Debugging in Linux

Presents the Linux commands to monitor, inspect the system state and debugging methods in Linux

* **Pre Requisite**
  + **Have developer package installed and built**
* **Content**

1. View system state with debugfs

2. Know which Linux kernel peripheral driver are started at boot time

3. Device tree in the VFS.

4. Peripheral VFS information UART example (tty in linux)

5. Example of I2C sysfs access of Humidity&temp sensors connected to I2C5 with extension board

6. VFS overview

7. Find device driver source code behind a device entry in /dev.

8. Linux Kernel log

9. TF-A /U-boot log

10. Framework debug log - DRM KMS, V4L2, GStreamer, Wayland Weston, Alsa

11. Advanced Linux advanced traces

12. Monitoring tool

13. A7 step-by-step debugging with GDB (Gnu DeBugger)

13.1 Debug TF-A

13.2 Debug the Linux kernel

13.3 Debug U-boot

13.4 Debug an application

**Color convention**

***Blue rectangle: commands on Linux Host***

***Open Linux Terminal Window and copy/paste into terminal window***

|  |
| --- |
| ***Linux host command lines*** |

Linux host termial window prompt

Ex: ***Open Hyperterminal with following command line from Linux host***

|  |
| --- |
| ***minicom –D /dev/ttyACM0*** |

***Then copy/paste board command lines in Linux host terminal window***

***Pink rectangle: commands on the board***

***Type commands in the hyper terminal window connected to the board***

|  |
| --- |
| ***Board command lines*** |

Target hyper terminal window prompt



**Wiki user guide reference**

[**Debugfs**](https://wiki.st.com/stm32mpu/index.php/Debugfs)

[**How to use the kernel dynamic debug**](https://wiki.st.com/stm32mpu/index.php/How_to_use_the_kernel_dynamic_debug)

[How to find Linux kernel driver associated to a device](https://wiki.st.com/stm32mpu/index.php/How_to_find_Linux_kernel_driver_associated_to_a_device)

[**Trace and debug tools**](https://wiki.st.com/stm32mpu/index.php/Trace_and_debug_tools)

[**STM32MP1 Platform trace and debug environment overview**](https://wiki.st.com/stm32mpu/index.php/STM32MP1_Platform_trace_and_debug_environment_overview)

[**Debugging tools**](https://wiki.st.com/stm32mpu/index.php/Category:Debugging_tools)

[GDB](https://wiki.st.com/stm32mpu/index.php/GDB)

[GDB commands](https://wiki.st.com/stm32mpu/index.php/GDB_commands)

[Gdbgui](https://wiki.st.com/stm32mpu/index.php/Gdbgui)

[Debugging the Linux kernel using the GDB](https://wiki.st.com/stm32mpu/index.php/Debugging_the_Linux_kernel_using_the_GDB)

[U-Boot-How to debug](https://wiki.st.com/stm32mpu/index.php/U-Boot_-_How_to_debug)

[TF-A-How to debug](https://wiki.st.com/stm32mpu/index.php/TF-A_-_How_to_debug)

[OP-TEE-How to debug](https://wiki.st.com/stm32mpu/index.php/OP-TEE_-_How_to_debug)

[STM32-CoPro-MPU plugin for SW4STM32](https://wiki.st.com/stm32mpu/index.php/STM32-CoPro-MPU_plugin_for_SW4STM32)

Serial/TTY overview

This documents refers some time to division “STM32MP1 handsn Trace and Debug.pptx”

about trace and debug that is located in

* [here](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging/TF-A%20U-Boot%20Kernel%20default%20log)

or

* <http://best.st.com/docshare/MCDAppli/Trainings/Documents/Trainings/Products/MP1/MP1%20hands%20on/STM32MP1%20hands%20on%20Trace%20and%20debug%20ext.pdf>

1. **View system state with debugfs**

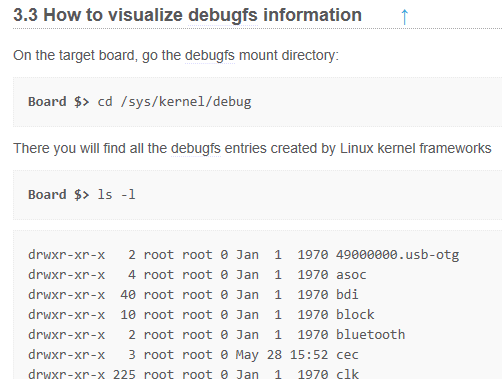
The kernel provides Virtual files (Virtual file system (VFS) means in RAM initialized at boot, not a permanent storage in Flash

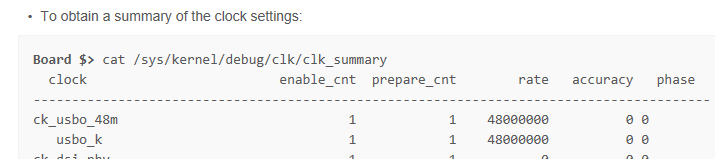
that gives information about **the system configuration and device drivers state**.

See the article: [ Debugfs ]

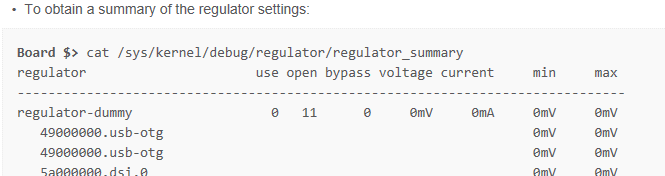
* You can see information about the current setting of many STM32MP1 peripheral.

Under **/sys/kernel/debug** the files are filled by the linux peripheral drivers.

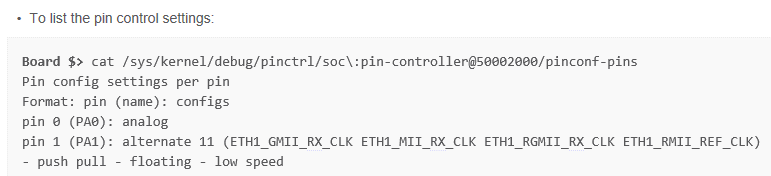


* Interesting configuration info on clock tree values, regulator values, gpios, standby or stop wake-up sources, peripheral registers values

If you want to know how the RCC clock are configured (by TF-A) look the TF-A device tree source or in the procfs, see article for further info [STM32MP15\_clock\_tree]

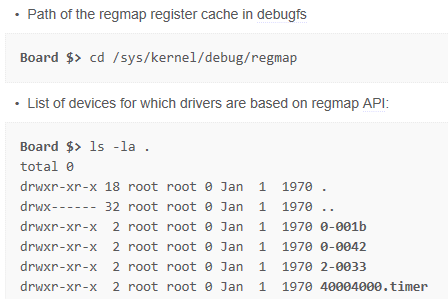


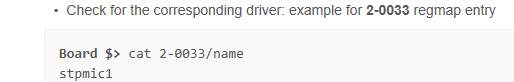
More info in chapter 6 of https://wiki.st.com/stm32mpu/index.php/Regulator\_overview

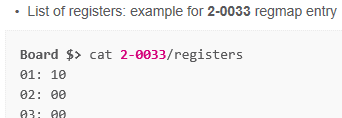


More info in chap 5 of https://wiki.st.com/stm32mpu/wiki/Pinctrl\_overview#How\_to\_debug

Read access to some registers of some peripheral like STPMIC1





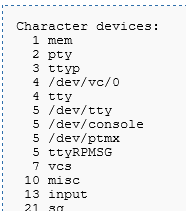


1. **Know which Linux kernel peripheral driver are started at boot time**

***List of all the device drivers ( ie STM32MP1 peripheral driver) that are initialized***

***ie whose driver is “probed” at kernel boot time . There 2 devices type: character and block devices (for storage)***

|  |
| --- |
| ***cat /proc/devices*** |



Example : if you want to know which STM32 Uart is started

|  |
| --- |
| ***ls -d /sys/devices/platform/soc/\*.serial*** |

1. **device tree in the VFS.**

All the nodes declared in the device-tree, including the disabled ones are under: **/proc/device-tree**

**All nodes of the STM32MP1 are under /proc/device-tree/soc**

* **At boot time, the kernel starts (probe) the peripherals whose field in the device tree is “Status=OKAY”.**
* **You can check if a device is configured to be started or not at boot time.**

/proc/device-tree/soc/i2c@40012000/status contains “OKAY” means driver will be “probed’.

