

STM32MP1 hands on

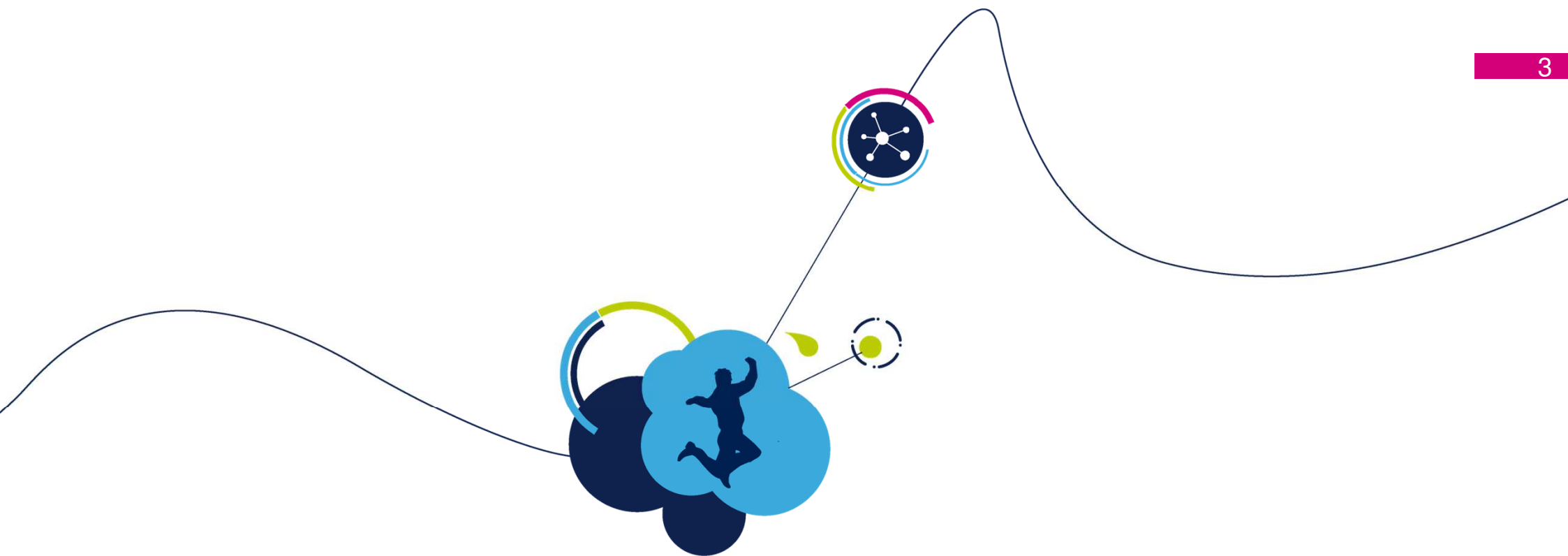
Trace and Debug

Version: 2.0



ST Restricted

- Purpose of this presentation is to give to STMicroelectronics platform developers insight/knowledge/tips about Trace and debug for STM32MP1 platform.
- "Hands on" means **learning by doing**.
- This presentation describes for this topic the following sections:
 - **Learning program**: What are the objectives and benefits of this training. What is the overall plan and expected duration.
 - **Prerequisites**: What is mandatory to study, to have (in term of material) and knowledge (other Training, other Hands On, etc...) before starting this training.
 - **Theoretical school**: Documentation reference (to be consulted on request), self learning or presentation to follow.
 - **Practicing school**: Learning by doing part (sequence of exercises and practical work).
 - **Evaluation**: How to make sure all the objectives of the training where learnt?

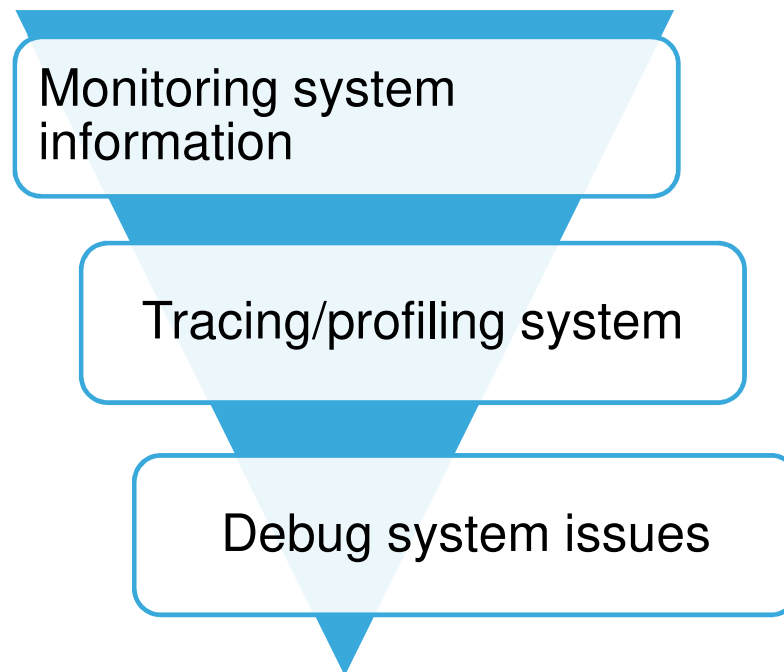


Learning program

Objectives

4

- This learning program provides some guidelines for a “white box” approach of STM32MP1 platform, to give trainee the set of tools and guide lines for getting system information, tracing/profiling system behavior and debugging system issues.



The benefits for the trainee are:

- To have a toolkit for tracing, profiling and debugging through some labs. Objective is not to be expert, but to know how to put them in place, and make basic usage.
- To be able to select the appropriate tool for system investigations, depending on initial observations.

- Progression is following an incremental thread to help trainee from getting simple system information up to make step by step deeper source code debugging or post mortem analysis using core dump.
- Cortex-A7 Linux kernel and Cortex-M4 firmware (copro FW) are covered.
- Trainee will also use some tools for making system profiling. This intends to address system performance studies.

