Category:How to trace and debug

Contents

1. Category:How to trace and debug 3
2. How to access information in sysfs 4
3. How to check that a device tree resource is correctly set 8
4. How to debug Weston 8
5. How to detect memory leaks 11
6. How to diagnose a boot failure 11
7. How to enable earlyprintk for Linux kernel 15
8. How to find Linux kernel driver associated to a device 15
9. How to get DRM KMS logs 21
10. How to get Terminal 21
11. How to get name and current status of a DRM connector 24
12. How to monitor the GCNANO GPU load 25
13. How to monitor the display framerate 25
14. How to profile video framerate 26
15. How to read or write peripheral registers 36
16. How to retrieve Cortex-M4 logs after crash 44
17. How to use the kernel dynamic debug 48

Stable: 17.06.2020 - 15:26 / Revision: 12.12.2019 - 13:27

Category:How to trace and debug

Stable: 17.06.2020 - 10:20 | Revision: 12.12.2019 - 13:27

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

This category groups together articles explaining how to trace, monitor and debug software components for the STM32MPU Embedded Software distribution, and the STM32 MPUs microprocessor devices and boards.

Pages in category "How to trace and debug"

The following 16 pages are in this category, out of 16 total.

- How to access information in sysfs
- How to check that a device tree resource is correctly set
- How to debug Weston
- How to detect memory leaks
- How to diagnose a boot failure
- How to enable earlyprintk for Linux kernel
- How to find Linux kernel driver associated to a device
- How to get DRM KMS logs
- How to get name and current status of a DRM connector
- How to get Terminal
- How to monitor the display framerate
- How to monitor the GCNANO GPU load
- How to profile video framerate
- How to read or write peripheral registers
- How to retrieve Cortex-M4 logs after crash
- How to use the kernel dynamic debug

Stable: 24.01.2020 - 09:32 / Revision: 24.01.2020 - 09:31

A quality version of this page, approved on *24 January 2020*, was based off this revision.

Contents

1 Article purpose 5
2 Sysfs (/sys) pseudo filesystem 6
3 Sysfs usage 7
3.1 Example from Linux application 7
3.2 Example for shell command / bash script 7
4 References 8

1 Article purpose

This article provides some information about the sysfs pseudo filesystem usage from the user space.

2 Sysfs (/sys) pseudo filesystem

Sysfs provides a mean to export kernel data structures, their attributes, and the linkages between them to the user space.

Please refer to sysfs part of pseudo filesystem page.

3 Sysfs usage

Linux kernel provides a documentation[1] about the rules for sysfs usage.

Some examples are also described below with two different approaches for using sysfs entries from the user space:

- Linux application in C language
- bash script.

3.1 Example from Linux application

The below example is a typical sequence for using sysfs entry (here a PWM component):

- open a file descriptor of the sysfs entry file

```
10  len=snprintf(buf, sizeof(buf), "/sys/class/pwm/pwmchip0/pwm%d/duty_cycle", pwm_channel
);
11  fd = open(buf, O_RDWR);
```

- if fd is correctly opened, write/read value in the file: pay attention to the "text" format

```
12  if (fd < 0)
13  {
14      perror("pwm/duty_cycle");
15      return fd;
16  }
```

- read: store data to buffer

```
18  read(fd, buf, sizeof(buf));
```

- write: write data from buffer

```
20  len = snprintf(buf, sizeof(buf), "%d", 900000);
21  write(fd, buf, len);
```

- close file descriptor

```
30  close(fd);
```

3.2 Example for shell command / bash script

Operations on sysfs entries can be done by using command lines (i.e. *echo* for writing, *cat* for reading).

In this way, it is possible to use a bash script to execute a configuration sequence, similarly to what a user would do by typing multiple shell commands.

An example is provided in How_to_use_PWM_with_sysfs_interface.

4 References

- Documentation/admin-guide/sysfs-rules.rst

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Linux® is a registered trademark of Linus Torvalds.

Pulse Width Modulation

Stable: 04.02.2020 - 08:03 / Revision: 04.02.2020 - 07:55

A quality version of this page, approved on *4 February 2020*, was based off this revision.

Refer to UART issue scenario which provides examples for checking:

- Linux kernel driver probe
- Pins configuration
- Clocks configuration
- Interrupts configuration

Linux® is a registered trademark of Linus Torvalds.

Stable: 03.10.2019 - 13:43 / Revision: 03.10.2019 - 13:41

A quality version of this page, approved on *3 October 2019*, was based off this revision.

1 Wayland logs

To get the logs of the wayland protocol messages, set this environment variable:

```
Board $> export WAYLAND_DEBUG=1
```

2 Saving Weston logs in a file

Standard Weston logs are available in the **/var/log/weston.log** file when Weston is started using the systemd service.

```
Board $> cat /var/log/weston.log
```

3 Displaying Weston logs on a console

Start Weston with the following command:

```
Board $> weston --tty=1 &
```

All Weston logs are then displayed on the console.

Stable: 04.02.2020 - 08:03 / Revision: 04.02.2020 - 07:56

A quality version of this page, approved on *4 February 2020*, was based off this revision.

Refer to kmemleak article for Linux kernel space and to valgrind article for Linux user space.