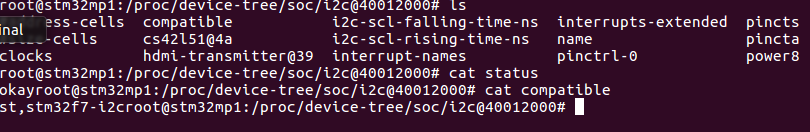
* **Device tree “compatible” field tell the name of the drivers that can drive this IP.**

/proc/device-tree/soc/i2c@40012000/compatible

* **In the driver source code retrieve from this name (grep) the exact the driver source file.**

***“od” below is to open binary file  
(see hand’son lab1 slide 21 for more info.)***

|  |
| --- |
| ***cd /proc/device-tree/soc/i2c@40012000***  ***cat compatible***  ***cat status***  ***od -xi --endian=big i2c-scl-rising-time-ns*** |

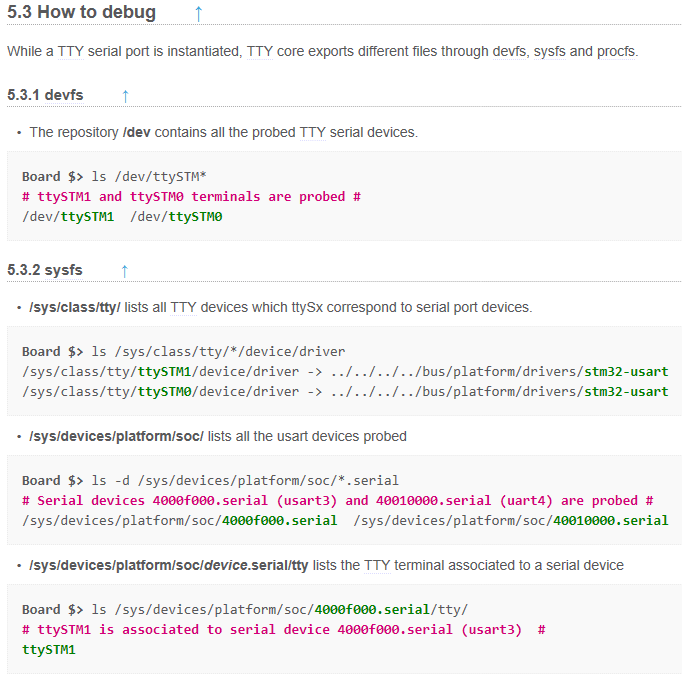


1. **Peripheral VFS information USART example (tty in linux)**

<https://wiki.st.com/stm32mpu/index.php/Serial_TTY_overview>

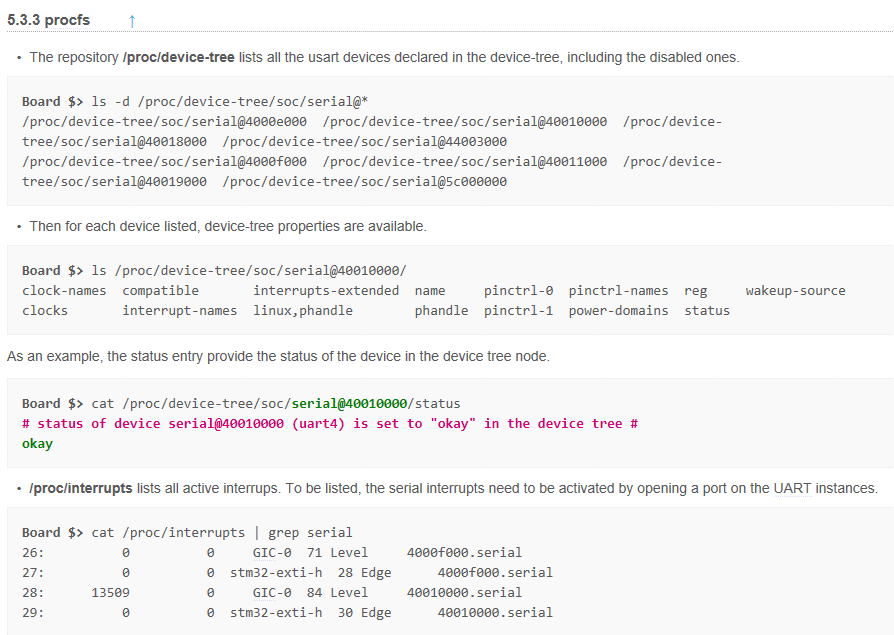
To know

* which USART driver is started (probed) at boot time
* which STM32 USART is behind ttySTM0 or ttySTM1



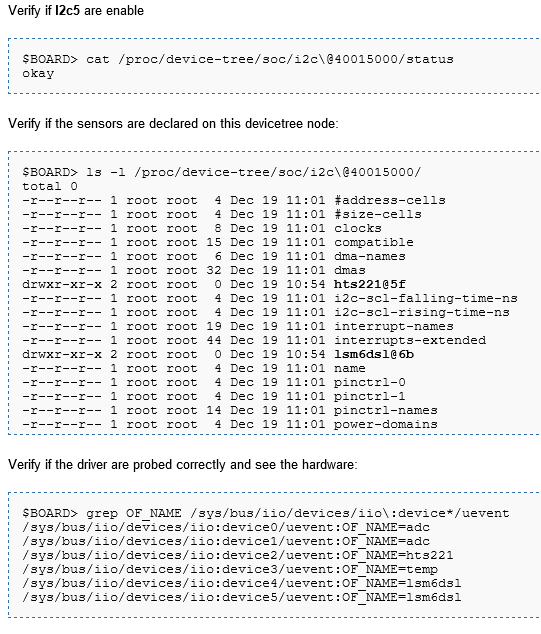
To know how the UART is configured (speed, parity, …)

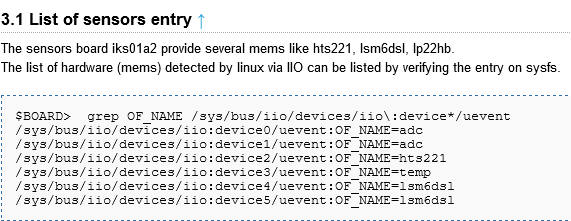
read **the files** in the /proc/device-tree

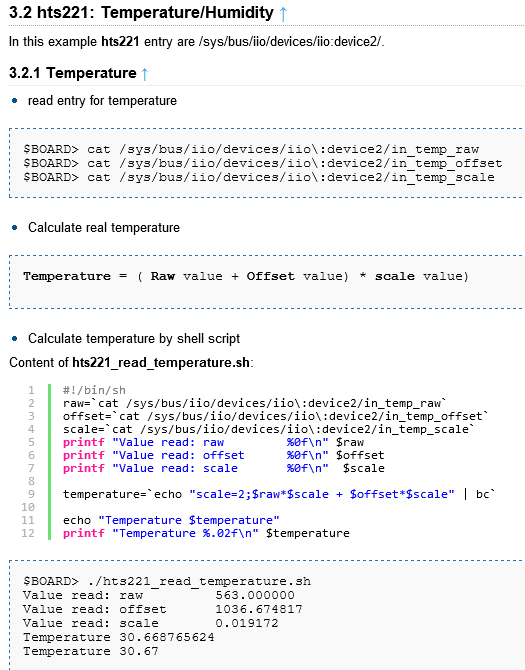


1. **Example of I2C sysfs access of Humidity&temp sensors connected to I2C5 with extension board**

<http://intranet.lme.st.com:8000/php-bin/mm_wiki/index.php/IKS01a2_with_STM32mp157c-dk2>