Basic system monitoring and tracing

- Linux kernel & Copro FW
- Lab.1 to Lab.4

Theory + practice: 1d

Advanced Linux kernel logs and traces for system tracing, debugging and profiling

- Linux kernel
- Lab.5

Theory + practice: 2d

Software execution debugging and memory leak monitoring

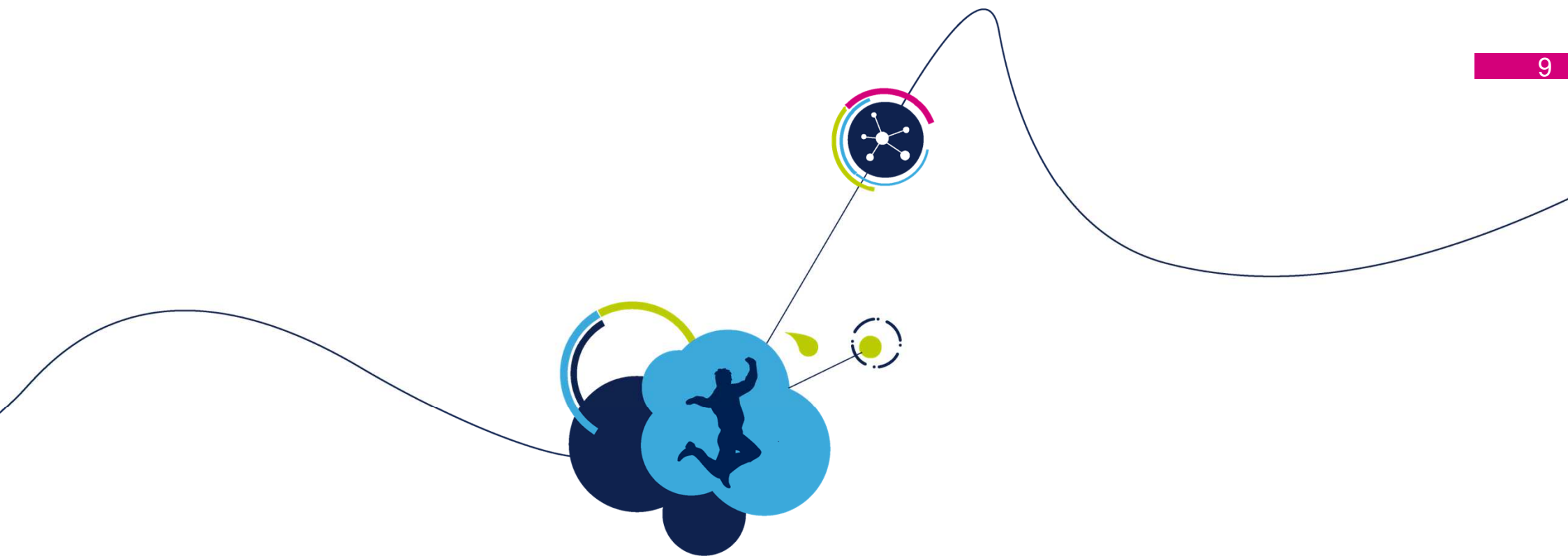
- Linux kernel&App and copro FW
- Lab.6 & Lab.7

Theory + practice: 2.5d

Not yet covered 8

Following parts are not covered in current version:

- Cross-debug, debug both Cortex-A7 and Cortex-M4
- Core dump for user-land application on Cortex-A7
- HDP: i.e. Trace IRQ out 0 from Cortex-A7 (check if applicable)
- STM: i.e. Trace UART for console (check if applicable)



Prerequisites

Prerequisites 1/2

10

- You must have executed the STM32MP1 OpenSTLinux Hands-on for platform environment setup and played with different kits.
- You must have a Developer kit available and the corresponding binaries from the Starter kit loaded in your board. You must also have hands-on kernel driver and hands-on application sources and System Workbench for STM32 IDE installed (Ubuntu).
- Materials: USB Key

Prerequisites 2/2

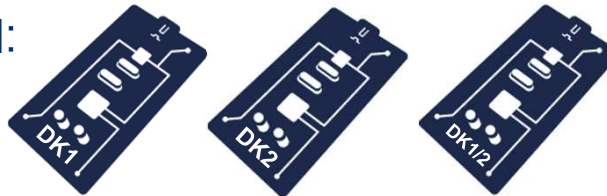
11

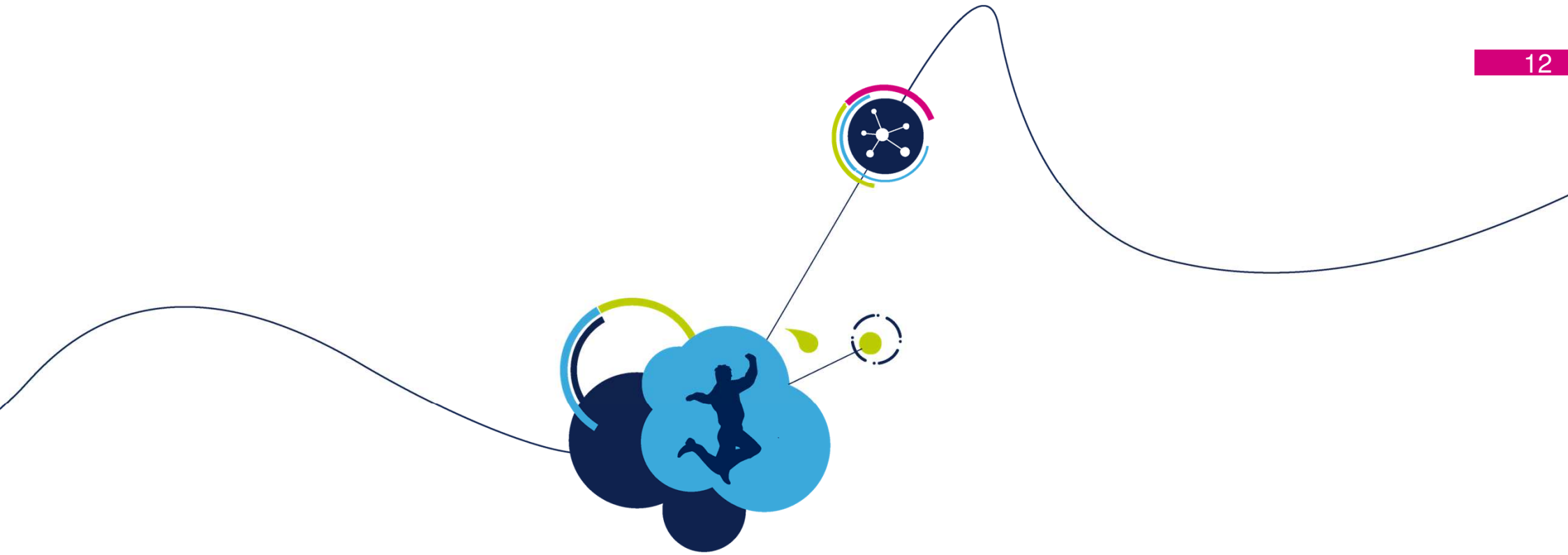
- In this presentation all the references will be quoted as below:
 - { element }: reference to a deliverable present in this area.
 - [article]: reference to an article of the user guide (wiki).
 - “element” or “article” are not direct link but keyword you should easily find with a search.
- For each lab, applicable hardware board(s) are identified as below:

- STM32MP15 Eval board:



- STM32MP15 Disco board:

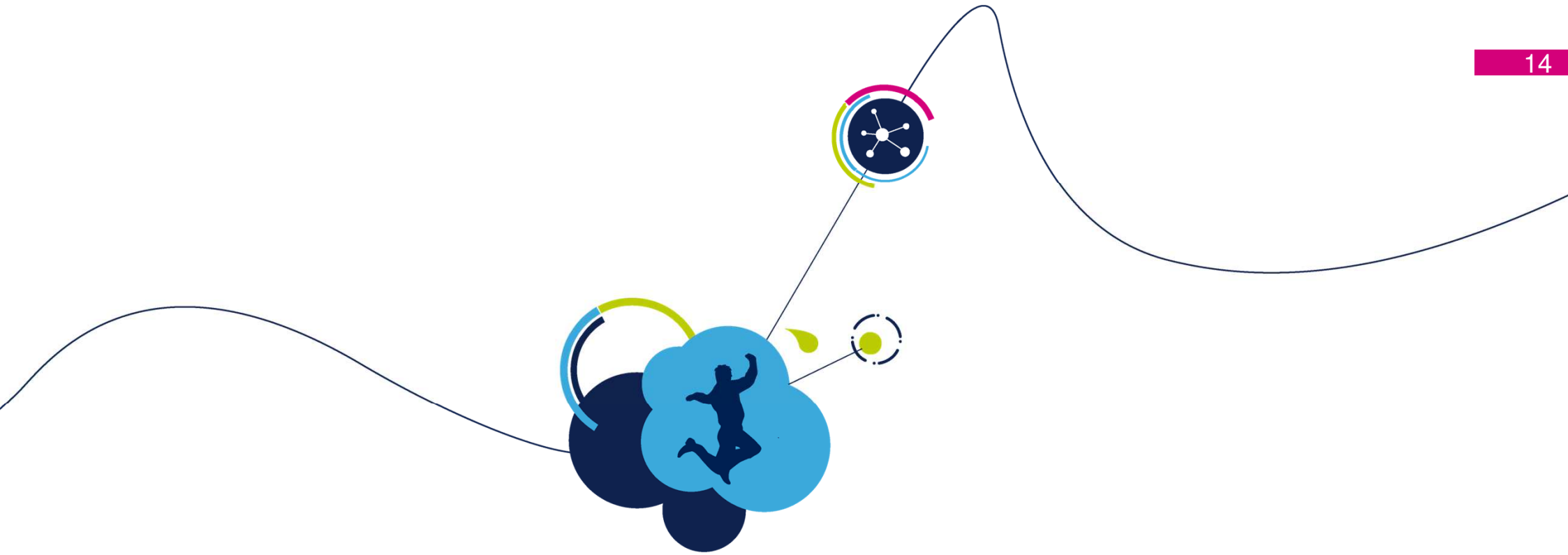




Theoretical school

Duration: 2h

- OpenSTLinux training
 1. { STM32MP1 - Trace and debug training }
 2. { STM32-MPU-IDE_presentation }
- OpenSTLinux User guide articles
 1. [Trace and debug tools]
 2. [Category:Trace and debug tools]



Practicing school

Lab.1

System information in Linux kernel VFS (Virtual File System)

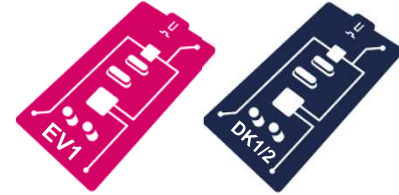
Theoretical school: Linux kernel VFS

16

- **Duration:** 1h
- OpenSTLinux User guide articles
 1. [Virtual File System (VFS)]
 2. [Debugfs]
 3. [How to find Linux kernel driver associated to a device]
 4. [STM32MP15 clock tree]

Practicing school: System info in VFS 1/8

17



- **Duration:** 2h
- **Objective:** Be able to read system information from the sysfs and debugfs
- **Environment:** Starter kit
- **Step1:** Get information from debugfs

TIPS



- /sys/kernel/debug Linux kernel file system entry exists for clk_summary
- For clock value reference, please refer to [STM32MP15 Clock tree] article

Practicing school: System info in VFS 2/8

18



- Get system clocks status from debugfs file `clk_summary`



Use case	Expected value	Observed value
Current value for clk-hsi clock	64000000	

• **Step2:** Read FS entry in VFS



- Check in VFS for assigned interrupts for `i2c2`




TIPS

- `/proc` Linux kernel file system entry exists for interrupts
- For each CPU, a counter of IRQ occurrence is given which depends of current context
- In First column, IRQ Id is dynamically assigned by Linux kernel at boot, and can be different from several openstlinux software.