Stable: 06.11.2020 - 16:10 / Revision: 06.11.2020 - 16:04

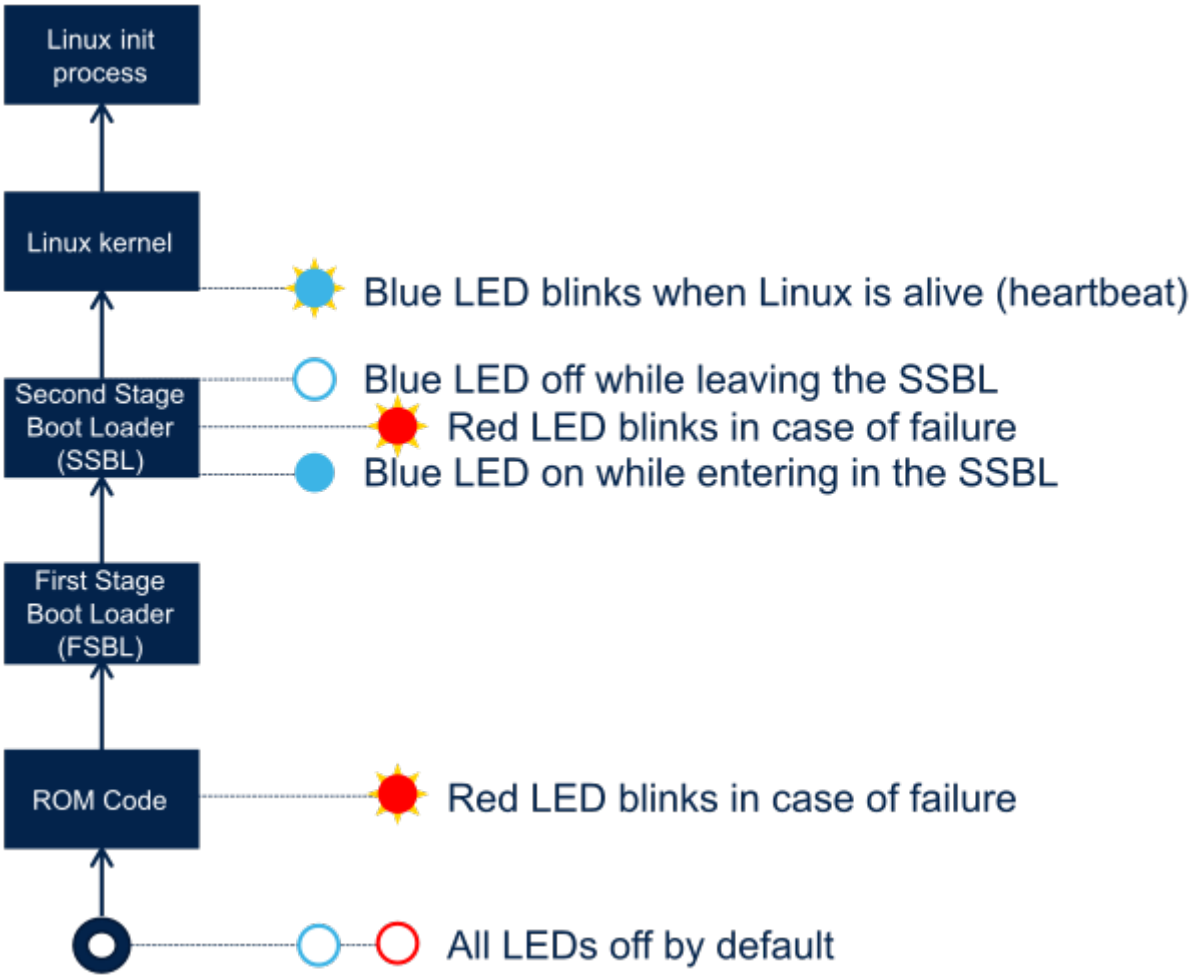
A quality version of this page, approved on *6 November 2020*, was based off this revision.

1 Introduction

The blue and red LEDs on STM32MP15 boards are used to notify the user in case of boot failure during the boot chain execution and at runtime.

2 Description and debug

The statuses of blue and red LEDs allow to see at which stage the execution failed: the diagram below shows how each boot component uses these LEDs while the table gives more information on the way to interpret the LED statuses when the boot fails. Among the boot components, the FSBL can be TF-A or U-Boot SPL and the SSBL is U-Boot, as explained in the boot chain overview.



Blue LED	Red LED	System state	Action
Off	Blinking	The execution failed during ROM code execution	Check that: • your microSD card is properly inserted in the board • the boot pins configuration selects the boot device • your boot memory was well programmed with STM32Cube Programmer
On	Blinking	The execution failed during the second stage bootloader (SSBL)	See U-Boot - How to debug to investigate the failure
Off	Off	The execution may have failed: 1. In FSBL execution 2. After the SSBL execution but before the heartbeat (blue LED) started	1. See TF-A - How to debug to investigate the failure 2. See U-Boot - How to debug and how to trace and debug to investigate the failure
Off or on	Off	The execution may have failed during Linux® kernel execution between two heartbeat pulses (blue LED): a fatal error leading to a kernel panic might have occurred...	See Dmesg and Linux kernel log to investigate the failure
Blinking	Off	Your platform is alive	Enjoy !

Light-emitting diode

First Stage Boot Loader

Trusted Firmware for Arm Cortex-A

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Secondary Program Loader, *Also known as* ***U-Boot SPL***

Second Stage Boot Loader

Stable: 03.02.2020 - 08:05 / Revision: 03.02.2020 - 08:01

A quality version of this page, approved on *3 February 2020*, was based off this revision.

Please refer to earlyprintk dedicated article.

Stable: 16.02.2021 - 16:09 / Revision: 16.02.2021 - 16:07

A quality version of this page, approved on *16 February 2021*, was based off this revision.

Contents

1 Introduction 16
2 Find kernel driver for a device 17
2.1 Major and minor numbers for a Linux kernel device 17
2.2 List of available devices 17
2.3 Device entries in /dev 18
2.4 System device entries in /sys/dev 18
2.5 Driver associated to a platform device 19
3 References 21

1 Introduction

This article shows the user how to find the Linux® kernel driver associated to a kernel device.

This can, for example, be useful when debugging devices that the user does not know, or monitoring for correct system behavior.

2 Find kernel driver for a device

2.1 Major and minor numbers for a Linux kernel device

The device files in the Linux kernel are associated to a MAJOR and a MINOR number, giving each file a unitary identity. This abstraction of device handling is a basic features of the Linux kernel.

A list of MAJOR numbers, and rules for MINOR numbers are given in *Documentation/admin-guide/devices.txt* of the Linux kernel source[1], or in kernel.org[2].

2.2 List of available devices

A list of the available devices for the Linux kernel can be read from the procfs file */proc/devices*:

```
Board $> cat /proc/devices
```

This lists all of the available devices, according to their classification as a character or a block device.
The number preceding the device name corresponds to the MAJOR number of the device (for example, "4" is the MAJOR number for the "tty" device):

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyRPMSG
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
153 spi
166 ttyACM
180 usb
189 usb_device
199 galcore
226 drm
245 cec
246 media
247 ttySTM
248 bsg
249 watchdog
250 iio
251 ptp
252 pps
```

```
253 rtc
254 gpiochip

Block devices:
  1 ramdisk
  7 loop
  8 sd
 11 sr
 31 mtdblock
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
254 virtblk
259 blkext
```

For further information about the major and minor numbers for a Linux kernel driver, refer to the Linux tutorial web page[3].

Note: 'Misc devices have a specific setup; you can find the list of misc devices with the corresponding MINOR number in the */proc/misc* file.

2.3 Device entries in /dev

Each device has a corresponding entry in the /dev directory of the Linux kernel pseudo filesystem.

```
Board $> ls -lR /dev
```

Be careful, /dev contains some sub-directories containing device entries, that is, *input*. That the reason why *-R* should be used.

This command lists all of the device entries, including the device type and the associated MAJOR and MINOR numbers

For example:

```
crw-rw---- 1 root video 81, 0 Dec 18 16:26 video0
```

This device *video0* is of type character (c), with MAJOR number of 81 and MINOR number of 0.

2.4 System device entries in /sys/dev

All devices, classified by type (char or block), and identified by their MAJOR/MINOR number can be found in the *dev* subdirectory of the sysfs file system entry (*/sys*).

A platform device is then linked to each MAJOR/MINOR number.

For example:

```
Board $> ls -l /sys/dev/char/81\:0
lrwxrwxrwx 1 root root 0 Dec 18 17:00 81:0 -> ../../devices/platform/soc/4c006000.dcmi
/video4linux/video0
```

The device *video0* is linked to the platform device *4c006000.dcmi/video4linux/video0*.

2.5 Driver associated to a platform device

If the device is linked to a platform device, you can find the corresponding driver definition in the device tree with the compatible parameter.

For example: Look for device 4c006000.dcmi/video4linux/video0 in arch/arm/boot/dts/stm32mp151.dtsi .