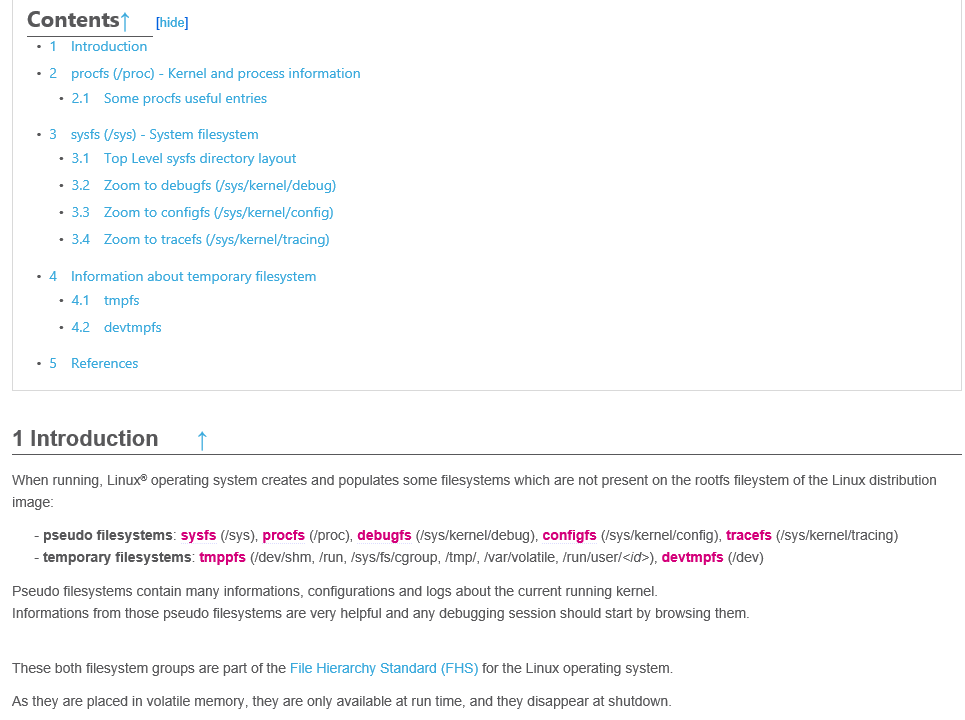


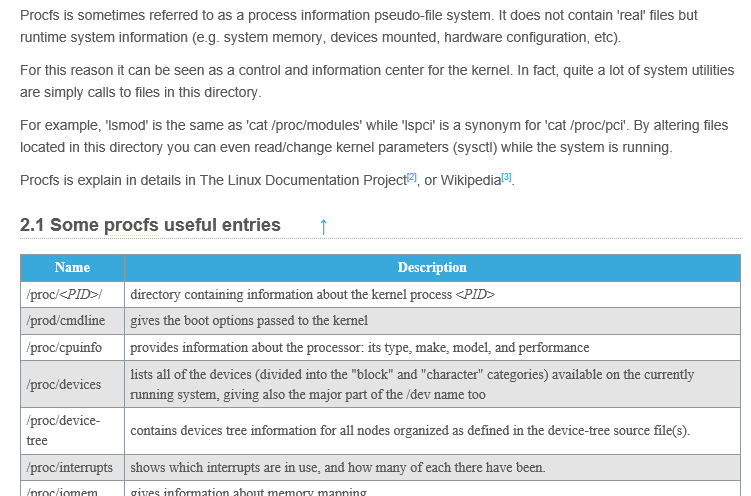
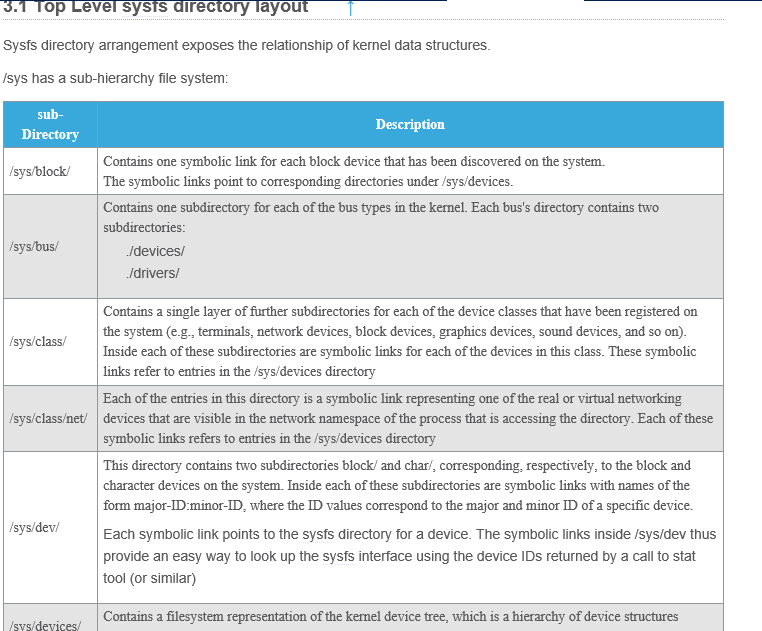


1. **VFS overview**

Short descriptions of a sysfs , procfs and tempfs

<https://wiki.st.com/stm32mpu/index.php/Pseudo_filesystem#sysfs_.28.2Fsys.29_-_System_filesystem>





1. **Find driver source code behind a device entry in /dev.**

See article to see how to retrieve the **driver source code** from the **device of /dev**

**Ex :**  **How to find the source code of the device driver of /dev/ttySTM0 (UART4) ?**

[ How to find Linux kernel driver associated to a device ]

1. **Linux Kernel log**

**The Kernel log is the most first common way to get traces log when something is getting wrong  
 in the kernel side.**

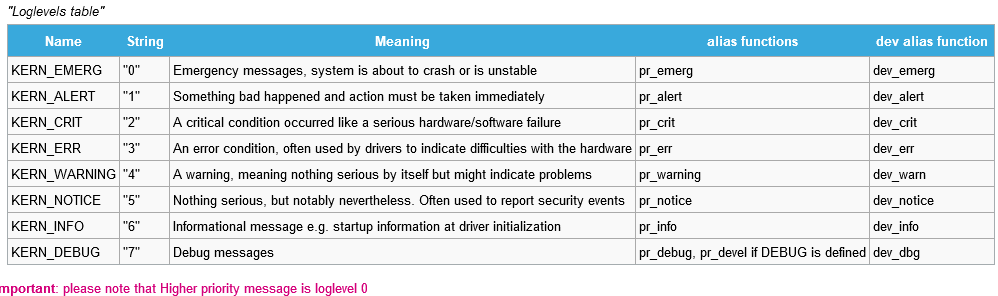
All the kernel drivers/modules *send trace messages* in **a kernel ring buffer** that is accessible for reading with **dmesg** command. Also at early stage -before the ring buffer is initialized-, we can get also the early boot messages.

**See complete information on dmesg**

**[** Dmesg and Linux kernel log **]**

* 1. **See the traces**

Each message sent by the drivers/modules has a “log level”, pre-defined by the source driver code developer.

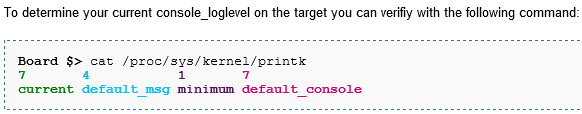


**The messages with log level 0-6 are always present in the kernel ring**

The **debug** messages (log level 7) are present in the ring buffer under condition: the driver/module has to be compiled with the macro **DEBUG** enabled or if the **dynamic debug** is enabled for a given driver/ module.

In parallel, these messages can be sent to the console (via the UART4 -on ST reference board).

The messages arriving on the console can be filtered according to the console log level filter (the are all in the ring buffer)

**The “console log level” is stored in /proc/sys/kernel/printk**

“Current” console\_log leve is 7

=> message 1-7 => **the debug messages (loglevel 8) are not visible on the console.**

**If you want to get the debug messages into console at boot time, change the current console\_log level**

**For that, modify the kernel command line in the extlinux.conf file**

root=/dev/mmcblk0p5 rootwait rw console=ttyS3,115200 loglevel=8

**If you want to get debug message into the console, change the current *console log* level:**

|  |
| --- |
| ***dmesg –n 8***  ***or***  ***echo 8 > /proc/sys/kernel/printk*** |

**If you want to clear the kernel ring buffer**

|  |
| --- |
| ***dmesg –C*** |

* 1. **how to activate the debug messages from one kernel module/driver?**

Debug trace activation can be at **compilation time (define DEBUG**) or **at run time** with the “**kernel Dynamic debug**” methode

**kernel Dynamic debug**

The debug message can be activated for a whole module, for file in the module **if know in which file send a trace message**, or for given message at a line in the file.

This is thanks to /sys/kernel/dynamic\_debug/control VFS file.

Go further in article: [ How to use the kernel dynamic debug ]

**Example:**

**Dynamic debug trace activation for a given file of Usb driver**

|  |
| --- |
| ***echo 'file drivers/usb/core/hub.c +p' > /sys/kernel/debug/dynamic\_debug/control*** |

**Plug the USB key, observe the “hub”, “usb” prefixed debug messages in the ring buffer with dmesg,**

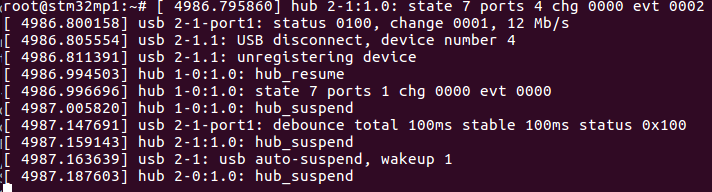
**Note that they are not visible in the console.**

|  |
| --- |
| ***dmesg*** |

**If you want to see , the “hub” “usb” debug messages in the console**

|  |
| --- |
| ***dmesg –n 8*** |

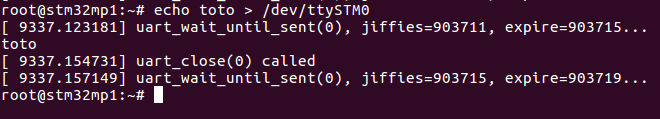
**unplug, Usb key , then observe the “hub” “usb” debug messages are now coming into the console**



**Traces of the Uart driver**

**by sending “toto” characters from target (via ttySTM0 -Uart4) to the Linux host terminal console**

|  |
| --- |
| ***echo "file drivers/tty/\* +p" > /sys/kernel/debug/dynamic\_debug/control***  ***dmesg -n 8***  ***echo toto > /dev/ttySTM0*** |



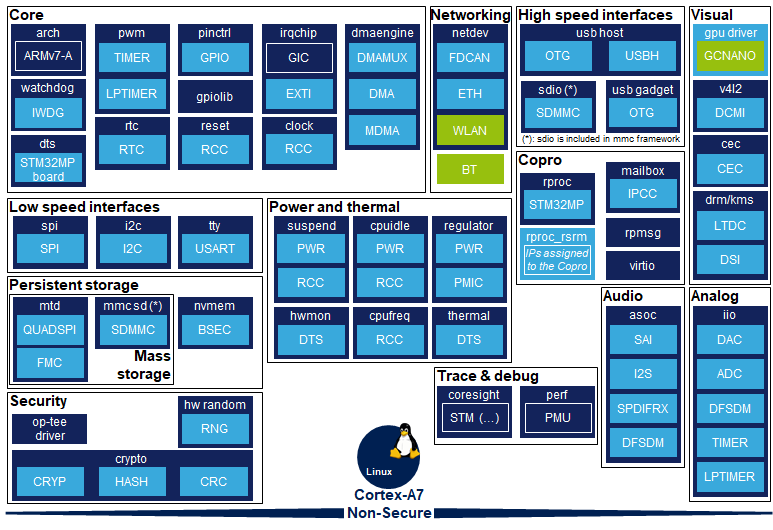
**Go further info in article: Serial TTY overview**

* 1. **Debug the Kernel drivers: *use dynamic debug but not only***

Question is to how to activate the dynamic debug traces for a given kernel module/driver?

