.

#### 4.4.1 With STM32MPU Embedded Software package

For instance, to modify the kernel bootargs you can do it in the following ways:

- Mount a boot partition from the Linux kernel console, and then update the `extlinux.conf` file using the vi editor (see man page [\[3\]](#), or introduction page [\[4\]](#)). In example:

```
Board $> mount /dev/mmcblk0p4 /boot
# As example for SDCard boot on STM32MP15 Evaluation board, otherwise /boot/<bootdevice>_<
platform>-<boardId>_extlinux/extlinux.conf
Board $> vi /boot/mmc0_stm32mp157c-ev1_extlinux/extlinux.conf
```

Update kernel command-line by adding ftrace parameter:

- function tracer mode

```
root=/dev/mmcblk0p5 rootwait rw console=ttyS3,115200 ftrace=function ftrace_filter=*uart*
```

- function\_graph tracer mode



```
root=/dev/mmcblk0p5 rootwait rw console=ttyS3,115200 ftrace=function_graph
ftrace_graph_filter=*uart*
```

Save and quit file update, and then reboot the board

or

- Edit the extlinux.conf file from the microSD™ card (if used as boot device)

### Warning

Admin rights required

- Insert microSD card on host PC
- Check for boot partition mounted (i.e /media/\$USER/bootfs)
- Edit the extlinux file corresponding to your setup (i.e /media/\$USER/bootfs/mmc0\_stm32mp157c-ev1\_extlinux/extlinux.conf)
- Modify the command-line following your ftrace tracer configuration required (see above)
- Save modification, then insert the microSD card on your target
- Boot and check for kernel command-line

#### 4.4.2 With STM32MPU Embedded Software package for Android

For instance, to modify the kernel bootargs you can do it in the following ways which require boot image rebuilt:

- Edit file `device/stm/<STM32Series>/<BoardId>/Boardconfig.mk`
- Update kernel command-line by adding ftrace parameter in the **BOARD\_KERNEL\_CMDLINE** variable:
  - function tracer mode

```
...
# ===== #
# Kernel command line #
# ===== #
BOARD_KERNEL_CMDLINE := console=ttySTM0,115200 androidboot.console=/dev/ttySTM0
consoleblank=0 earlyprintk
BOARD_KERNEL_CMDLINE += skip_initramfs ro rootfstype=ext4 rootwait
BOARD_KERNEL_CMDLINE += init=/init firmware_class.path=/vendor/firmware
BOARD_KERNEL_CMDLINE += androidboot.hardware=stm
BOARD_KERNEL_CMDLINE += ftrace=function ftrace_filter=*uart*
...
```

- function\_graph tracer mode

```
...
# ===== #
# Kernel command line #
# ===== #
BOARD_KERNEL_CMDLINE := console=ttySTM0,115200 androidboot.console=/dev/ttySTM0
consoleblank=0 earlyprintk
BOARD_KERNEL_CMDLINE += skip_initramfs ro rootfstype=ext4 rootwait
BOARD_KERNEL_CMDLINE += init=/init firmware_class.path=/vendor/firmware
BOARD_KERNEL_CMDLINE += androidboot.hardware=stm
BOARD_KERNEL_CMDLINE += ftrace=function_graph ftrace_graph_filter=*uart*
...
```

- Rebuild and reload the boot image



### 4.4.3 Checking for trace

When booted, to check for the trace, you have to mount the tracefs:

```
Board $> mount -t tracefs nodev /sys/kernel/tracing
```

#### Below information is related to the Android™ distribution

Need to enable root access rights

- Using ADB shell is ADB link available:

```
PC $> adb root
PC $> adb shell
Board $> ...
```