```
...
dcmi: dcmi@4c006000 {
 compatible = "st,stm32-dcmi";
 reg = <0x4c006000 0x400>;
 interrupts = <GIC_SPI 78 IRQ_TYPE_NONE>;
 resets = <&rcc CAMITF_R>;
 clocks = <&rcc DCMI>;
 clock-names = "mclk";
 dmas = <&dmamux1 75 0x400 0x05>;
 dma-names = "tx";
 status = "disabled";
};
...
```

The driver associated to the video0 device is ***st,stm32-dcmi***.

If the driver belongs to your Linux kernel tree, you can search for the driver by declaring st,stm32-dcmi as a compatible device.

- In the previous example, when looking for the driver compatible with *st,stm32-dcmi*, you find *drivers/media/platform/stm32/stm32-dcmi.c* driver

```
PC $> cd <your_kernel_source_path>
PC $> grep -rs "st,stm32-dcmi" *
...
drivers/media/platform/stm32/stm32-dcmi.c:        { .compatible = "st,stm32-dcmi"},
...
```

If the driver is not part of your Linux kernel source tree, it is present as a kernel object library file and you can check on the board:

```
Board $> cd /lib/modules/<kernel_version>
Board $> grep <compatible_name> modules.alias
```

This gives you the name of the module driver.

For example, for the gcnano driver used for the GPU:

```
Board $> grep "st,gcnano" modules.alias
 alias of:N*T*Cst,gcnano galcore
```

This means that the module name is *galcore.ko*.

3 References

- Documentation/admin-guide/devices.txt
- https://www.kernel.org/doc/Documentation/admin-guide/devices.txt
- http://www.linux-tutorial.info/?page_id=253

Linux® is a registered trademark of Linus Torvalds.

Process File System (See https://en.wikipedia.org/wiki/Procfs for more details)

System File System (See https://en.wikipedia.org/wiki/Sysfs for more details)

Generic Interrupt Controller

Serial Peripheral Interface

Digital Camera Memory Interface

Graphics Processing Units

Stable: 04.10.2019 - 17:42 / Revision: 04.10.2019 - 17:41

A quality version of this page, approved on *4 October 2019*, was based off this revision.

The full content of this article is available in the article named " Enable DRM/KMS traces with the sysfs".

Stable: 26.09.2019 - 12:46 / Revision: 26.09.2019 - 12:44

A quality version of this page, approved on *26 September 2019*, was based off this revision.

2 ways can be used to connect your PC to the board and then to control your board:

- through serial link (UART / USB)
- through network connection

Contents

1 Remote terminal via serial link (UART/USB) 22
1.1 On Linux® PC 22
1.2 On Windows® PC 23
2 Remote terminal over network 24

1 Remote terminal via serial link (UART/USB)

First of all, you need to connect your host PC to the board via UART.

Depending on the board, some additional materials may be needed to physically interconnect the board to the console serial port (extension board or adapter to interconnect the UART from the board to the USB PC input).

Pictures describing board connections can be found in Category:ST boards

1.1 On Linux® PC

On Linux, we recommend to use *Minicom*.

You need first to check that Minicom is installed on your PC.

- From a Terminal window on your PC:

```
PC $> minicom
```

- If a message indicating Minicom does not exist, then **install** it:

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

- To **configure** Minicom:

The UART configuration that is set with the below command has to match the one from your board: to be checked within board user manual.

```
PC $> sudo minicom -s
```

Hardware flow control is generally set to off (default value is on) and a baud rate of 115200 is usual.

- To **launch** Minicom:

```
PC $> minicom -D /dev/tty...
```

See How to use TTY with User Terminal article to find the appropriate tty instance to be used.

See below example for a standard UART (ttyS0) or STLink interface (ttyACM0):

```
PC $> minicom -D /dev/ttyS0
PC $> minicom -D /dev/ttyACM0
```

On the terminal, the prompt is changed to:

```
root@stm32mp1:~#
```

More information on Minicom can be found at the following link: https://help.ubuntu.com/community/Minicom.

1.2 On Windows® PC

Any of the Windows terminal emulator applications can be used. "Tera Term" is one of them: http://tera-term.en.lo4d.com/.
Then, the configuration is quite simple:

- Plug the cable and start the board
- Open a terminal emulator application on your PC and configure with the serial setup menu the USB port to be used and the baud rate of the serial link (in general 115200 baud)
- More information on the board serial port can generally be found in the hardware user manual **of the board**

The setup is correct when you can send Linux commands through the terminal emulator and get feedback. You can for example reboot the board and see the boot sequence.

2 Remote terminal over network

The board can also be accessed via the network through Ethernet connection, using ssh.

To do so, first **get the IP address** of the board:

- if it exists, using the board user interface (Terminal window or specific application)
- using a console (see chapter Remote terminal via Serial link (UART/USB)

If the network connection is through Ethernet then "InterfaceNetwork" = "Eth0", else if the network connection is using Wifi then "InterfaceNetwork" = "WLAN0"

```
Board $> ifconfig InterfaceNetwork
eth0      Link encap:Ethernet  HWaddr xx:xx:xx:xx:xx:FB
          inet addr:xxx.xxx.xxx.xxx <ip address>  Bcast:xxx.xxx.xxx.xxx Mask:xxx.xxx.xxx.
xxx
          inet6 addr: fe80::280:e1ff:fe01:14fb/64 Scope:Link
```

Then, from another terminal on a remote PC connected to the same network, use the command described in How to perform ssh connection.

Universal Asynchronous Receiver/Transmitter

Linux® is a registered trademark of Linus Torvalds.

ST in-circuit debugger and programmer for the STM8 and STM32 microcontroller families (See ST-LINK for more details)

Frame Buffer (could be the Kernel framebuffer linked to the display, a GPU framebuffer, an imaging framebuffer...)

Stable: 07.10.2019 - 09:33 / Revision: 07.10.2019 - 09:32

A quality version of this page, approved on *7 October 2019*, was based off this revision.

Use the following command to get the DRM connector names and associated status:

```
Board $> for p in /sys/class/drm/*/status; do con=${p%/status}; echo -n "${con#*/card?-}:
"; cat $p; done
```

Result example:

```
DSI-1: connected
HDMI-A-1: connected
```

Direct Rendering Manager (kernel module that gives direct hardware access to DRI clients, find more information on official DRI web site http://dri.freedesktop.org/wiki/DRM)

Display Serial Interface (MIPI® Alliance standard)

High-Definition Multimedia Interface (HDMI standard)

Stable: 26.02.2021 - 14:28 / Revision: 11.01.2021 - 17:25

A quality version of this page, approved on *26 February 2021*, was based off this revision.

When a GPU animation is running on the display, the related GCNANO estimated GPU load can be monitored from the GCNANO driver level, by using the following command:

```
Board $> (while true; do
gpu1=$(cat /sys/kernel/debug/gc/idle); \
sleep 4; \
gpu2=$(cat /sys/kernel/debug/gc/idle); \
echo $gpu1 $gpu2 | tr -d '\n' | tr -d ',' | tr -d 'ns' | awk -F" " '{printf("gpu load %.
0f%%\n", ($10-$2)*100/($10+$12+$14+$16-$2-$4-$6-$8))}'; \
done) &
```

The GCNANO estimated GPU load is then periodically output in the user console as a percentage "%":