Practicing school: System info in VFS 3/8

19



Expected results							Read result						
IRQ Id	CPU0	CPU1	Type	HW irq	Trig.	Peripheral device	IRQ Nb	CPU0	CPU1	Type	HW irq	Trig.	Peripheral device
43:	275	0	GIC-0	65	Edge	40013000.i2c							
44:	0	0	GIC-0	66	Edge	40013000.i2c							
45:	0	0	exti	22	Edge	40013000.i2c							

Practicing school: System info in VFS 4/8

20

- **Step3:** Read device tree entry in VFS



- From device tree VFS, get list of i2c2 devices, and their status.



TIPS

- Entry point in VFS for device-tree is /proc/device-tree.
- Path and device address can be read from Linux kernel device tree source files.



Device name	Expected status	Read status
camera@3c	okay	
mfx@42	-	
wm8994@1b	okay	

- ✓ Does it match with status on device-tree source files?

Practicing school: System info in VFS 5/8

21

• **Step4:** Read device tree node parameter from VFS



- From device tree vfs, get i2c2 node parameters values.



TIPS

- Text value can be easily read using command:

```
Board $> cat <parameter>
```

- but for binary values (numeric or structure) you can use command:

```
Board $> od -xi --endian=big <parameter>
```

- It displays a U32 table of variable size corresponding to the number of entries for this parameter. It will be a hexadecimal value, with decimal value given just below. First column is data address.



Device name	Expected status	Read status
compatible	st,stm32f7-i2c	
i2c-scl-rising-time-ns	185	

- ✓ Does it match with status on device-tree source files?

Practicing school: System info in VFS 6/8

22

- **Step5:** Find Linux kernel driver associated to a device entry in VFS



TIPS

- When a device is mounted on the target, a Linux kernel driver is associated with specific assigned Major/Minor numbers which allow to identified it in the /sys/dev/.
- /proc/devices file contains the list of all devices connected to the board with the corresponding major number.
- « ls » command with « -l » option will give the major/minor number



- Target is to find the Linux kernel driver associated to the DRM devices (Direct Rendering Manager)
- Get the DRM device major and minor numbers:

```
Board $> cat /proc/devices | grep drm
```



✓ You should find: 226 drm → this is the major number of this device

Practicing school: System info in VFS 7/8

23



- Find the associated device: check the device entry with 226 as major number

```
Board $> cd /dev  
Board $> ls -lR | grep 226  
crw-rw---- 1 root video 226, 0 Dec 18 16:26 card0
```



- ✓ Only one entry for major 226, which give a minor or 0
- ✓ 'c' letter before the rules information on the left means this is a character device

- As a char device, please find the corresponding system device entry

```
Board $> cd /sys/dev/char  
Board $> ls -l 226\:0  
lrwxrwxrwx 1 root root 0 Dec 18 16:26 226:0 -> ../../devices/platform/soc/5a001000.display-controller/drm/card0
```



- ✓ With this information, you can find the associated driver by looking to the device tree
- ✓ Expected driver name: **st,stm32-ltdc**

Practicing school: System info in VFS 8/8

24

- ? Q&A session (if possible with support contact):**
- **Duration:** 20mn
 - **Objective:** To check if well understand interest of VFS for getting Linux System information at any time
 - **Attendees:** beginners

Lab.2

Linux System monitoring

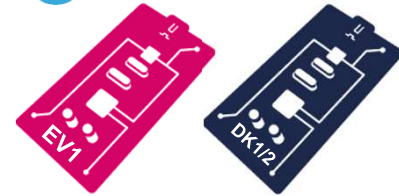
Theoretical school: System monitoring

26

- **Duration:** 30mn
- OpenSTLinux User guide articles
 1. [netdata]

Practicing school: System monitoring 1/4

27



- **Duration:** 30mn
- **Objective:** Use embedded tool which provides useful system monitoring information
- **User setup:** Starter kit
- **Part1 – netdata**



TIPS

- Netdata is reusing the information coming from the Virtual File System seen in Lab.1

Practicing school: System monitoring 2/3

28



- Visualize netdata default web page connected to your target

```
PC $> firefox http://<ip_of_board>:19999
```

- Start openstlinux-hands-driver or openstlinux-hands-appli



- ✓ Observe interrupt corresponding to the usecase (by selecting it on the left legend, you are able to visualize only it)
- ✓ Visualize STM32MP1 Dashboard netdata web page connected to your target:

```
PC $> firefox http://<ip_of_board>:19999/stm32.html
```



- ✓ What is cpu load usage for netdata application: <5%, %5<>10% or >10%

Expected status	Read status
Less than 5% (<5%)	

Practicing school: System monitoring 3/3

29



Q&A session (if possible with support contact):

- **Duration:** 20mn
- **Objective:** Clarify any question related to usage of netdata
- **Attendees:** beginners

Lab.3

Copro firmware log on Linux kernel VFS

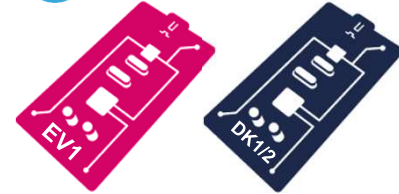
Theoretical school: Copro FW log

31

- **Duration:** 1h
- OpenSTLinux User guide articles
 1. [STM32Cube MPU Package]

Practicing school: Copro FW log 1/4

32



- **Duration:** 1h
- **Objective:** To know how to get logs from Copro firmware on the Linux VFS. First debug level without using IDE Debug mode
- **Environment:**
 - Step 1 → Starter kit
 - Step 2 → Developer kit

Practicing school: Copro FW log 2/4

33

- **Environment: Starter kit**
- **Step1:** Boot OpenAMP_TTY_Echo example and check for log



- Start the OpenAMP_TTY_Echo Copro FW example:

```
Board $> cd /usr/local/examples/OpenAMP_TTY_echo  
Board $> ./fw_cortex_m4.sh start
```



- ✓ Check for the remoteproc0 trace in Linux kernel VFS:

```
Board $> cat /sys/kernel/debug/remoteproc/remoteproc0/trace0
```



- Stop the OpenAMP_TTY_Echo Copro FW example

```
Board $> ./fw_cortex_m4.sh stop
```



- ✓ Does remoteproc0 trace still available? Why?