- Using UART console shell:

```
Board $> su
Board $> ...
```

Then look at for trace content (i.e. for function trace):

```
Board $> cat /sys/kernel/tracing/trace | head -20
# tracer: function
#
#
# -----=> irqs-off
# /-----=> need-resched
# /-----=> hardirq/softirq
# /-----=> preempt-depth
# /-----=> delay
#
# TASK-PID   CPU#  | FUNCTION
#-----|-----|-----
systemd-1 [000] | ..... | 1.087213: uart_register_driver <-uart_init
systemd-1 [000] | ..... | 1.087847: uart_get_rs485_mode <-stm32_serial_probe
systemd-1 [000] | ..... | 1.088436: uart_add_one_port <-stm32_serial_probe
systemd-1 [000] | ..... | 1.098000: uart_parse_options <-stm32_console_setup
systemd-1 [000] | ..... | 1.098007: uart_set_options <-stm32_console_setup
systemd-1 [000] | ..... | 1.098014: uart_get_baud_rate <-stm32_set_termios
systemd-1 [000] | d..1 | 1.098019: uart_update_timeout <-stm32_set_termios
systemd-1 [000] | d..1 | 1.098090: uart_console_write <-stm32_console_write
systemd-1 [000] | d..1 | 1.105231: uart_console_write <-stm32_console_write
systemd-1 [000] | d..1 | 1.114697: uart_console_write <-stm32_console_write
systemd-1 [000] | d..1 | 1.120300: uart_console_write <-stm32_console_write
```

### 4.4.4 Capturing an oops (from startup) to the serial console

An interesting application to enable ftrace at boot, is to capture the function calls leading up to a panic by placing the following parameters on the kernel command line:

```
root=/dev/mmcblk0p5 rootwait rw console=ttyS3,115200 ftrace=function ftrace_dump_on_oops
```



---

When the oops occurs, the ftrace buffer will be automatically dumped on the console message.

## 4.5 Erasing trace

This is possible to erase content of trace for ftrace with the following command:

```
Board $> echo > /sys/kernel/tracing/trace
```



## 5 To go further

### 5.1 Adding print information for ftrace

In addition to the printing Linux kernel functions, it is possible to trace the specific debug information with ftrace by using `trace_printk` function.

It can be used just like `printk()`, and can also be used in any context (interrupt code, NMI code, and scheduler code).

`trace_printk` does not output to the console, but writes to the ftrace ring buffer and can be read via the trace file.

To use `trace_printk` function, you have to include `linux/ftrace.h` in your source code:

```
...
#include <linux/ftrace.h>
...
```

Then use `trace_printk` syntax as `printk` (see below example):

```
...
trace_printk("%s: %d uart_tx_stopped(port) %i\n", __FUNCTION__, __LINE__, uart_tx_stopped
(port));
...
```

### 5.2 Stack Trace

*Extracted from Kernel documentation for ftrace<sup>[2]</sup>.*

Since the kernel has a fixed sized stack, it is important to not waste it in functions. A kernel developer must be aware of what the functions allocate on the stack. If they add too much size, the system can be in danger of a stack overflow, and a corruption will occur, usually leading to a system panic.

There are some tools that check this, usually with interrupts periodically checking the usage. But if you can perform a check at every function call that will become very useful. As ftrace provides a function tracer, it makes it convenient to check the stack size at every function call. This is enabled via the stack tracer.

The Linux kernel configuration option `CONFIG_STACK_TRACER` enables the ftrace stack tracing functionality.

```
Symbol: STACK_TRACER
Location:
  Kernel Hacking --->
    Tracers -->
      [*] Trace max stack
```

To enable it, write a '1' into `/proc/sys/kernel/stack_tracer_enabled`.

```
Board $> echo 1 > /proc/sys/kernel/stack_tracer_enabled
```

You can also enable it from the kernel command line to trace the stack size of the kernel during boot up, by adding "stacktrace" to the kernel command line parameter.



```
root=/dev/mmcblk0p5 rootwait rw console=ttyS3,115200 stacktrace
```