```
gpu load 75%
gpu load 75%
gpu load 75%
```

Notes:

- Stop monitoring the GPU load with the command "**kill -9 `ps -o ppid= -C sleep`**".
- Adjust the GPU load update period by modifying the "sleep" value (4 seconds in the example).
- Use the command "dmesg -n8" to mix both user and kernel console outputs.
- Debugfs configuration needs to be enabled.
- The detailed calculation is: GPU load = (On-previous_On) / (Total-previous_Total) with Total=On+Off+Idle+Suspend, all variables coming from the command:

```
Board $> cat /sys/kernel/debug/gc/idle
On:                2,071,009,284,477 ns
Off:              11,480,071,864,263 ns
Idle:                              0 ns
Suspend:           1,242,043,838,898 ns
...
```

Graphics Processing Units

Stable: 16.01.2020 - 15:11 / Revision: 16.01.2020 - 15:07

A quality version of this page, approved on *16 January 2020*, was based off this revision.

When an animation is running on the display, the related framerate can be monitored from the display driver level thanks to the command:

```
Board $> (while true; do export fps=`cat /sys/kernel/debug/dri/0/state | grep fps -m1 |
grep -o '[0-9]\+'`; echo display ${fps}fps; sleep 4; done) &
```

The display framerate is then periodically output in the user console in "fps" (**f**rames **p**er **s**econd):

```
display 50fps
display 50fps
display 50fps
```

Notes:

- Stop monitoring the framerate with the command "kill -9 `ps -o ppid= -C sleep`".
- Adjust the framerate update period by modifying the "sleep" value (4 seconds in the example).
- Use the command "dmesg -n8" to mix both user and kernel console outputs.
- Debugfs configuration needs to be enabled.

Stable: 24.09.2019 - 09:53 / Revision: 24.09.2019 - 09:52

A quality version of this page, approved on *24 September 2019*, was based off this revision.

This article aims to debug & profile framerate performances of any GStreamer video use-case, including camera preview or video playback use-cases.

Contents

1 Debugging framerate issues 27
1.1 Framerate of played content 27
1.2 Display framerate 27
1.3 Check default traces 28
1.4 Frame drop traces 29
1.5 Frame drop due to display subsytem 29
2 Profiling framerate: fpsdisplaysink 31
3 Disabling frame synchronisation 35

1 Debugging framerate issues

A variety of symptoms related to framerate issues may be observed such as:

- Jerky video
- Video freeze
- Excessively slow Video framerate (typically one frame per second, see below)
- Too slow or too fast video motion
- Audio & video not synchronized
- ...

When such symptoms are observed, one can check below chapters to ease investigations and analysis of the problems.

1.1 Framerate of played content

Before investigating possible framerate issues, check the expected multimedia content framerate.

For a video, refer to gst-discoverer to get the video file framerate:

```
Board $> gst-discoverer-1.0 <my video> -v | grep -i "Frame rate"
     Frame rate: 30/1
```

For this video, the framerate is 30 fps (**f**rames **p**er **s**econd).

For a camera preview use-case, the framerate is set in the pipeline:

```
Board $> gst-launch-1.0 v4l2src ! "video/x-raw, width=1280, Height=720, framerate=
(fraction)15/1" ! queue ! autovideosink -e
```

Here the expected framerate is 15 fps.

1.2 Display framerate

When an animation is running on the display, the related framerate can be monitored from the display driver level thanks to the command:

```
Board $> (while true; do export fps=`cat /sys/kernel/debug/dri/0/state | grep fps -m1 |
grep -o '[0-9]\+'`; echo display ${fps}fps; sleep 4; done) &
```

The display framerate is then periodically output in the user console in "fps" (**f**rames **p**er **s**econd):

```
display 50fps
display 50fps
display 50fps
```

Notes:

- Stop monitoring the framerate with the command "kill -9 `ps -o ppid= -C sleep`".

- Adjust the framerate update period by modifying the "sleep" value (4 seconds in the example).
- Use the command "dmesg -n8" to mix both user and kernel console outputs.
- Debugfs configuration needs to be enabled.

It should conform to the expected framerate for the played content. If it is not the case, continue investigations with next chapters.

1.3 Check default traces

By default, the traces show Warnings when the GStreamer video sink receives a **lot of late buffers**:

```
WARNING: from element /GstPipeline:pipeline0/[...] A lot of buffers are being dropped.
```

In that case, GStreamer falls into a recovery mode consisting into displaying **one frame every second**.

If such warning is displayed, it means that some elements before the video sink are not fast enough to sustain the targeted framerate.

Here is a test pipeline illustrating this behaviour:

```
Board $> gst-launch-1.0 videotestsrc ! "video/x-raw, framerate=(fraction)200/1" !
autovideosink
```

```
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
WARNING: from element /GstPipeline:pipeline0/GstWaylandSink:waylandsink0: A lot of
buffers are being dropped.
Additional debug info:
../../../../git/libs/gst/base/gstbasesink.c(2901): gst_base_sink_is_too_late ():
/GstPipeline:pipeline0/GstWaylandSink:waylandsink0:
There may be a timestamping problem, or this computer is too slow.
WARNING: from element /GstPipeline:pipeline0/GstWaylandSink:waylandsink0: A lot of
buffers are being dropped.
Additional debug info:
../../../../git/libs/gst/base/gstbasesink.c(2901): gst_base_sink_is_too_late ():
/GstPipeline:pipeline0/GstWaylandSink:waylandsink0:
There may be a timestamping problem, or this computer is too slow.
WARNING: from element /GstPipeline:pipeline0/GstWaylandSink:waylandsink0: A lot of
buffers are being dropped.
```

This test pipeline generates a pattern at 200fps which is not sustainable by the overall system, leading to frames coming to the video sink very late.

Warning

This trace only appears if *lot of frames* are dropped, see following chapters for finer grain analysis.

1.4 Frame drop traces

Fine grain information can be provided around frame drops by enabling video sink traces:

```
--gst-debug=basesink:6 2>&1 | grep drop
```

One trace message will be output for each frame dropped.

With the test pipeline:

```
Board $> gst-launch-1.0 -e videotestsrc ! "video/x-raw, framerate=(fraction)100/1" !
autovideosink --gst-debug=basesink:6 2>&1 | grep drop
```

```
0:00:01.135448709   585   0x1a5a60 DEBUG               basesink gstbasesink.c:3626:
gst_base_sink_chain_unlocked:<autovideosink0-actual-sink-wayland> buffer late, dropping
0:00:01.147437126   585   0x1a5a60 DEBUG               basesink gstbasesink.c:3626:
gst_base_sink_chain_unlocked:<autovideosink0-actual-sink-wayland> buffer late, dropping
0:00:01.157521667   585   0x1a5a60 DEBUG               basesink gstbasesink.c:3626:
gst_base_sink_chain_unlocked:<autovideosink0-actual-sink-wayland> buffer late, dropping
0:00:01.168611
```

If no traces are observed, the GStreamer synchronisation system has not dropped any frame.

Warning

Even if no drop is observed with this trace, frames could nevertheless be dropped by the video sink GStreamer element because of the display subsystem, see next chapter for more details.

.

1.5 Frame drop due to display subsytem

Frames could also be dropped by the video sink GStreamer element because of the display subsystem not being fast enough to sustain the incoming framerate.

This frame dropping strategy is specific to the selected video sink GStreamer element.

Here is a trace that can be enabled to show when wayland subsystem cannot sustain the incoming framerate, and consequently, drop frames:

```
 --gst-debug=waylandsink:6 2>&1 | grep -i "redraw pending"
```

Here is an example:

```
Board $> gst-launch-1.0 -v -e videotestsrc ! video/x-raw, format=I420, framerate=100/1 !
queue ! fpsdisplaysink sync=false video-sink=waylandsink -v --gst-debug=waylandsink:6
2>&1 | grep  -e "redraw pending" -e "dropped"
```

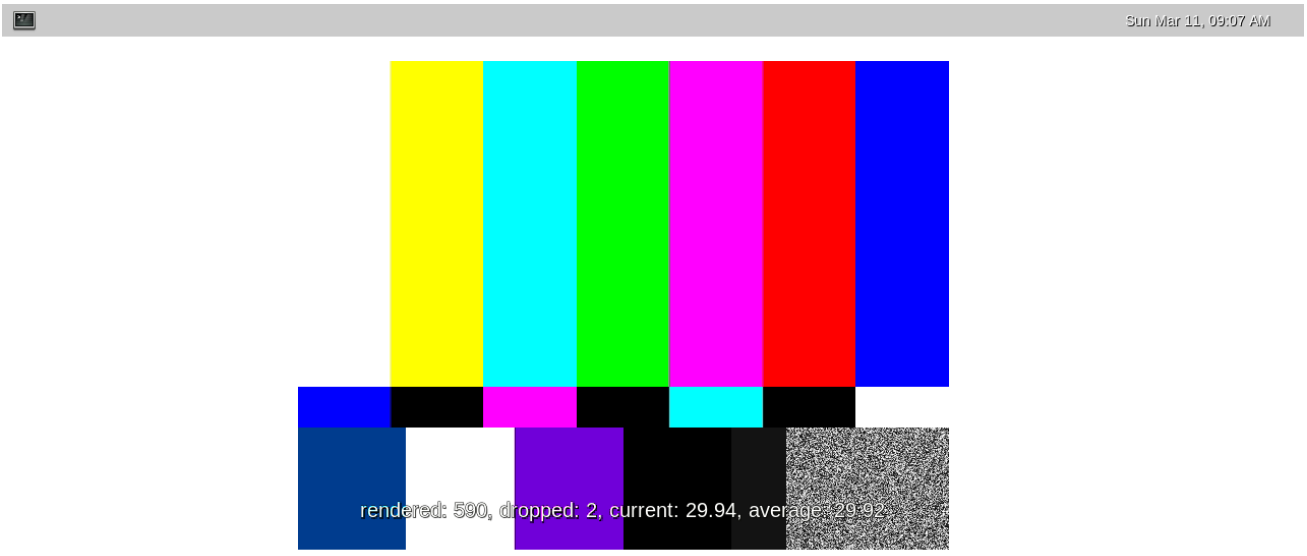
```
0:00:00.392392792  1020   0x19dd50 LOG              waylandsink gstwaylandsink.c:822:
gst_wayland_sink_show_frame:<waylandsink0> buffer 0xb510d860 dropped (redraw pending)
0:00:00.392863376  1020   0x19dd50 LOG              waylandsink gstwaylandsink.c:822:
gst_wayland_sink_show_frame:<waylandsink0> buffer 0xb510d900 dropped (redraw pending)
0:00:00.394748751  1020   0x19dd50 LOG              waylandsink gstwaylandsink.c:822:
gst_wayland_sink_show_frame:<waylandsink0> buffer 0xb510d9a0 dropped (redraw pending)
0:00:00.422420584  1020   0x19dd50 LOG              waylandsink gstwaylandsink.c:822:
gst_wayland_sink_show_frame:<waylandsink0> buffer 0xb510d900 dropped (redraw pending)
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 132, dropped: 0, current: 76.90, average: 87.14
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 132,
dropped: 0, current: 76.90, average: 87.14
```

In this example, the frames are generated fast enough to not be dropped because of their lateness, see trace message **"dropped: 0"**. This is the display subsystem that is responsible of not rendering frames fast enough, see trace message **"dropped (redraw pending)"**.

2 Profiling framerate: fpsdisplaysink

The framerate measurement can be done using **fpsdisplaysink** GStreamer element.

The measure is shown directly on the screen on a display overlay:



Available information:

- the number of rendered frames
- the number of dropped frames
- the current framerate
- the average framerate

fpsdisplaysink can replace any existing video sink into a GStreamer pipeline. To do so, replace:

```
gst-launch-1.0 [...] ! <current video sink>
```

with:

```
gst-launch-1.0 [...] ! fpsdisplaysink video-sink=<current video sink>
```

This could also be done with high level GStreamer bins such as **playbin**. To do so, replace:

```
gst-launch-1.0 playbin [...] video-sink=<current video sink>
```

with:

```
gst-launch-1.0 playbin [...] video-sink="fpsdisplaysink video-sink=<current video sink>"
```

Same can be done for high level GStreamer utility such as gst-play. To do so, replace:

```
gst-play-1.0 [...]
```

with:

```
gst-play-1.0 [...] --videosink="fpsdisplaysink video-sink=autovideosink"
```

Information could also be displayed in the console using the GStreamer "-v" verbose option. Here is a test pipeline illustrating this behaviour:

```
Board $> gst-launch-1.0 videotestsrc ! "video/x-raw, width=640, height=480, framerate=
(fraction)30/1" ! fpsdisplaysink video-sink=autovideosink -v
```

```
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 618, dropped: 3, fps: 25.04, drop rate: 1.93
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 618,
dropped: 3, fps: 25.04, drop rate: 1.93
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 629, dropped: 10, fps: 20.52, drop rate: 13.06
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 629,
dropped: 10, fps: 20.52, drop rate: 13.06
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 644, dropped: 10, current: 29.75, average: 29.55
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 644,
dropped: 10, current: 29.75, average: 29.55
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 660, dropped: 10, current: 30.19, average: 29.57
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 660,
dropped: 10, current: 30.19, average: 29.57
```

More options are also available on fpsdisplaysink, refer to the help:

```
Board $> gst-inspect-1.0 fpsdisplaysink
```

```
Factory Details:
  Rank                     none (0)
  Long-name                Measure and show framerate on videosink
  Klass                    Sink/Video
  Description              Shows the current frame-rate and drop-rate of the videosink as
overlay or text on stdout
  Author                   Zeeshan Ali <zeeshan.ali@nokia.com>, Stefan Kost <stefan.
kost@nokia.com>

Plugin Details:
  Name                     debugutilsbad
  Description              Collection of elements that may or may not be useful for
debugging
  Filename                 /usr/lib/gstreamer-1.0/libgstdebugutilsbad.so
```

```
  Version                  1.12.3
  License                  LGPL
  Source module            gst-plugins-bad
  Source release date      2017-09-18
  Binary package           GStreamer Bad Plug-ins source release
  Origin URL               Unknown package origin

GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstBin
                         +----GstFPSDisplaySink

Implemented Interfaces:
  GstChildProxy

Pad Templates:
  SINK template: 'sink'
    Availability: Always
    Capabilities:
      ANY


Element Flags:
  no flags set

Bin Flags:
  no flags set

Element Implementation:
  Has change_state() function: 0xb6a63d60

Element has no clocking capabilities.
Element has no URI handling capabilities.