Practicing school: Copro FW log 3/4

34

- **Environment:** Developer kit

- **Step2:** Add log in OpenAMP_TTY_Echo example

- Launch the OpenAMP_TTY_Echo example in IDE (System Workbench for STM32 including STM32MPU plugin).
- Update the main function to add a debug trace using log_dbg function
- Enable the right level to be able to display this trace



TIPS

- Default LOGLEVEL for Cortex-M4 firmware is given in Utilities/Log/log.h
- Compile the firmware, and run it



- ✓ Check your debug trace in the remoteproc0 trace in Linux kernel VFS

Practicing school: Copro FW log 4/4

35



Q&A session (if possible with support contact):

- **Duration:** 20mn
- **Objective:** Have a good usage of this trace, which is first trace level available without using SW IDE
- **Attendees:** beginners

Lab.4

Kernel log and trace

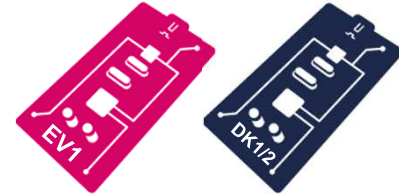
Theoretical school: Kernel log and trace

37

- **Duration:** 1h
- OpenSTLinux User guide articles
 1. [Linux tracing, monitoring and debugging]
 2. [Dmesg and Linux kernel log]
 3. [How to use the kernel dynamic debug (dev_dbg)]

Practicing school: Kernel log and trace 1/6

38



- **Duration:** 1h
- **Objective:** Understand basic Linux kernel log and trace. Way to enable, read, configure, and filter log and trace messages.
- **Environment:** Starter kit
- **Step1:** Console log level



- Boot target



- ✓ Get current console log level:

Use case	Expected	Observed value
Console log level	7	?

Practicing school: Kernel log & trace 2/6

39

- **Step2: Dynamic log**



- Enable dynamic log level for usb core functions (drivers/usb/core/hub.c)
- Unplug then replug USB key



- ✓ Compare messages in console to dmesg content.
- ✓ Does *usb* and *hub* components trace displayed?

Expected		Observed	
In console	In dmesg	In console	In dmesg
No	Yes	?	?

Practicing school: Kernel log & trace 3/6

40

- **Step3:** KERN_DEBUG log level for console



- Enable console log level KERN_DEBUG for console
- Unplug then replug USB key



- ✓ Compare messages in console to dmesg content.
- ✓ Does *usb* and *hub* components trace displayed?

Expected		Observed	
In console	In dmesg	In console	In dmesg
Yes	Yes	?	?

Practicing school: Kernel log & trace 4/6

41

- **Step4:** console KERN_DEBUG log level and dynamic trace at boot



- Enable console KERN_DEBUG log level and dynamic trace for usb core functions in order to be activated at boot. Choose one of two possible methods: by editing extlinux.conf file on host PC side if using SD card, or directly on the target side.
- Plug usb key, then reboot.



- ✓ Compare messages in console to dmesg content.
- ✓ Does *usb* and *hub* components trace displayed?

Expected		Observed	
In console	In dmesg	In console	In dmesg
Yes	Yes	?	?



A little Quiz...

(click for answers if in Slide show mode)

42

- **For getting most of information from the kernel log without making any modification, is better to read console messages or result of dmesg? Why?**
 - > Result of dmesg, as it contain all messages without any log level filtering.

Practicing school: Kernel log and trace 6/6

43

? Q&A session (if possible with support contact):

- **Duration:** 20mn
- **Objective:** Check basic way of working for Kernel log is understood
 - Manage kernel log level
 - Dmesg vs serial console log
 - Enable dynamic trace for Linux kernel components
- **Attendees:** beginners

Lab.5

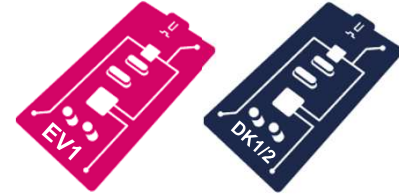
Linux kernel advanced traces

Theoretical school: Advanced traces

- **Duration:** 2h
- OpenSTLinux User guide articles
 1. [strace]
 2. [ftrace]
 3. [LTTng]
 4. [perf]

Practicing school: Advanced traces 1/10

46



- **Total duration:** 11h
- **Objective:** Use Linux kernel advanced trace available for getting debug information on kernel module, on running applications...
- **User setup:**
 - Step 1 (strace, perf) → Starter kit
 - Step 2 to Step 4 (Ftrace, LTTng) → Developer kit

Practicing school: Advanced traces 2/10

47

- **Environment: Starter kit**

- **Part1 – strace (1h)**



- Start openstlinux-hands Linux application (built from the OpenSTLinux hands-on), using strace tool binary

```
Board $> strace openstlinux-hands
```



- ✓ Check openstlinux-hands application is running normally
- ✓ Press User PA14 button
- ✓ Check trace in the console; it should contain all Linux kernel system calls. You can also check links with the source code of the application



TIPS

- This is easy way to trace kernel system call from User space without instrumenting code

Practicing school: Advanced traces 3/10

48

- **Environment:** Developer kit

- **Part 2 – ftrace** (4h)

Step1 - Function mode



- Enable ftrace in kernel configuration
- Make a function trace when executing openstlinux-hands binary



- ✓ Redirect trace to a file, export to Host PC, then edit it to find openstlinux-hands reference. You can then make a link with source code of the application

Practicing school: Advanced traces 4/10

49

Step2 - Function mode using filter



- Reset trace
- Add a function filter to trace all *gpio* function
- Make a new function trace when executing openstlinux-hands binary

✓ Edit trace result, then check for content and link with application source code

Step3 - function_graph mode using filter



- Reset trace
- Add a graph_function filter to trace all *gpio* function
- Make a graph_function trace when executing openstlinux-hands binary

LP3
JR39
JR67

✓ Edit trace result, then check for content and link with application source code

Slide 49

LP3 kernelshark comme viewer serait un plus pour la navigation dans le fichier
Loic PALLARDY; 22/05/2018