The picture below allows accessing to different Linux software frameworks which provide specific trace and debug information in their **"*How to trace and debug the framework*"** dedicated chapter.

**Click** below on desired IP to get the IP *overview* article then goes into “how to trace and debug the framework” to **activate the traces**



1. [ALSA overview](https://wiki.st.com/stm32mpu/index.php/ALSA_overview)
2. [Category:Arm architecture](https://wiki.st.com/stm32mpu/index.php/Category:Arm_architecture)
3. [Bluetooth overview](https://wiki.st.com/stm32mpu/index.php/Bluetooth_overview)
4. [Clock overview](https://wiki.st.com/stm32mpu/index.php/Clock_overview)
5. [CEC overview](https://wiki.st.com/stm32mpu/index.php/CEC_overview)
6. [Linux remoteproc framework overview](https://wiki.st.com/stm32mpu/index.php/Linux_remoteproc_framework_overview)
7. [Resource manager for coprocessing](https://wiki.st.com/stm32mpu/index.php/Resource_manager_for_coprocessing)
8. [Linux Mailbox framework overview](https://wiki.st.com/stm32mpu/index.php/Linux_Mailbox_framework_overview)
9. [Linux RPMsg framework overview](https://wiki.st.com/stm32mpu/index.php/Linux_RPMsg_framework_overview)
10. [Coprocessor management overview](https://wiki.st.com/stm32mpu/index.php/Coprocessor_management_overview)
11. [Hardware random overview](https://wiki.st.com/stm32mpu/index.php/Hardware_random_overview)
12. [Crypto API overview](https://wiki.st.com/stm32mpu/index.php/Crypto_API_overview)
13. [OP-TEE overview](https://wiki.st.com/stm32mpu/index.php/OP-TEE_overview)
14. [Linux kernel device tree](https://wiki.st.com/stm32mpu/index.php/STM32MP15_device_tree)
15. [Dmaengine overview](https://wiki.st.com/stm32mpu/index.php/Dmaengine_overview)
16. [DRM KMS overview](https://wiki.st.com/stm32mpu/index.php/DRM_KMS_overview)
17. [Vivante GCNANO GPU overview](https://wiki.st.com/stm32mpu/index.php/Vivante_GCNANO_GPU_overview)
18. [I2C overview](https://wiki.st.com/stm32mpu/index.php/I2C_overview)
19. [IIO overview](https://wiki.st.com/stm32mpu/index.php/IIO_overview)
20. [Interrupt overview](https://wiki.st.com/stm32mpu/index.php/Interrupt_overview)
21. [Overview of GPIO pins](https://wiki.st.com/stm32mpu/index.php/Overview_of_GPIO_pins)
22. [MMC overview](https://wiki.st.com/stm32mpu/index.php/MMC_overview)
23. [MTD overview](https://wiki.st.com/stm32mpu/index.php/MTD_overview)
24. [NVMEM overview](https://wiki.st.com/stm32mpu/index.php/NVMEM_overview)
25. [CAN overview](https://wiki.st.com/stm32mpu/index.php/CAN_overview)
26. [Ethernet overview](https://wiki.st.com/stm32mpu/index.php/Ethernet_overview)
27. [WLAN overview](https://wiki.st.com/stm32mpu/index.php/WLAN_overview)
28. [Power overview](https://wiki.st.com/stm32mpu/index.php/Power_overview)
29. [Regulator overview](https://wiki.st.com/stm32mpu/index.php/Regulator_overview)
30. [Thermal overview](https://wiki.st.com/stm32mpu/index.php/Thermal_overview)
31. [Reset overview](https://wiki.st.com/stm32mpu/index.php/Reset_overview)
32. [RTC overview](https://wiki.st.com/stm32mpu/index.php/RTC_overview)
33. [Serial TTY overview](https://wiki.st.com/stm32mpu/index.php/Serial_TTY_overview)
34. [SPI overview](https://wiki.st.com/stm32mpu/index.php/SPI_overview)
35. [PWM overview](https://wiki.st.com/stm32mpu/index.php/PWM_overview)
36. [USB overview](https://wiki.st.com/stm32mpu/index.php/USB_overview)
37. [V4L2 camera overview](https://wiki.st.com/stm32mpu/index.php/V4L2_camera_overview)
38. [Watchdog overview](https://wiki.st.com/stm32mpu/index.php/Watchdog_overview)

See wiki :

* STM32MP15 Linux kernel overview
* <https://wiki.st.com/stm32mpu/index.php/Linux_tracing,_monitoring_and_debugging#Trace_and_debug_overview_per_Linux_software_frameworks>

Example

* For SPI click on [SPI overview](https://wiki.st.com/stm32mpu/index.php/SPI_overview) goes

to **5.1.2 Dynamic trace** you get the following instruction to activate the debug traces in kernel ring buffer with dmesg

|  |
| --- |
| ***echo "file spi\* +p" > /sys/kernel/debug/dynamic\_debug/control*** |

if you want to see the message in the console

|  |
| --- |
| ***dmesg –n 8*** |

* For USART [Serial TTY overview](https://wiki.st.com/stm32mpu/index.php/Serial_TTY_overview)

|  |
| --- |
| ***echo "file drivers/tty/\* +p" > /sys/kernel/debug/dynamic\_debug/control*** |

* 1. **Add debug trace in the kernel module or driver**

Add in the source code

**printk("GPIO example exit myName \n");**

and recompile

you will see the trace with dmesg.

See the lab adding a trace in a kernel module in the doc Lab-DeveloperPackageVx.docx in [here](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session4-DeveloperPackage)

“step 10 : STM32MP1 Developer Package Debug Linux kernel module with printk”

* 1. **Additional traces for the kernel framework**

Some framework in the kernel and some application can log interesting message in

/var/log/messages file.

|  |
| --- |
| ***cat /var/log/messages*** |

1. **TF-A /U-boot log**

These logs are not visible via dmesg as the Kernel is not started at this stage.

* TF-A log ; This give board, Soc revision , boot reason (wake up exit), DDR configuration.

See the traces in [here](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging/TF-A%20U-Boot%20Kernel%20default%20log)

By default TF-A the **debug** release in the distribution we see INFO, NOTICE TF-A log level.

In TF-A code :

include/common/debug.h

#define LOG\_LEVEL\_NONE U(0)

#define LOG\_LEVEL\_ERROR U(10)

#define LOG\_LEVEL\_NOTICE U(20)

#define LOG\_LEVEL\_WARNING U(30)

#define LOG\_LEVEL\_INFO U(40)

#define LOG\_LEVEL\_VERBOSE U(50)

tf-a-stm32mp-2.0-r0/arm-trusted-firmware-2.0/Makefile

# Use LOG\_LEVEL\_INFO by default for debug builds

LOG\_LEVEL := 40

else

# Use LOG\_LEVEL\_NOTICE by default for release builds

LOG\_LEVEL := 20

* U-boot

See the traces in [here](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging/TF-A%20U-Boot%20Kernel%20default%20log)

1. **Framework debug log -**

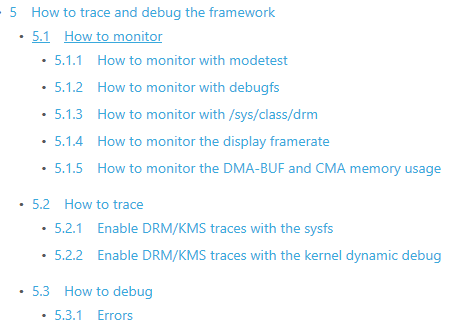
**DRM KMS, V4L2, GStreamer, Wayland Weston, Alsa**

Log activation and content is specific to the framework. See the page below.

* Display: DRM KMS

<https://wiki.st.com/stm32mpu/index.php/DRM_KMS_overview>

There are chapters dedicated:

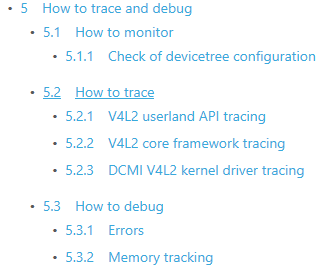


* Video playback: GStreamer

No info on wiki

* Camera: V4L2

<https://wiki.st.com/stm32mpu/index.php/V4L2_camera_overview>



* Windowing: Weston

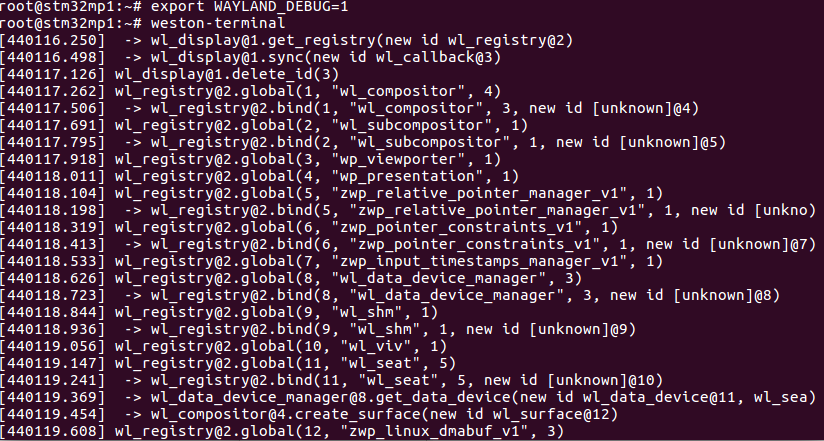
Log can be activated

<https://wiki.st.com/stm32mpu/index.php/How_to_debug_Weston>

Activate the Weston log traces into the console

|  |
| --- |
| ***export WAYLAND\_DEBUG=1***  ***weston-terminal*** |

Activation of the wayland terminal

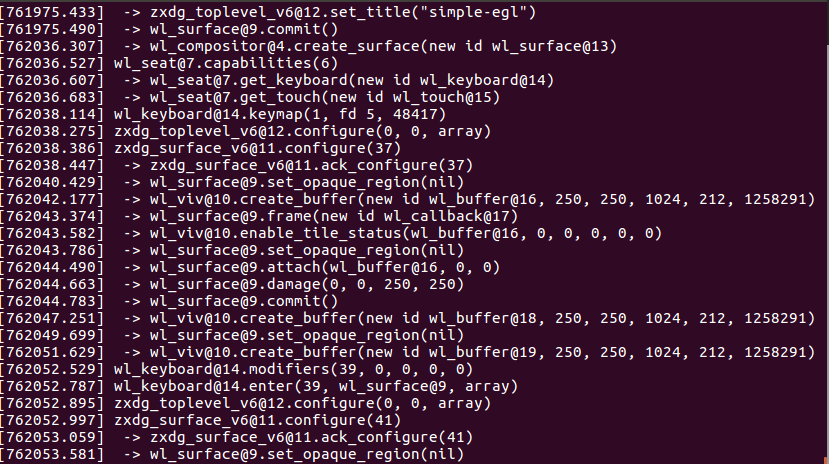


Examples of appli : Weston-simple

<https://wiki.st.com/stm32mpu/index.php/Wayland_Weston_overview>

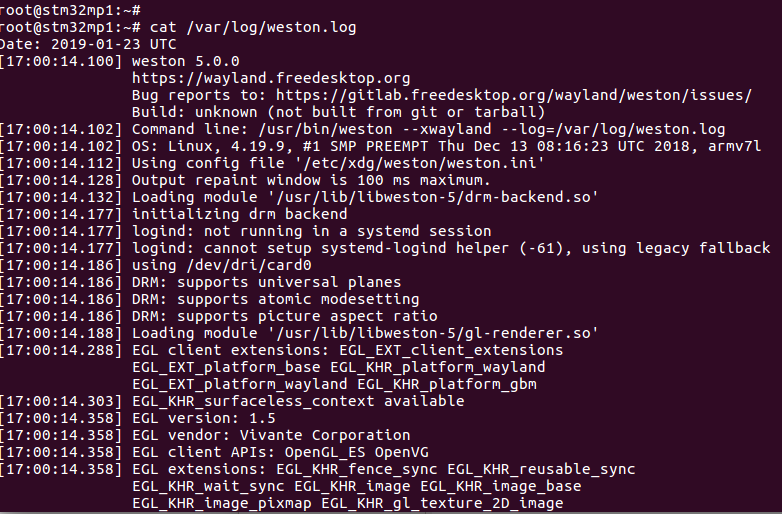
<https://wiki.st.com/stm32mpu/index.php/How_to_stop_and_start_Weston>

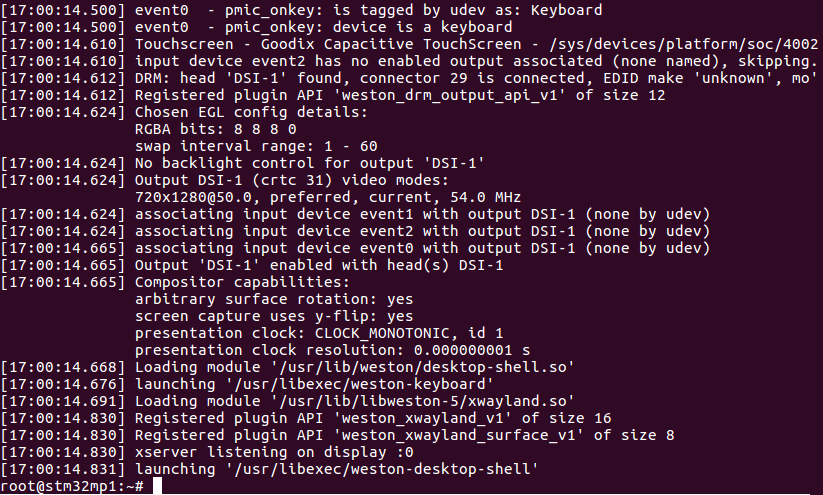
|  |
| --- |
| ***export WAYLAND\_DEBUG=1***  ***weston-simple-egl*** |



Weston init traces into in a log file

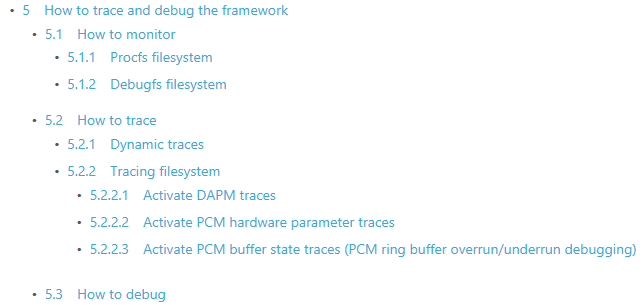
|  |
| --- |
| ***cat /var/log/weston.log*** |



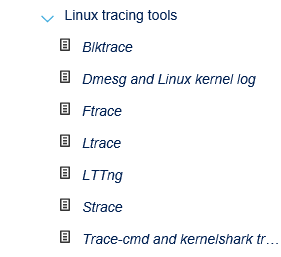


* Audio : ASLA

<https://wiki.st.com/stm32mpu/index.php/ALSA_overview>



1. **Advanced Linux advanced traces**



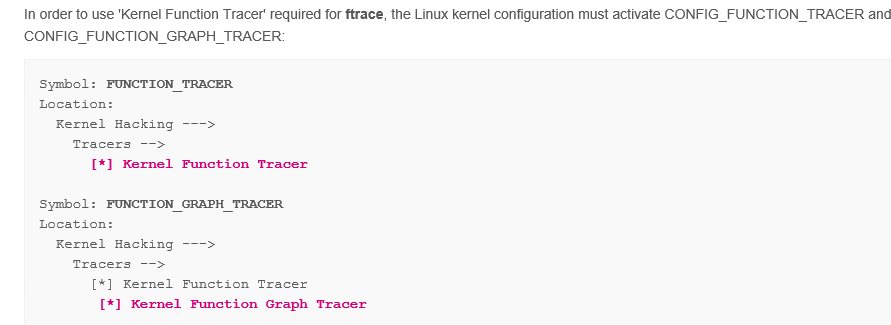
* 1. **Ftrace to observe the function call in the kernel side**

Ftrace show all the kernel functions that are called by an application.

* First, recompile the kernel and external module with Ftrace enabled

<https://wiki.st.com/stm32mpu/index.php/Ftrace#Using_the_STM32MPU_Embedded_Software_distribution>

|  |
| --- |
| ***bitbake virtual/kernel -c menuconfig*** |



Kernel uImage is precompiled with Ftrace enabled in [here](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging/Kernel_uImage_with_ftrace) , copie uImage into your VM desktop ( ~/Desktop )

Install on target uImage (kernel image)

|  |  |
| --- | --- |
| scp ~/Desktop/uImage [root@192.168.7.2:/boot](mailto:root@192.168.7.2:/boot) | |
| ***Sync***  ***reboot*** |

Then, use ftrace …

First

|  |
| --- |
| ***mount -t tracefs nodev /sys/kernel/tracing*** |

See the traces type activated by default :

function = function call

function\_grap = function call and its sub-functions calls

nop

|  |
| --- |
| ***cat /sys/kernel/tracing/available\_tracers*** |

Visualize function call in graph

|  |
| --- |
| ***echo function\_graph > /sys/kernel/tracing/current\_tracer***  ***cat /sys/kernel/tracing/current\_tracer***  ***function\_graph*** |