Pads:
  SINK: 'sink'

Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: "fpsdisplaysink0"
  parent              : The parent of the object
                        flags: readable, writable
                        Object of type "GstObject"
  async-handling      : The bin will handle Asynchronous state changes
                        flags: readable, writable
                        Boolean. Default: false
  message-forward     : Forwards all children messages
                        flags: readable, writable
                        Boolean. Default: false
  sync                : Sync on the clock (if the internally used sink doesn't have this
property it will be ignored
                        flags: readable, writable
                        Boolean. Default: true
  text-overlay        : Whether to use text-overlay
                        flags: readable, writable
                        Boolean. Default: true
  video-sink          : Video sink to use (Must only be called on NULL state)
                        flags: readable, writable
                        Object of type "GstElement"
  fps-update-interval : Time between consecutive frames per second measures and update
(in ms). Should be set on NULL state
                        flags: readable, writable
                        Integer. Range: 1 - 2147483647 Default: 500
  max-fps             : Maximum fps rate measured. Reset when going from NULL to READY.-1
```

```
means no measurement has yet been done
                        flags: readable
                        Double. Range:               -1 -   1.797693e+308
Default:                -1
  min-fps               : Minimum fps rate measured. Reset when going from NULL to READY.-1
means no measurement has yet been done
                        flags: readable
                        Double. Range:               -1 -   1.797693e+308
Default:                -1
  signal-fps-measurements: If the fps-measurements signal should be emited.
                        flags: readable, writable
                        Boolean. Default: false
  frames-dropped        : Number of frames dropped by the sink
                        flags: readable
                        Unsigned Integer. Range: 0 - 4294967295 Default: 0
  frames-rendered       : Number of frames rendered
                        flags: readable
                        Unsigned Integer. Range: 0 - 4294967295 Default: 0
  silent                : Don't produce last_message events
                        flags: readable, writable
                        Boolean. Default: false
  last-message          : The message describing current status
                        flags: readable
                        String. Default: null

Element Signals:
  "fps-measurements" :  void user_function (GstElement* object,
                                            gdouble arg0,
                                            gdouble arg1,
                                            gdouble arg2,
                                            gpointer user_data);
```

3 Disabling frame synchronisation

The GStreamer frame dropping mechanism (due to frame lateness) can be disabled using option "sync=false" applied on the video sink.

When frame dropping is disabled, all frames received by the video sink are sent to the display subsystem without any timestamp check.

In this case, the maximum framerate sustainable by the system can be reached.

Here are some typical GStreamer pipelines where video frame synchronization has been disabled:

```
Board $> gst-play-1.0  --videosink="autovideosink sync=false"
```

```
Board $> gst-play-1.0  --videosink="waylandsink sync=false"
```

```
Board $> gst-launch-1.0 playbin ...  video-sink="waylandsink sync=false"
```

```
Board $> gst-launch-1.0 filesrc ... ! waylandsink sync=false
```

```
Board $> gst-launch-1.0 filesrc ... ! kmssink sync=false
```

Using fpsdisplaysink with "sync=false" option allows to get the maximum sustainable framerate value.

Here is an example:

```
Board $> gst-launch-1.0 videotestsrc ! "video/x-raw, width=640, height=480, framerate=
(fraction)100/1" ! fpsdisplaysink sync=false video-sink="autovideosink" -v
```

```
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 51, dropped: 0, current: 22.33, average: 24.67
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 51,
dropped: 0, current: 22.33, average: 24.67
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 63, dropped: 0, current: 22.83, average: 24.30
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 63,
dropped: 0, current: 22.83, average: 24.30
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0/GstTextOverlay:fps-display-text-
overlay: text = rendered: 75, dropped: 0, current: 22.93, average: 24.07
/GstPipeline:pipeline0/GstFPSDisplaySink:fpsdisplaysink0: last-message = rendered: 75,
dropped: 0, current: 22.93, average: 24.07
```

In the above example, the maximum framerate is around 23fps while target is 100fps.

fourcc of YUV420 planar pixel format

Stable: 28.07.2020 - 14:30 / Revision: 28.07.2020 - 14:22

A quality version of this page, approved on *28 July 2020*, was based off this revision.

Contents

1 Article purpose 37
2 Introduction 38
3 Installing Devmem on your target board 39
3.1 Using the STM32MPU Embedded Software distribution 39
3.1.1 Starter Package 39
3.1.2 Developer Package 39
3.1.3 Distribution Package 39
4 Getting started 41
5 To go further 42
5.1 Checking UART4 baud rate 42
5.2 Driving LEDs with Devmem 42
6 Restricted access 44

1 Article purpose

This article provides the basic information needed to start using the Devmem tool, which allows the reading and writing of peripheral registers.

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
Devmem	Tracing tools	Used for reading and writing in peripheral registers	✓	✓	✓	✗	✗	✗

3 Installing Devmem on your target board

You can install Devmem either with the Starter Package, the Developer Package or the Distribution Package. For the Starter Package, board internet access is required.

3.1 Using the STM32MPU Embedded Software distribution

3.1.1 Starter Package

Enter the following commands to install Devmem:

```
Board $> sudo apt-get update
Board $> sudo apt-get install devmem2
```

For more information about apt-get, see the Package_repository_for_OpenSTLinux_distribution article.

3.1.2 Developer Package

Enter the following commands:

- Download sources :

```
PC $> wget http://free-electrons.com/pub/mirror/devmem2.c
```

- Please ensure that the SDK environment for compiling Linux application setup is ready
- Compile the application :

```
PC $> make devmem2
```

- Install it on the board:

```
PC $> scp devmem2 root@<board ip address>:/usr/bin
```

3.1.3 Distribution Package

Enter the following commands:

- Build the package with bitbake:

```
PC $> bitbake devmem2
```

- Add it to the targeted image:

```
PC $> echo 'IMAGE_INSTALL_append += "devmem2"' >> meta-st/meta-st-openstlinux/recipes-st
/images/st-image-weston.bbappend
```

- Rebuild the image:

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

You must then Flash the generated image onto your board using STM32CubeProgrammer.

4 Getting started

To use Devmem, enter the following command:

```
Board $> devmem2 [address] [type] [data]
```

Where:

- [address] corresponds to the register address you want to read
- [type] corresponds to the desired output value size (b, h, w or l, respectively byte, halfword, word or long). Assigning a type is not necessary if you only read the register - the default output type is a word.
- [data] corresponds to the data value you wish to write to the register, assigning a type is mandatory in this case . If data is not assigned, the register is only read.

5 To go further

5.1 Checking UART4 baud rate

The UART baud rate is generated from the UART clock frequency, with the USARTDIV parameter field in the USART_BRR register. By default in the OpenSTLinux Starter Package delivery, the UART4 baud rate is set to 115 000, the clock frequency CLOCK_UART4_K value is 64 000 000 Hz and USARTDIV is equal to 556. These values are known, and the USARTDIV value is retrieved by using Devmem to read the STM32MP1 registers.

On the STM32MP157 Reference Manual, we see that the base address of UART4 is 0x4001 0000 and the USARTDIV value is located at offset 0x0C.