JR39 to be checked
Jean-philippe ROMAIN; 22/05/2018

JR67 Propose to add a new step, for next revision
Jean-philippe ROMAIN; 31/05/2018

Practicing school: Advanced traces 5/10

50

- **Environment:** Developer kit

- **Part3 – LTTng** (4h)



- Add LTTng-modules: import, compile and install



TIPS

- Same Kernel configuration done in previous Part2 for ftrace is re-used for LTTng
- Interest for LTTng compare to ftrace is to be able to export trace in log viewer based on eclipse as Trace Compass and which can be re-used in IDE
- LTTng required to integrate some external libraries



- Install Openstlinux-hands-driver kernel module on the target if not already present (*path /lib/module/4.14.13/extra/*)

Practicing school: Advanced traces 6/10

51



- Insert openstlinux-hands-driver kernel module

```
Board $> insmod /lib/module/4.14.13/extra/openstlinux-hands-driver.ko
```

- Generate a LTTng trace with all kernel events when pressing on User PA13 to enable/disable GreenLED



- Install TraceCompass and open LTTng trace previously generated



- ✓ Check in 'Resources' panel, the event linked to IRQ value used by Openstlinux-hands-driver module
- ✓ You should see gpio_value event associated: what is meaning of get= and value=?

Practicing school: Advanced traces 7/10

52

- **Environment:** Starter kit

- **Part4 – perf** (2h)

Step 1: perf top



- Execute *perf top* command to see live event count , and CPU Central processing unit load by functions

```
Board $> perf top
```



TIPS

- Compare to “*top*” command, “*perf top*” gives better user interface with list of running functions linked to running objects (kernel, libraries, application...)
- “*perf top*” has same constraint than “*top*”, as result is given at a given timing (every 3s) an so events which occur between two update are not displayed

Practicing school: Advanced traces 8/10

53

Step 2: perf record / perf report



- Execute *perf record* command as prefix of a command to execute (system command, or application...). Here an example using dd tool to create a file of 100MB size

```
Board $> perf record -g dd if=/dev/urandom of=test_100MB bs=10M count=10 iflag=fullblock
```

- Visualize result using perf report command (by default it will take the latest record done, named perf.data)

```
Board $> perf report
```

Practicing school: Advanced traces 9/10

Step 3: Flame graph



TIPS

- -g option set with perf record command allow to visualize result as a Flame graph



- From the previous record done, you can generate a FlameGraph.
- You can to ensure FlameGraph tool suite is installed on your Host Linux PC

```
Board $> perf script > perf_dd.out
```

Import the file to host PC side

```
PC $> cd <FlameGraph_dir>
```

```
PC $> ./stackcollapse-perf.pl perf_dd.out > out.dd_folded
```

```
PC $> ./flamegraph.pl out.dd_folded > dd.svg
```

```
PC $> firefox dd.svg
```



TIPS

- When visualizing FlameGraph, you can zoom function on graph

Practicing school: Advanced traces 10/10

55

- ?** **Q&A session (if possible with support contact):**
- **Duration:** 20mn
 - **Objective:** Be able to put in place the right trace level/type in order to debug system issues
 - **Attendees:** beginners

Lab.6

Debugging with GDB

Theoretical school: Debugging with GDB

- **Duration:** 2h
- OpenSTLinux User guide articles
 1. [STM32Cube MPU Package] + [Eclipse IDE]
 2. [Gdb]
 3. [GDB commands]
 4. [Gdbgui]

Practicing school: Debugging with GDB

1/13

58

- **Duration:** 12h
- **Objective:** Use GDB debug tool which allow to make step by step software execution of an application or OS firmware (Linux kernel or MCU FW)
- **User setup:** Developer kit
- **IDE:** System Workbench for STM32MPU (Linux Ubuntu machine)



Practicing school: Debugging with GDB

2/13

59

- **Environment: Developer kit**
- **Part1 – Debugging Linux application using gdbserver (3h)**

Step1 – Using GDB command line



- Install openstlinux-hands-appli Linux application on the target if not already present
- On target side (Board), start gdbserver on openstlinux-hands Linux application
- On host side (PC), start gdb (arm-openstlinux_weston-linux-gnueabi-gdb from your developer kit), and connect it to remote target board
- Set a break point on openstlinux-hands-appli source:

In main function (openstlinux-hands.c):

case GPIOEVENT_EVENT_FALLING_EDGE:

data.values[0] = !data.values[0];

→ ret = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data);

Practicing school: Debugging with GDB

3/13

60



- Continue executing the application



- ✓ Check software is well stopped on the breakpoint when pressing User PA13.
- ✓ Read new led value to be programmed.
- ✓ Do step by step software execution up the new is taking new value.
- ✓ Then continue to run the software, and check for new software break when pressing User PA13.

Step2: Using GDBGUI (Browser-based debugger graphic user interface)



- Execute previous steps by using GDBGUI tool



- ✓ Check for same behavior as Step1

Practicing school: Debugging with GDB

4/13

61

- **Part2 – Debugging Linux kernel with ST-link (JTAG/SWD)(4h):**

Step1: Running GDB and debug in command line mode



- Install openstlinux-hands-appli Linux application on the target if not already present
- Configure GDB environment files (setup.gdb and path_env.gdb) in order to attach Linux kernel in running mode
- Start GDB, and then check attach on running mode is ok (no more able to use UART console).
- Launch openstlinux-hands-appli application
- Set a breakpoint in function linehandle_ioctl

```
(gdb) break gpiod_set_array_value_complex
```

```
Breakpoint 1 at 0xc039d16c: file /mnt/data/views/openstlinux-4.14-rocko-mp1-18-06-15/sources/linux-stm32mp/drivers/gpio/gpiolib.c, line 2611.
```

Practicing school: Debugging with GDB

5/13

62



✓ Check the list of breakpoint(s)

(gdb) info break

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0xc039d16c	in gpiod_set_array_value_complex at /mnt/data/views/openstlinux-4.14-rocko-mp1-18-06-15/sources/linux-stm32mp/drivers/gpio/gpiolib.c:2611.