**Start f traces**

|  |
| --- |
| ***echo 1 > /sys/kernel/tracing/tracing\_on*** |

See all possible functions that can be traced

|  |
| --- |
| ***cat /sys/kernel/tracing/available\_filter\_functions*** |

**Filter** **the functions** with i2c in the name (can be usb\_, uart, etc)

|  |
| --- |
| ***echo "\*i2c\*" > /sys/kernel/tracing/set\_graph\_function*** |

See all possible functions that can be traced: only i2c ones

|  |
| --- |
| ***cat /sys/kernel/tracing/available\_filter\_functions*** |

Clear the uart selection (ic2 ones)

|  |
| --- |
| ***echo > /sys/kernel/tracing/available\_filter\_functions*** |

**Show the 20 first trace lines**

|  |
| --- |
| ***cat /sys/kernel/tracing/trace | head -20*** |

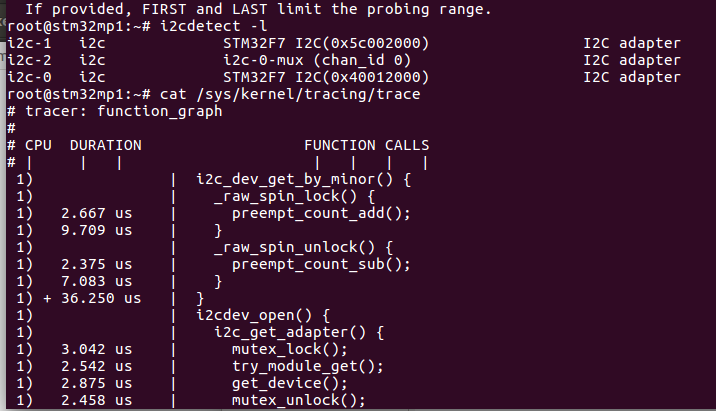
Clean the trace

|  |
| --- |
| ***echo > /sys/kernel/tracing/trace*** |

Example to show I2c interaction with call graph

|  |
| --- |
| ***echo function\_graph > /sys/kernel/tracing/current\_tracer***  ***echo "\*i2c\*" > /sys/kernel/tracing/set\_graph\_function***  ***echo 1 > /sys/kernel/tracing/tracing\_on***  ***echo > /sys/kernel/tracing/trace***  ***cat /sys/kernel/tracing/trace***  ***i2cdetect –l***  ***cat /sys/bus/iio/devices/iio\:device2/in\_temp\_raw***  ***cat /sys/kernel/tracing/trace*** |

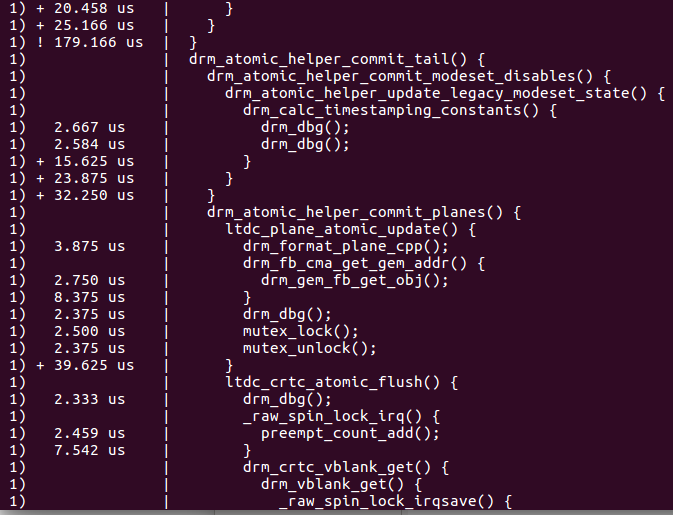
I2cdetect activity kernel activity on target consol



Weston compositor call to Drm

|  |
| --- |
| ***echo "\*drm\*" > /sys/kernel/tracing/set\_graph\_function***  ***Weston-terminal*** |

Weston traces log (on the target console)



It is possible show all the functions called without call graph.

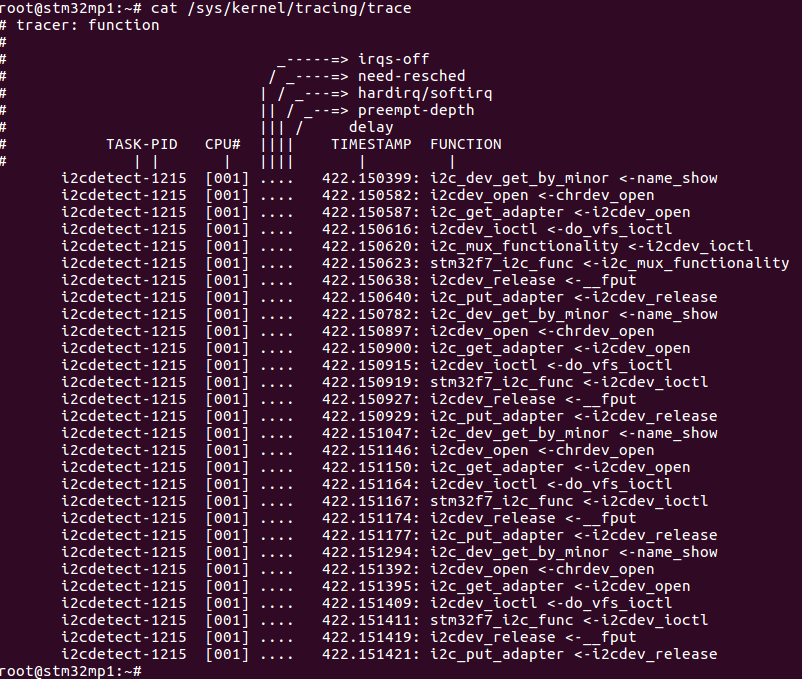
This can be a first approach to check if the wanted function is effectively called.

Also we will catch more functions for a given trace depth size.

Then we can use the function call graph to see which function calls the “wanted” one.

Example to show all functions called whose name contains “i2c”

|  |
| --- |
| ***echo function >  /sys/kernel/tracing/current\_tracer***  cat /sys/kernel/tracing/current\_tracer  echo "\***i2c**\*" > /sys/kernel/tracing/set\_ftrace\_filter  ***echo 1  >  /sys/kernel/tracing/tracing\_on***  ***echo > /sys/kernel/tracing/trace***  ***cat /sys/kernel/tracing/trace***  ***i2cdetect –l***  ***cat /sys/kernel/tracing/trace*** |



See the lab5 of “STM32MP1 hands on Trace and Debug.pptx” in [here](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging/TF-A%20U-Boot%20Kernel%20default%20log)

You can find exercise on

* + Strace
  + Ftrace
  + LTTng
  + Perf - Perf top
  + FlameGraph

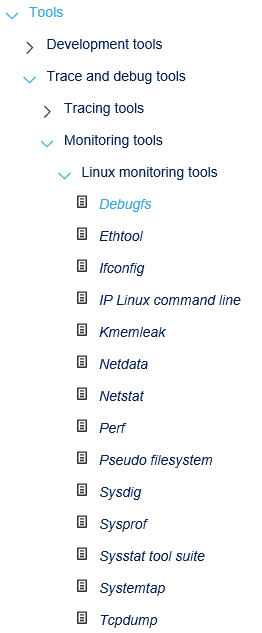
1. **Monitoring tool**

Ethtool: usefull to get the Ethernet I/F state, configuration

iPerf3 : (not in list below) to test end-to-end Ethernet connection performances

Ifconfig :IP I/f

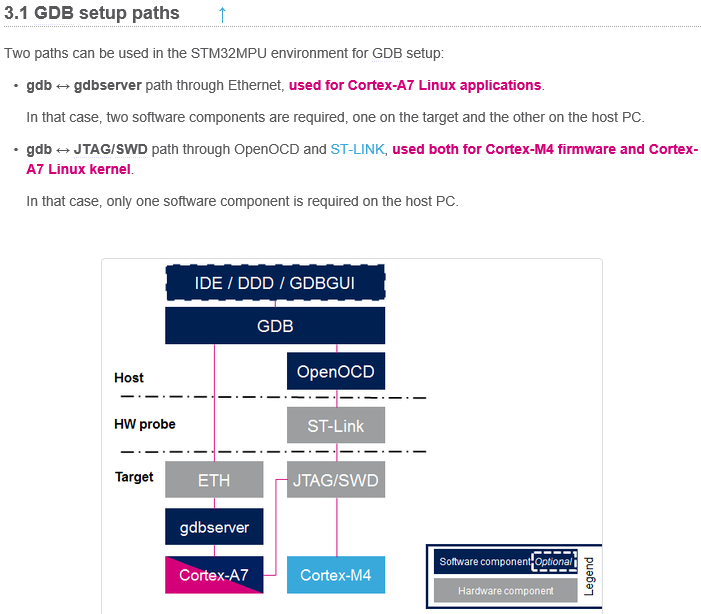
NetData :all system resources monitoring

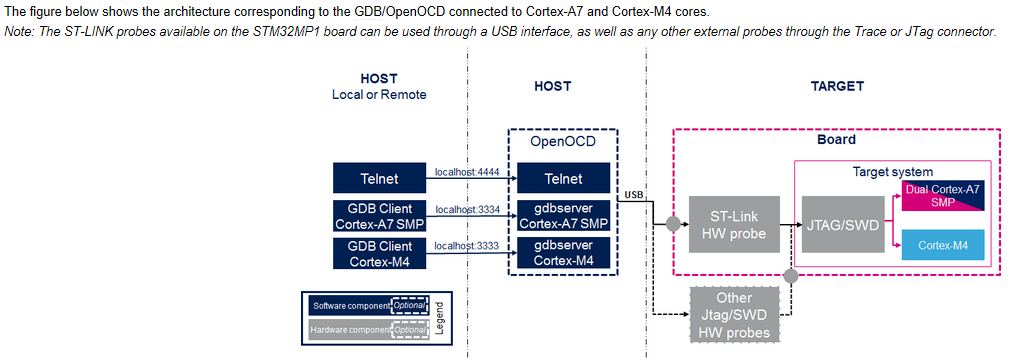


1. **A7 step-by-step debugging with GDB (Gnu DeBugger)**

Reference:

<https://wiki.st.com/stm32mpu/wiki/GDB>





With **GDB,** we can control the Debug cell IP inside the Soc.

Gdb is connected to GDB server (inside OpenOCD) that controls Jtag connection or ST-Link “probe”.