When booted, to check for the trace, you have to mount first the tracefs, then display the trace content:

```
Board $> mount -t tracefs nodev /sys/kernel/tracing

Board $> cat /sys/kernel/tracing/stack_max_size
2928

Board $> cat /sys/kernel/tracing/stack_trace
  Depth   Size   Location      (82 entries)
  -----
0)    4328     4   __rcu_read_unlock+0x14/0x68
1)    4324    180  select_task_rq_fair+0x8ac/0xb7c
2)    4144     64  try_to_wake_up+0x100/0x3fc
3)    4080     16  wake_up_process+0x20/0x24
4)    4064     24  swake_up_locked.part.0+0x20/0x38
5)    4040     24  swake_up+0x38/0x48
6)    4016     16  rcu_gp_kthread_wake+0x4c/0x50
7)    4000     24  rcu_report_qs_rsp+0x50/0x84
8)    3976    120  rcu_report_qs_rnp+0x258/0x2ec
9)    3856     80  rcu_process_callbacks+0x290/0x43c
10)   3776     96  __do_softirq+0x12c/0x3ec
11)   3680     16  irq_exit+0xd0/0x118
12)   3664     48  __handle_domain_irq+0x90/0xfc
13)   3616     40  gic_handle_irq+0x5c/0xa0
14)   3576     68  __irq_svc+0x6c/0xa8
15)   3508     28  unwind_get_byte+0x20/0x74
16)   3480    160  unwind_frame+0x1a8/0x6b0
17)   3320     32  walk_stackframe+0x34/0x40
18)   3288     56  __save_stack_trace+0xa4/0xa8
19)   3232     16  save_stack_trace+0x30/0x34
20)   3216     72  create_object+0x120/0x278
21)   3144     40  kmemleak_alloc+0x8c/0xd4
22)   3104     64  kmem_cache_alloc+0x184/0x2f0
23)   3040     64  __kernfs_new_node+0x58/0x15c
24)   2976     24  kernfs_new_node+0x2c/0x48
25)   2952     24  __kernfs_create_file+0x28/0xb8
26)   2928     56  sysfs_add_file_mode_ns+0xc4/0x1a0
27)   2872     24  sysfs_create_file_ns+0x4c/0x58
28)   2848     56  kobject_add_internal+0x174/0x358
29)   2792     40  kobject_add+0x50/0x98
30)   2752     32  irq_sysfs_add+0x44/0x60
31)   2720     72  __irq_alloc_descs+0x174/0x234
32)   2648     48  irq_domain_alloc_descs+0x64/0xe4
33)   2600     56  irq_create_mapping+0x108/0x1fc
34)   2544     56  irq_create_fwspec_mapping+0x140/0x318
35)   2488     88  irq_create_of_mapping+0x5c/0x64
36)   2400    168  of_irq_get+0x68/0x78
37)   2232     24  stpmul_regulator_parse_dt+0x68/0x80
38)   2208     96  regulator_register+0x218/0x970
39)   2112     32  devm_regulator_register+0x54/0x84
40)   2080    136  stpmul_regulator_probe+0x350/0x5f4
41)   1944     32  platform_drv_probe+0x60/0xbc
42)   1912     64  driver_probe_device+0x2f4/0x488
43)   1848     32  __device_attach_driver+0xac/0x14c
44)   1816     40  bus_for_each_drv+0x54/0xa4
45)   1776     40  __device_attach+0xc0/0x150
46)   1736     16  device_initial_probe+0x1c/0x20
47)   1720     32  bus_probe_device+0x94/0x9c
48)   1688     64  device_add+0x3c0/0x5d0
49)   1624     16  of_device_add+0x44/0x4c
50)   1608     40  of_platform_device_create_pdata+0x84/0xb4
```



```

51) 1568 104 of_platform_bus_create+0x160/0x2f8
52) 1464 56 of_platform_populate+0x9c/0x134
53) 1408 32 stpmul_probe+0x6c/0xac
54) 1376 40 i2c_device_probe+0x290/0x2dc
55) 1336 64 driver_probe_device+0x2f4/0x488
56) 1272 32 __device_attach_driver+0xac/0x14c
57) 1240 40 bus_for_each_drv+0x54/0xa4
58) 1200 40 __device_attach+0xc0/0x150
59) 1160 16 device_initial_probe+0x1c/0x20
60) 1144 32 bus_probe_device+0x94/0x9c
61) 1112 64 device_add+0x3c0/0x5d0
62) 1048 24 device_register+0x24/0x28
63) 1024 48 i2c_new_device+0x14c/0x2f4
64) 976 96 of_i2c_register_device+0x134/0x1dc
65) 880 40 of_i2c_register_devices+0x8c/0x100
66) 840 48 i2c_register_adapter+0x184/0x404
67) 792 48 i2c_add_adapter+0xa4/0x138
68) 744 160 stm32f7_i2c_probe+0x954/0xd08
69) 584 32 platform_drv_probe+0x60/0xbc
70) 552 64 driver_probe_device+0x2f4/0x488
71) 488 32 __driver_attach+0x110/0x12c
72) 456 40 bus_for_each_dev+0x5c/0xac
73) 416 16 driver_attach+0x2c/0x30
74) 400 48 bus_add_driver+0x1d0/0x274
75) 352 24 driver_register+0x88/0x104
76) 328 16 __platform_driver_register+0x50/0x58
77) 312 16 stm32f7_i2c_driver_init+0x24/0x28
78) 296 112 do_one_initcall+0x54/0x178
79) 184 72 kernel_init_freeable+0x1dc/0x274
80) 112 24 kernel_init+0x18/0x124
81) 88 88 ret_from_fork+0x14/0x24