✓ Continue software running

(gdb) continue

✓ Press User14 key, and then check software execution is stopped at expected break



TIPS

- By using GDB TUI (Text User Interface), you can have a multi view of source code, assembly code, and gdb prompt.

(gdb) layout split

- You can come back to basic GDB command prompt interface:

(gdb) tui disable

Practicing school: Debugging with GDB

6/13

63



- ✓ Check for the GPIO label (first entry in the desc_array table)

```
(gdb) p *desc_array[0]  
$1 = {gdev = 0xeea7e400, flags = 67, label = 0xec135640 "User PA14", name = 0x0}
```

- ✓ Continue software running up to line « int value = value_array[i]; » using « next » command
- ✓ Check requested update value for GPIO

```
(gdb) p value_array[0]  
$1 = 1 (if led will be enable), 0 (if led will be disable)
```

- ✓ Then « continue » command to continue running the program

Practicing school: Debugging with GDB

7/13

64

Step2: Running GDB and debug using GDBGUI

- Install openstlinux-hands-appli Linux application on the target if not already present
- Configure GDB environment files (setup.gdb and path_env.gdb) in order to attach Linux kernel in running mode
- Start GBD using GDBGUI, and then check attach on running mode is ok (no more able to use UART console).
- Launch openstlinux-hands-appli application
- Load file drivers/gpoi/gpiolib.c (fetch source files on the top left corner)
- Set a breakpoint in *function linehandle_ioctl* in code line « int value = value_array[i]”



TIPS

- In GDBGUI, to set a breakpoint on a line, just click on the line number in the source file



- ✓ Step in code execution ,and check different value of variables following the case, set led On / Off

Practicing school: Debugging with GDB

8/13

65

• Part3 – Debugging kernel external module with ST-link (JTAG/SWD)(3h)



Step1: Running GDB and debug in command line mode

- Install openstlinux-hands-driver Linux driver on the target if not already present
- Configure GDB environment files (setup.gdb and path_env.gdb) in order to attach Linux kernel in running mode
- Start GDB, and then check attach on running mode is ok (no more able to use UART console)
- Insert openstlinux-hands-driver driver
- Load symbol for openstlinux-hands-driver driver
- Set a breakpoint in function *isr*

(gdb) break isr

Breakpoint 2 at 0xbf0e90e4: file /local/views/hands-on/openstlinux-distribution/openstlinux-hands-driver/openstlinux-hands-driver.c, line 15.

Practicing school: Debugging with GDB

9/13

66



- ✓ Continue software running

```
(gdb) continue
```

- ✓ Press User14 key, and then check software execution is stopped at expected break
- ✓ NB: You can also use TUI mode (*layout split* command to visualize source code)
- ✓ Progress in the source code execution and check for gpio value

```
(gdb) next
```

```
...
```

```
(gdb) p value
```

```
$1 = 1 (or 0 following user press steps)
```

- ✓ Then « continue » command to continue running the program


Practicing school: Debugging with GDB

10/13

67

Step2: Running GDB and debug using GDBGUI

- Install openstlinux-hands-driver Linux driver on the target if not already present
- Configure GDB environment files (setup.gdb and path_env.gdb) in order to attach Linux kernel in running mode
- Start GBD using GDBGUI, and then check attach on running mode is ok (no more able to use UART console).
- Install openstlinux-hands-driver driver
- Load symbol for openstlinux-hands-driver driver (by command line)
- Load file <your_path>/hands-on/openstlinux-distribution/openstlinux-hands-driver/openstlinux-hands-driver.c (fetch source files on the top left corner)
- Set a breakpoint in *isr* function

 ✓ Step in code execution ,and check different value of variables following the case, set led On / Off

Practicing school: Debugging with GDB

11/13

68

- **Part4 – Debugging Cortex-M4 firmware with JTAG from IDE/SWD(2h)**



Step1: With FW loading

- Start the OpenAMP_TTY_Echo Copro FW example in debug mode.
- Break execution, and set a breakpoint on the main function inside the case VirtUart0RxMsg is available:

In main function (main.c):

```
if (VirtUart0RxMsg) {  
    VirtUart0RxMsg = RESET;  
    → VIRT_UART_Transmit(&huart0, VirtUart0ChannelBuffRx, VirtUart0ChannelRxSize);  
}
```

- Continue running the binary, and then play with echo mode. As soon as the software execution is broken at the breakpoint, check for the string received, try to modify it, and continue software execution.

✓ Echo value on console should be the new string you updated.



ST Restricted

Practicing school: Debugging with GDB

12/13

69



Step2: do same steps as previous but without FW loading (attach on running fw)



TIPS

- Option for FW loading is set in *Startup* sheet of *Debug Configurations* panel

Practicing school: Debugging with GDB

13/13

70



Q&A session (if possible with support contact):

- **Duration:** 60mn
- **Objective:** Well understand the path for using GDB following the specific part we want to debug, and how to put it in place
- **Attendees:** beginners

Lab.7

Monitoring memory leak

Theoretical school: Monitoring memory leak

72

- **Duration:** 1h
- OpenSTLinux User guide articles
 1. [Kmemleak]
 2. [Valgrind]

Practicing school: Monitoring memory leak

73

1/8

- **Duration:** 4h
- **Objective:** Be able to detect memory leak during software execution on Linux kernel and Linux application parts
- **User setup:** Developer kit



Practicing school: Monitoring memory leak

2/8

74

- **Environment:** Developer kit
- **Part1 – Monitoring memory leak for Linux kernel by using kmemleak tool (2h)**



Step1 –

- Enable kmemleak (Kernel memory leak detector tool) in Linux kernel config
- Install openstlinux-hands-driver kernel module on your PC to be ready for compilation if not already present. Do not forget also change to be done in Linux kernel device tree
- You will have to modify the openstlinux-hands-driver.c source file in order to add new function for this session. It will create a memory leak.