```
Board $> devmem2 0x4001000C
/dev/mem opened.
Memory mapped at address 0xb6fc300c.
Read at address 0x4001000C (0xb6fc300c): 0x0000022C
```

This is equal to 556 decimal.

5.2 Driving LEDs with Devmem

In this example, we check the output value of GPIO port A pins 13 and 14 corresponding to LEDs 6 and 5 on DK2. According to the Reference Manual, the base address for GPIOA is 0x5000 2000 and the output value of the LEDs is located at the offset 0x14.

However, the GPIO clock must first be enabled for the results to be readable from the registers. Otherwise the resulting output value is 0:

```
Board $> devmem2 0x50002014
/dev/mem opened.
Memory mapped at address 0xb6f43014.
Read at address 0x50002014 (0xb6f43014): 0x00000000
```

The clock is named RCC_MP_AHB4ENSETR and is located at address 0x5000 0A28.

```
Board $> devmem2 0x50000A28
/dev/mem opened.
Memory mapped at address 0xb6fdca28.
Read at address 0x50000A28 (0xb6fdca28): 0x00000000
```

In this register, the first eleven bits (from LSB to MSB) allow GPIOA to GPIOK respectively to be enabled (1 enable, 0 disable). In this case only GPIOA needs to be enabled, so we write 0x1 to the register.

```
Board $> devmem2 0x50000A28 w 0x1
/dev/mem opened.
Memory mapped at address 0xb6fdca28.
Read at address 0x50000A28 (0xb6fdca28): 0x00000000
Write at address 0x50000A28 (0xb6fdca28): 0x00000001, read back 0x00000001
```

We now check the content of the register corresponding to GPIOA:

```
Board $> devmem2 0x50002014
/dev/mem opened.
Memory mapped at address 0xb6f43014.
Read at address 0x50002014(0xb6f43014): 0x00002400
```

In this case only LED 5 is enabled. The following table shows the register output value as a function of LED 5 and 6 states.

	LED 5 enabled	LED 5 disabled
LED 6 enabled	0x400	0x4400
LED 6 disabled	0x2400	0x6400

For example to enable both LEDs, set the value of the register to 0x400:

```
Board $> devmem2 0x50002014 w 0x400
/dev/mem opened.
Memory mapped at address 0xb6f43014.
Read at address 0x50002014 (0xb6f43014): 0x00002400
Write at address 0x50002014 (0xb6f43014): 0x00000400, readback 0x00000400
```

The change can be seen on your board.

6 Restricted access

Keep in mind that the Devmem tool only allows access to peripheral registers. Also all the registers in M4 and A7 cores are secured and attempting to read them causes the board to reboot.

When trying to access a register, make sure the related peripheral has not been isolated on the MCU side, otherwise access is impossible due to access rights.

Universal Asynchronous Receiver/Transmitter

Universal Synchronous/Asynchronous Receiver/Transmitter

General-Purpose Input/Output (A realization of open ended transmission between devices on an embedded level. These pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and applications requirements.)

Reset and Clock Control

Light-emitting diode

Microcontroller Unit (MCUs have internal flash memory and are intended to operate with a minimum amount of external support ICs. They commonly are a self-contained system-on-chip (SoC) designs.)

Stable: 03.02.2020 - 08:41 / Revision: 03.02.2020 - 08:27

A quality version of this page, approved on *3 February 2020*, was based off this revision.

Contents

1 Introduction 45
2 Prerequisite 46
3 Getting an M4 coredump file 47
4 Reading the logs from the coredump file 48

1 Introduction

When the Arm®Cortex®-M4 firmware crashes, Linux® running on Arm®Cortex®-A7, can generate an M4 coredump file. This M4 coredump records the states of the working memory of the Cortex-M4 firmware which permits to assist in diagnosing errors.

If some Cortex-M4 firmware logs have been output to the trace buffer declared in the resource table, they can be retrieved from the M4 coredump.

2 Prerequisite

An M4 coredump file is generated under the following conditions:

- the Cortex-M4 informs the Cortex-A7 when it crashes. This is the case for instance if the Cortex-M4 firmware implements the watchdog detection (see WWDG internal peripheral),
- the remoteproc node of the Linux device tree defines the *recovery* property,
- a Linux service allowing to capture coredump files is running. In the STM32MPU Embedded Software distribution this is implemented by the *m4-dump.rules* udev rule.

3 Getting an M4 coredump file

When the Cortex-M4 crashes, Linux running on the Cortex-A7 is informed and stores a timestamped M4 coredump file in /var /crash.

Example: listing M4 coredumps

```
Board $> ls -l /var/crash
total 1188
-rw-r--r-- 1 root root 608500 Nov 30 16:02 m4-fw-error_2018-11-30_16-02-21.dump
-rw-r--r-- 1 root root 606420 Nov 30 16:07 m4-fw-error_2018-11-30_16-07-26.dump
```

4 Reading the logs from the coredump file

Start with getting the offset address of the trace log:

```
PC $> readelf -l m4-fw-error_2018-11-30_16-02-21.dump

Elf file type is CORE (Core file)
Entry point 0x10003129
There are 6 program headers, starting at offset 52

Program Headers:
 Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 LOAD           0x0000f4 0x00000000 0x00000000 0x10000 0x10000 RWE 0
 LOAD           0x0100f4 0x30000000 0x30000000 0x40000 0x40000 RWE 0
 LOAD           0x0500f4 0x10000000 0x10000000 0x40000 0x40000 RWE 0
 LOAD           0x0900f4 0x10040000 0x10040000 0x02000 0x02000 RWE 0
 LOAD           0x0920f4 0x10042000 0x10042000 0x02000 0x02000 RWE 0
 LOAD           0x0940f4 0x100200e4 0x100200e4 0x00800 0x00800 RWE 0
```

The trace log (assuming that the resource table defines a trace buffer) is stored in the last memory segment. In this example, its offset is 0x0940f4.

You can read the M4 binary coredump file from this offset to get the logs or you can use the following command to output them (use **-c +N** option where N is *log offset + 1*):

```
PC $> tail -c +$((0x940f4 + 1)) m4-fw-error_2018-11-30_16-02-21.dump
[00000.000][INFO ]Starting WWDG
[00001.422][INFO ]WWDG now!
```


Stable: 02.11.2020 - 10:48 / Revision: 19.10.2020 - 12:09

A quality version of this page, approved on *2 November 2020*, was based off this revision.

Contents

1 Introduction 49
2 More technical information 50
3 Examples 51
4 Synchronous tracing on the console 52
5 Debug messages during boot process 53
6 References 54

1 Introduction

As prerequisite to reading this article, please refer to the Dmesg and Linux kernel log page.

"Dynamic debug is designed to allow you to dynamically enable/disable kernel code to obtain additional kernel information. Currently, if **CONFIG_DYNAMIC_DEBUG** is set, all **pr_debug()/dev_dbg()** calls can be dynamically enabled per-callsite." extracted from the Linux kernel documentation[1].

The related debugfs entry is usually:

```
/sys/kernel/debug/dynamic_debug/control
```

Note that the verbose **dev_vdbg()** calls cannot be dynamically activated.

When the dynamic debug traces are activated, the trace results are printed in **dmesg (or /proc/kmsg)**, and in the console if console loglevel is set to 8.

2 More technical information

The dynamic debug trace configuration is done through a **control** file in the **debugfs** filesystem: *<debugfs>/dynamic_debug /control*

The command includes keywords and flag elements (for details see the Linux kernel documentation[1]).

- Keywords

Possible keywords are:

```
func : function name
file : source filename
module : module name
format : format string
line : line number (including ranges of line numbers)
```

The colored keywords above are illustrated by examples in the next chapter.

- Flags

The flag specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

```
- : remove the given flags
+ : add the given flags
= : set the flags to the given flags
```

Possible flags are:

```
f : Include the function name in the printed message
l : Include line number in the printed message
m : Include module name in the printed message
p : Causes a printk() message to be emitted to dmesg
t : Include thread ID in messages not generated from interrupt context
```

3 Examples

- Track all dev_*dbg/pr_debug() in a **file** (you can add several files if necessary):

```
Board $> mount -t debugfs none /sys/kernel/debug
Board $> echo "file stm32-adc.c +p" > /sys/kernel/debug/dynamic_debug/control
```

Note that just the file name or full file path can be given, here *stm32-adc.c* or *drivers/iio/adc/stm32-adc.c*

- Track only one **line** with dev_dbg() in a **file** (you can add several files and several lines if necessary, please use the last line number of the function call):

```
Board $> echo "file stm32-adc.c line 1438 +p" > /sys/kernel/debug/dynamic_debug/control
```

- For an entire "**module** (module means ~.ko, so not applicable for a statically linked driver)":

```
Board $> echo "module cfg80211 +p" > /sys/kernel/debug/dynamic_debug/control
```

- If you want to list all available traces *(warning: it is a long file so you may need to use "tee" or another solution to save it)*:

```
Board $> cat /sys/kernel/debug/dynamic_debug/control | tee /tmp/dynamic_log.log
```

- For instance, if you are looking for a particular **file** to find a particular **line**:

```
Board $> cat /sys/kernel/debug/dynamic_debug/control | grep adc
drivers/iio/adc/stm32-adc.c:1515 [stm32_adc]stm32_adc_conf_scan_seq =p "%s chan %d to %s%
d\012"
drivers/iio/adc/stm32-adc.c:1438 [stm32_adc]stm32_adc_awd_set =p "%s chan%d htr:%d ltr:%
d\012"
drivers/iio/adc/stm32-adc.c:2182 [stm32_adc]stm32_adc_dma_start =p "%s size=%d watermark=%
d\012"
drivers/iio/adc/stm32-adc.c:2304 [stm32_adc]stm32_adc_trigger_handler =p "%s bufi=%d\012"
drivers/iio/adc/stm32-adc.c:2443 [stm32_adc]stm32_adc_chan_of_init =p "Configured to use
injected\012"
drivers/iio/adc/stm32-adc.c:2364 [stm32_adc]stm32_adc_of_get_resolution =p "Using %u bits
resolution\012"
```

- Multiple commands can be written together, separated by ';' or '\n'.

```
Board $> echo "file stm32-adc.c +p; file stm32-adc-core.c +p" > /sys/kernel/debug
/dynamic_debug/control
```

- A another method is to use a wildcard. The match rule supports * (matches zero or more characters) and ? (matches exactly one character). For example, you can match all USB drivers:

```
Board $> echo "file drivers/usb/* +p" > /sys/kernel/debug/dynamic_debug/control
```

4 Synchronous tracing on the console

In the case of a crash, or impossibility to call dmesg, it is sometimes useful to have traces synchronously emitted on the console.
Only error, warning and informational traces are emitted synchronously on the console (that is, loglevel=5), so if you need to see the lower level traces too, you need to change the console loglevel to "8".

```
<enable the conditional traces>
Board $> echo 8 > /proc/sys/kernel/printk
or
Board $> dmesg -n 8
or
Board $> dmesg -n debug
```

Please follow this article to get a serial console for the target: How to get Terminal

Warning

As all traces are now synchronously emitted, real-time is affected

If you want to return to the default console log level, you have to get this default value from the procfs entry */proc/sys /kernel/printk*:

```
Board $> cat /proc/sys/kernel/printk
8       4       1       7
Board $> dmesg -n 7
Board $> cat /proc/sys/kernel/printk
7       4       1       7
```

5 Debug messages during boot process

In order to activate debug messages during the boot process, even before userspace and debugfs exist, use the kernel's command-line parameter: **dyndbg**

For instance, the kernel *bootargs* can be modified 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 [2], or introduction page[3]). For example:

```
Board $> mount /dev/mmcblk0p4 /boot
Board $> vi /boot/mmc0_stm32mp157c-ev1_extlinux/extlinux.conf
```

or

- Edit the extlinux.conf file by using UMS (USB Mass Storage): see How to use USB mass storage in U-Boot for details.

To mount partitions (mmc 0:microSD card / mmc 1: eMMC):

- Press any key to stop at U-Boot execution when booting the board.

```
Board $> ...
Board $> Hit any key to stop autoboot:  0
Board $> STM32MP>
```

- Then

```
STM32MP> ums 0 mmc 0
```

- Check for the boot partition mounted on your host PC (/media/*$USER*/bootfs)
- Edit the extlinux file corresponding to your setup (/media/*$USER*/bootfs/mmc0_extlinux/stm32mp157f-dk2_extlinux.conf

- Update the kernel command line, adding the dyndbg parameter:

```
root=PARTUUID=e91c4e10-16e6-4c0e-bd0e-77becf4a3582 rootwait rw console=ttySTM0,115200 dynd
bg="file drivers/usb/core/hub.c +p"
```

Save and quit file update, and then reboot the board.

Note: to display these debug messages in the console, in addition to the dmesg, add **loglevel=8** in the kernel command line.

- Reboot the board and check for a kernel command-line, and that debug messages are present in the dmesg output

6 References

- [1.01.1] Documentation/admin-guide/dynamic-debug-howto.rst
- http://ex-vi.sourceforge.net/vi.html
- http://ex-vi.sourceforge.net/viin/paper.html

- Useful external links

Document link	Document Type	Description
The dynamic debugging interface (lwn.net)	User guide	http://lwn.net
Documentation/dynamic-debug-howto.txt (lwn.txt)	User guide	http://lwn.net
Dynamic debug howto (kernel.org)	Standard	http://www.kernel.org

Debug File System (See https://en.wikipedia.org/wiki/Debugfs for more details)

Process File System (See https://en.wikipedia.org/wiki/Procfs for more details)

User-space Mode Setting

former spelling for e•MMC ('e' in italic)

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)