We can step-by-step debug the TF-A, U-boot, Kernel, Linux applications, M4 fw with ST-Link.

In the **developer package,** TF-A, Uboot, Kernel are compiled by default with the debugging symbols (CFLAGS += -g)

* 1. **Debug TF-A**

TF-A is running in A7 secure hw execution context.

1. Get Setup.gdb for above page, Path\_env.gdb and copie in your VM them into **$HOME/gdbscripts**
2. Edit Setup.gdb change it to tell if you debug TF-A,Uboot,Kernel

Example for TF-A :

set $debug\_phase = **1 <-debug TF-A fw**

set $debug\_mode = **0 <- debug from reset**

set $debug\_trusted\_bootchain = **0**

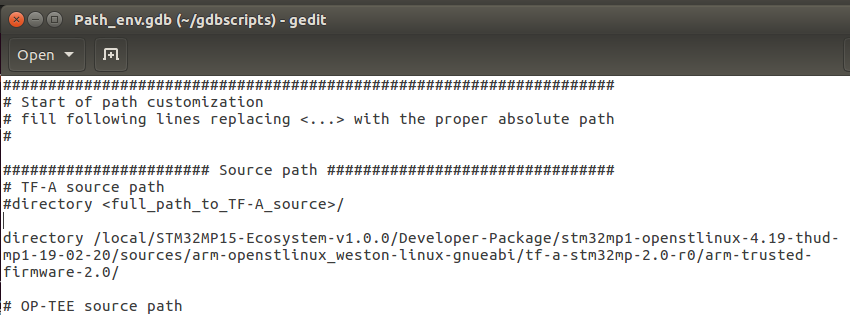
debug\_mode =0 , GDB will reset the Soc and reboot. Debug\_mode=1 GDB “attach” ie connect an already running code on target. It is not possible for TF-A.

1. Update path in Path\_env.gdb with your environment

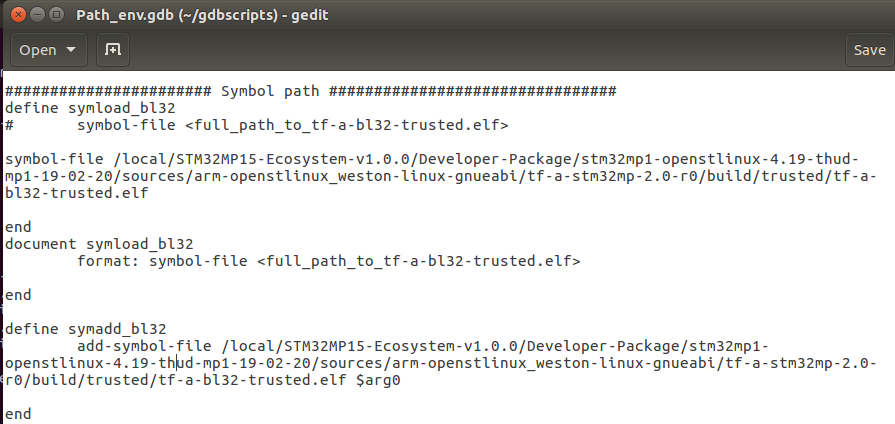
get ***it from session5 repository*** [***here***](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging) for development package environment

Example here for TF-A :

update “directory” with path to tf-a



update the path of “symbol\_bl2”, “symbol\_bl32” with the path in your environement to tf-a-bl2-trusted.elf   
and tf-a-bl32-trusted.elf



1. **Start OpenOCD** server first (cf §5.2.4) (connected Soc Debug cell IP via the ST-Link)

Pre-requisite: Your target is connected to ST-Link and is ON, Linux is running.

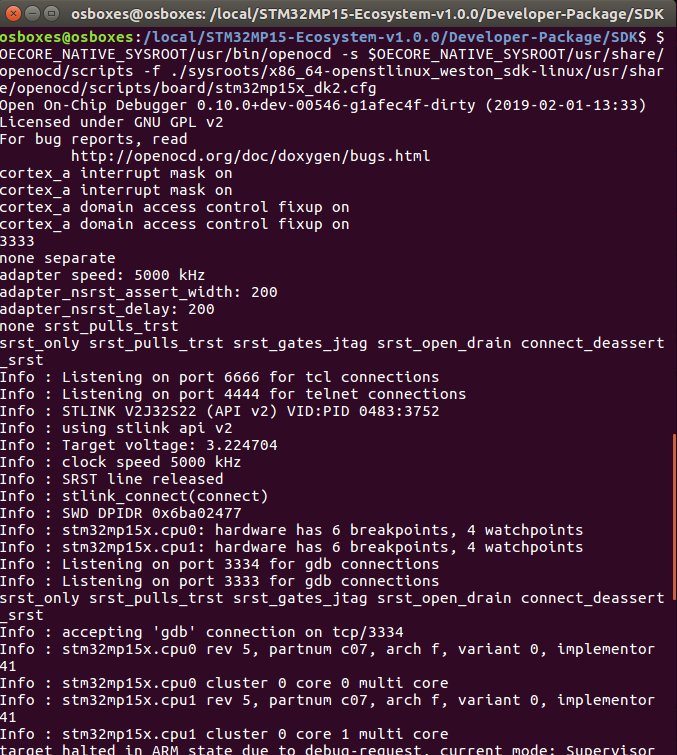
Open new terminal en linux host

|  |
| --- |
| ***cd /local/STM32MP15-Ecosystem-v1.0.0/Developer-Package/SDK***  ***source ./environment-setup-cortexa7t2hf-neon-vfpv4-openstlinux\_weston-linux-gnueabi***  ***$OECORE\_NATIVE\_SYSROOT/usr/bin/openocd -s $OECORE\_NATIVE\_SYSROOT/usr/share/openocd/scripts -f ./sysroots/x86\_64-openstlinux\_weston\_sdk-linux/usr/share/openocd/scripts/board/stm32mp15x\_dk2.cfg*** |

if you have issue to start OpenOCD:

Check that ST-Link is connected in Player/removable device menu.

Check that USB ST-Link drv is installed in your linux host ( stsw-link007, see https://wiki.st.com/stm32mpu/wiki/ST-LINK)



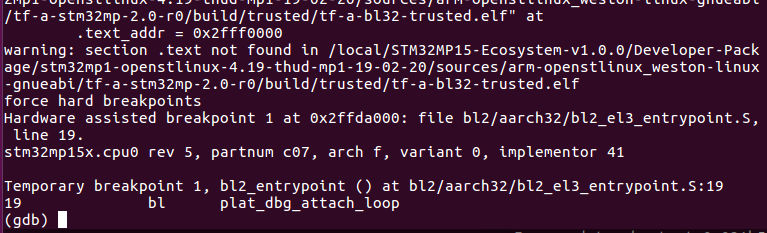
1. **start GDB** (cf §5.2.4) to debug TF-A from boot

Pre-requisite: Your target is connected to ST-Link and is ON, Linux is running.

Open new terminal en linux host

|  |
| --- |
| ***cd $HOME/gdbscripts/***  ***source /local/STM32MP15-Ecosystem-v1.0.0/Developer-Package/SDK/environment-setup-cortexa7t2hf-neon-vfpv4-openstlinu***  ***x\_weston-linux-gnueabi $OECORE\_NATIVE\_SYSROOT/usr/bin/arm-openstlinux\_weston-linux-gnueabi/arm-openstlinux\_weston-linux-gnueabi-gdb -x=$HOME/gdbscripts/Setup.gdb*** |

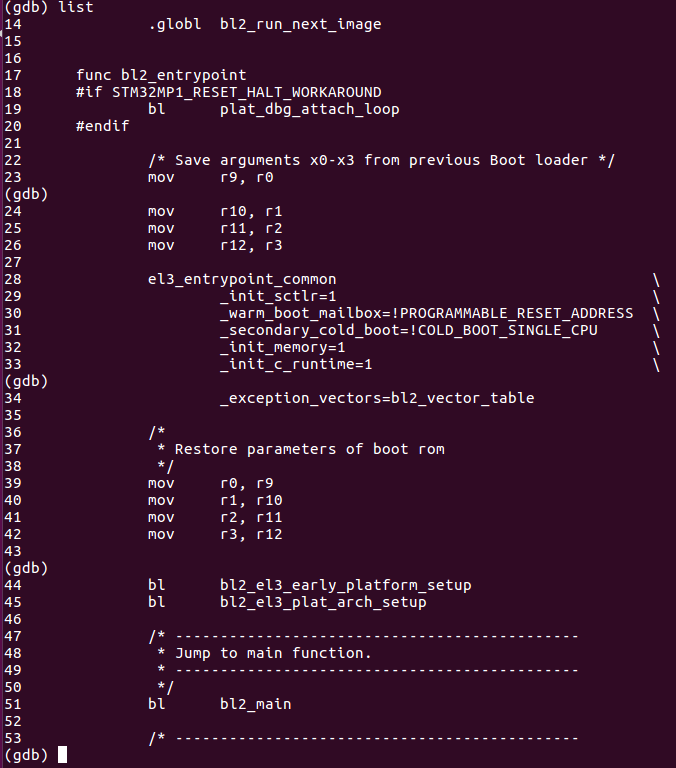
GDB shows A7 is stopped at the beginning of “bl2\_entrypoint()”