In openstlinux-hands-driver.c file beginning, add declaration of new function:

```
+ void display_led_state(int value);
```

which have then to be called in *isr* function before « return »:

```
+ display_led_state(!value);
```

```
return IRQ_HANDLER;
```

```
}
```

Practicing school: Monitoring memory leak

3/8

75



Add definition of the new function:

```
void display_led_state(int value)
{
    int num = 4; /* Max size + 1 for end character*/
    char* str_state;

    str_state = (char*) kmalloc (num * sizeof(char),GFP_KERNEL);

    if (value == 0) {
        strncpy(str_state, "OFF", num);
    }
    else {
        strncpy(str_state, "ON", num);
    }

    pr_info("openstlinux-hands-driver: led is now %s\n",str_state);
}
```

Practicing school: Monitoring memory leak

4/8

76



- Compile new Linux kernel module openstlinux-hands-driver.ko, and then install it on the target

```
PC $> scp openstlinux-hands-driver.ko root@<Your_target_board_IP>:/lib/modules/4.14.<xx>/extra/  
PC $> ssh root@<Your_target_board_IP> sync
```

- Insert the openstlinux-hands-driver and press User PA 13 button

```
Board $> insmod /lib/modules/4.14.<xx>/extra/ openstlinux-hands-driver.ko
```



- ✓ By using kmemleak tool, please check for memory leak. Some should be detected on your openstlinux-hands-driver
- ✓ Please check to solve the issue on the source code of the new function
- ✓ When corrected, please test your new module and check there is no more issue

Practicing school: Monitoring memory leak

5/8

77

- **Part2 – Monitoring memory leak for Linux application by using Valgrind (2h)**



Step1 –

- Install openstlinux-hands application on your PC to be ready for compilation if not already present.
- You will have to modify the openstlinux-hands.c source file in order to add new function for this session. It will create a memory leak.

Add declaration of new global variable *chrdev_name*:

```
...  
#include <linux/gpio.h>  
  
+char* chrdev_name;  
...
```

Practicing school: Monitoring memory leak

6/8

78



Modify allocation of *chrdev_name* in main function:

```
int main(int argc, char **argv)
{
    struct gpiohandle_request req;
    struct gpioevent_request ereq;
    struct gpiohandle_data data;
    struct gpioevent_data event;
    - char chrdev_name[20];
    int fd, ret;

    /* Register signals for application exit conditions */
    signal(SIGINT, exit_fct); /* Ctrl-C signal */
    signal(SIGTERM, exit_fct); /* kill command */

    + chrdev_name = (char*) malloc (20 * sizeof(char));

    ...
}
```

Practicing school: Monitoring memory leak

7/8

79



- Compile new openstlinux-hands application, and then install it on the target

```
Board $> mkdir /usr/local/bin → if not already exist
```

```
PC $> scp openstlinux-hands root@<Your_target_board_IP>:/usr/local/bin/
```

```
PC $> ssh root@<Your_target_board_IP> sync
```

- Execute the openstlinux-hands application with valgrind

```
Board $> valgrind /usr/local/bin/openstlinux-hands-driver.ko
```



- ✓ End application execution, and then check for valgrind result. A memory leak should be detected on your openstlinux-hands application
- ✓ Please check to solve the issue on the source code of the application
- ✓ When corrected, please test your new module and check there is no more issue

Practicing school: Monitoring memory leak

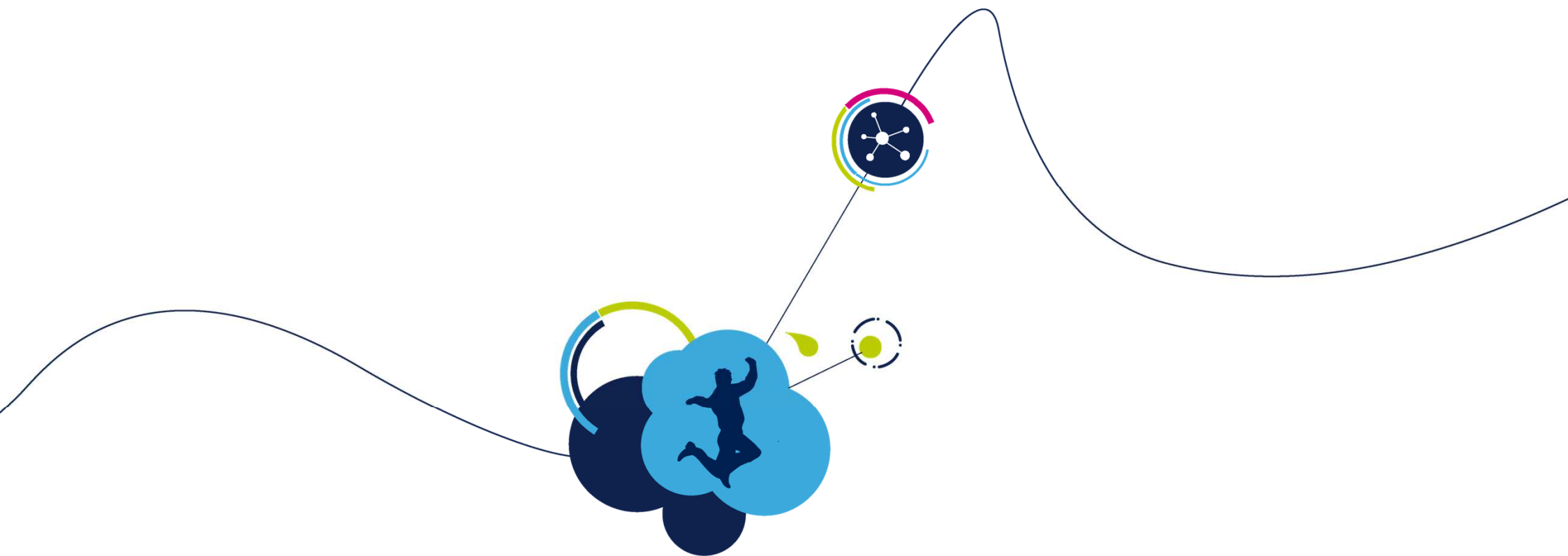
8/8

80



Q&A session (if possible with support contact):

- **Duration:** 60mn
- **Objective:** Well understand available tools to monitor memory leak issue, which can avoid problem during execution
- **Attendees:** beginners



Congratulations !