```

### 5.3 More tracers

More tracers are available for ftrace. Please refer to the Linux kernel documentation for ftrace<sup>[2]</sup>.

Tracer name	Description
blk	Block tracer. The tracer used by the <code>blktrace</code> user application
hwlat	Hardware Latency tracer. It is used to detect if the hardware produces any latency
irqsoff	Traces the areas that disable interrupts and saves the trace with the longest max latency
preempt off	Similar to irqsoff but traces and records the amount of time for which preemption is disabled
preempti rqssoff	Similar to irqsoff and preemptoff, but traces and records the largest time for which irqs and /or preemption is disabled
wakeup	Traces and records the max latency that it takes for the highest priority task to get scheduled after it has been woken up
wakeup_ rt	Traces and records the max latency that it takes for just RT tasks (as the current "wakeup" does)
wakeup_ dl	Traces and records the max latency that it takes for a SCHED_DEADLINE task to be woken (as the "wakeup" and "wakeup_rt" does)





Tracer name	Description
mmiotrace	A special tracer that is used to trace binary module. It traces all the calls that a module makes to the hardware
branch	This tracer can be configured when tracing likely/unlikely calls within the kernel
nop	This is the "trace nothing" tracer

## 5.4 Complementary tools

While the `debugfs` interface is rather simple, it can also be awkward to work with. Some tools are proposed to simplify the user experience with **ftrace**.

- trace-cmd command line reader and kernelshark trace viewer

**trace-cmd** is a command-line tool that works and interacts with `ftrace` instead of echoing commands into specific files and reading the result from another file. It proposes a high level user command interface to ease the `ftrace` usage.

Associated to `trace-cmd`, **kernelshark** tool proposes a trace viewer useful for analysing the traces.

- Linux Trace Tool suite (LTTng) and Trace Compass viewer

LTTng is an open source tracing framework for Linux partially based on `ftrace`, which also proposes a high level user interface through command line. We associate Trace Compass on Host PC side to a log viewer.



## 6 References

- 1.01.1 <https://elinux.org/Ftrace>
- 2.02.12.22.3 [Documentation/trace/ftrace.rst](#)
- <http://ex-vi.sourceforge.net/vi.html>
- <http://ex-vi.sourceforge.net/viin/paper.html>

- Useful external links

Document link	Document Type	Description
<a href="#">ftrace (kernel.org docs)</a>	Standard	Documentation from Linux kernel sources
<a href="#">Debugging the kernel using Ftrace - part1</a>	User Guide	<a href="http://lwn.net">http://lwn.net</a>
<a href="#">Debugging the kernel using Ftrace - part2</a>	User Guide	<a href="http://lwn.net">http://lwn.net</a>
<a href="#">Using the TRACE_EVENT() macro (with CREATE_TRACE_POINTS)</a>	Training	<a href="http://lwn.net">http://lwn.net</a>
<a href="#">Load perf analysis using ftrace</a>	User Guide	Linaro

Universal Asynchronous Receiver/Transmitter