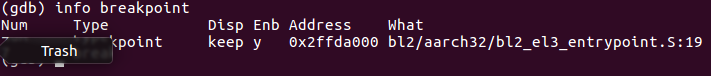
For all the command look in wiki at <https://wiki.st.com/stm32mpu/index.php/GDB_commands>

GDB shows A7 is stopped in line 17 in bl2\_entrypoint() , you can see the code :

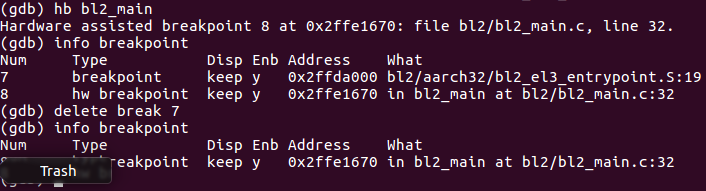
**list** (+ enter to see more code)



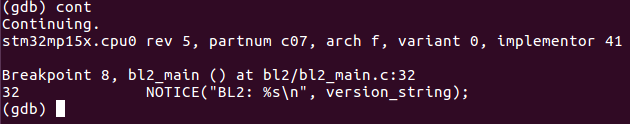
See breakpoints list: info breakpoint



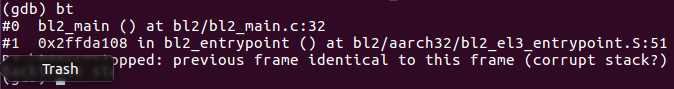
Set breakpoints , delete breakpoint: hb bl2\_main delete break 7



Continue execution to next breakpoint: cont



See the call stack with back trace: bt



Exit gdb : q

To access to variables, recompile TF-A with -**o0** option to deactivate the gcc optimizations

* 1. **Debug the Linux kernel**

1. Edit $HOME/gdbscripts/Setup.gdb

set $debug\_phase = **4 <- debug kernel**

set $debug\_mode = **0**

set $debug\_trusted\_bootchain = **0**

1. Edit path of Vmlinux in $HOME/gdbscript/Path\_env.gdb

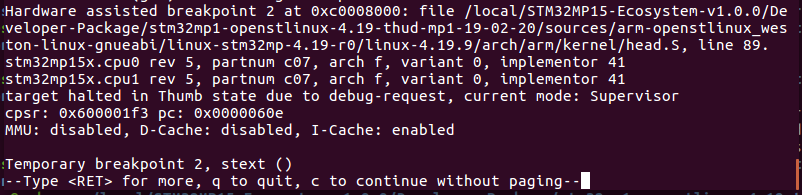
get ***it from session5 repository*** [***here***](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging) for development package environment

1. Open new terminal start OpenOcd

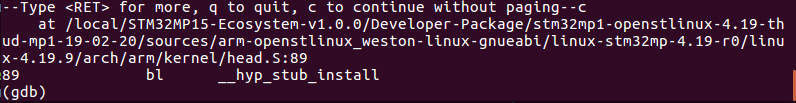
|  |
| --- |
| ***cd /local/STM32MP15-Ecosystem-v1.0.0/Developer-Package/SDK***  ***source ./environment-setup-cortexa7t2hf-neon-vfpv4-openstlinux\_weston-linux-gnueabi***  ***$OECORE\_NATIVE\_SYSROOT/usr/bin/openocd -s $OECORE\_NATIVE\_SYSROOT/usr/share/openocd/scripts -f ./sysroots/x86\_64-openstlinux\_weston\_sdk-linux/usr/share/openocd/scripts/board/stm32mp15x\_dk2.cfg*** |

1. In a new terminal start GDB

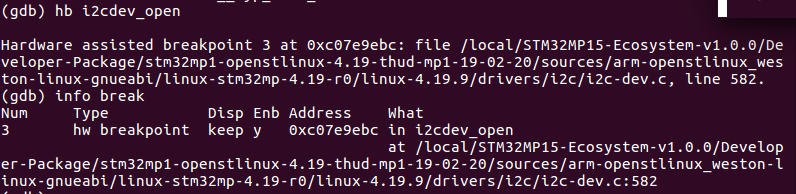
|  |
| --- |
| ***cd $HOME/gdbscripts/***  ***source /local/STM32MP15-Ecosystem-v1.0.0/Developer-Package/SDK/environment-setup-cortexa7t2hf-neon-vfpv4-openstlinux\_weston-linux-gnueabi***  ***$OECORE\_NATIVE\_SYSROOT/usr/bin/arm-openstlinux\_weston-linux-gnueabi/arm-openstlinux\_weston-linux-gnueabi-gdb -x=$HOME/gdbscripts/Setup.gdb*** |



Warning : press c to continue or enter



Set break point into i2cdev\_open : hb i2cdev\_open



Let run the kernel : cont



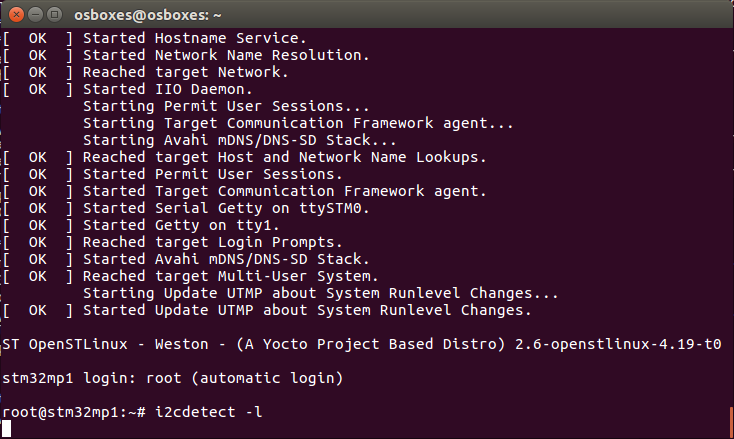
On the target, the I2cdetect application will call “i2cdev\_open” function on which we have just set a breakpoint

(we saw this with ftace below)

Open a terminal on the target :

|  |
| --- |
| ***minicom –D /dev/ttyACM0*** |

|  |
| --- |
| i2cdetect -l |



Gdb shows A7 is stopped in the i2cdev\_open :



* 1. **Debug U-boot**

1. Edit $HOME/gdbscripts/Setup.gdb

set $debug\_phase = **3 <- debug U-boot**

set $debug\_mode = **0**

set $debug\_trusted\_bootchain = **0**

1. Edit path of Vmlinux in $HOME/gdbscript/Path\_env.gdb

get ***it from session5 repository*** [***here***](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session5-Debugging) for development package environment

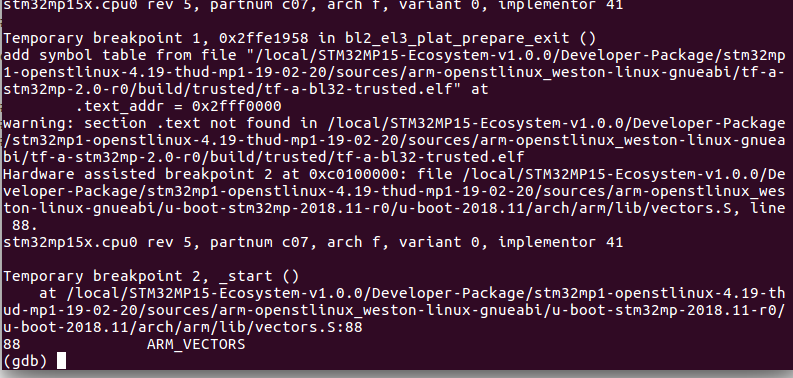
1. Open new terminal, start OpenOcd

|  |
| --- |
| ***cd /local/STM32MP15-Ecosystem-v1.0.0/Developer-Package/SDK***  ***source ./environment-setup-cortexa7t2hf-neon-vfpv4-openstlinux\_weston-linux-gnueabi***  ***$OECORE\_NATIVE\_SYSROOT/usr/bin/openocd -s $OECORE\_NATIVE\_SYSROOT/usr/share/openocd/scripts -f ./sysroots/x86\_64-openstlinux\_weston\_sdk-linux/usr/share/openocd/scripts/board/stm32mp15x\_dk2.cfg*** |

1. In a new terminal, start GDB

|  |
| --- |
| ***cd $HOME/gdbscripts/***  ***source /local/STM32MP15-Ecosystem-v1.0.0/Developer-Package/SDK/environment-setup-cortexa7hf-neon-vfpv4-openstlinux\_weston-linux-gnueabi***  ***$OECORE\_NATIVE\_SYSROOT/usr/bin/arm-openstlinux\_weston-linux-gnueabi/arm-openstlinux\_weston-linux-gnueabi-gdb -x=$HOME/gdbscripts/Setup.gdb*** |

Gdb shows A7 stopped entry point of U-boot



* 1. **Debug an application**

See the doc Lab-DeveloperPackageVx.docx in [here](file://PRGCWD0579.prg.st.com/TOMAS_Share/MPU/STM32MP1_STFAE_Part2/stm32mp1-session4-DeveloperPackage)

“Step9 : STM32MP1 Developer Package Debug Linux application with gdb”

See:

[https://wiki.st.com/stm32mpu/wiki/GDB#Debug Linux application with gdbserver](https://wiki.st.com/stm32mpu/wiki/GDB#Debug_Cortex-M4_firmware_with_GDB)