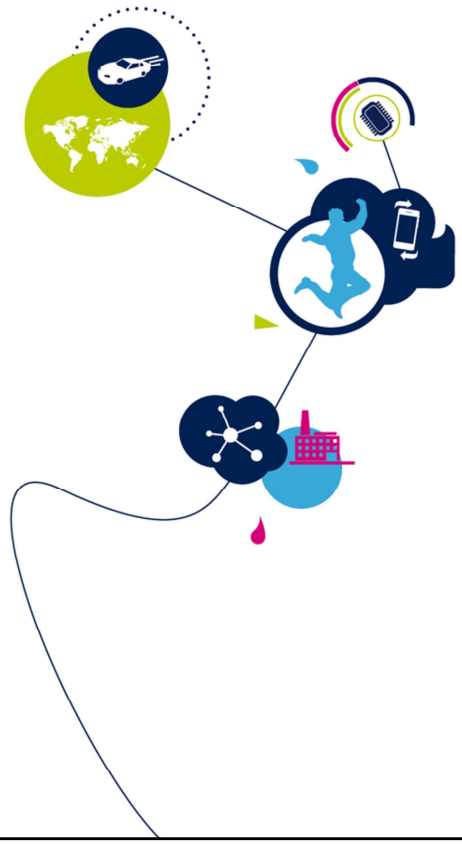
STM32MP1 Platform
Trace and debug solutions

Getting started with Linux kernel tracing, monitoring and debugging

Revision 1.0



life.augmented

Welcome to this training describing trace and debug solutions on the STM32MP1 platform.
This slide set covers only the Cortex-A7 part, running Linux kernel.
The Cortex-M4 part, running Cube software, is presented in another module: STM32MP1-Software-Coprocessor management.

Introduction

2

- STM32MP1 platforms offer various trace and debug paths and solutions
- Trace and debug on STM32MP1 platforms is either :
  - Hardware oriented: debugging Arm cores (Cortex-A7/Cortex-M4) using JTAG port, ETM, STM, ITM, watching internal signal using HDP … (*Not all available on Disco board, only JTag*)
    - Need hardware probe (ST-LINK, U-LINK, …) and/or dedicated host tools
  - Software oriented: tracing and monitoring OS using standard Linux tools : strace, perf, ftrace, kmemleak, etc…
- Any trace and debug session is either:
  - Non-invasive debug :
    - Reading/collecting logs and traces do not modify at all the system behavior
  - Invasive debug :
    - Reading/collecting logs and traces modify the system behavior
    - Depending on tools used, the behavior alteration could range from insignificant to not negligible

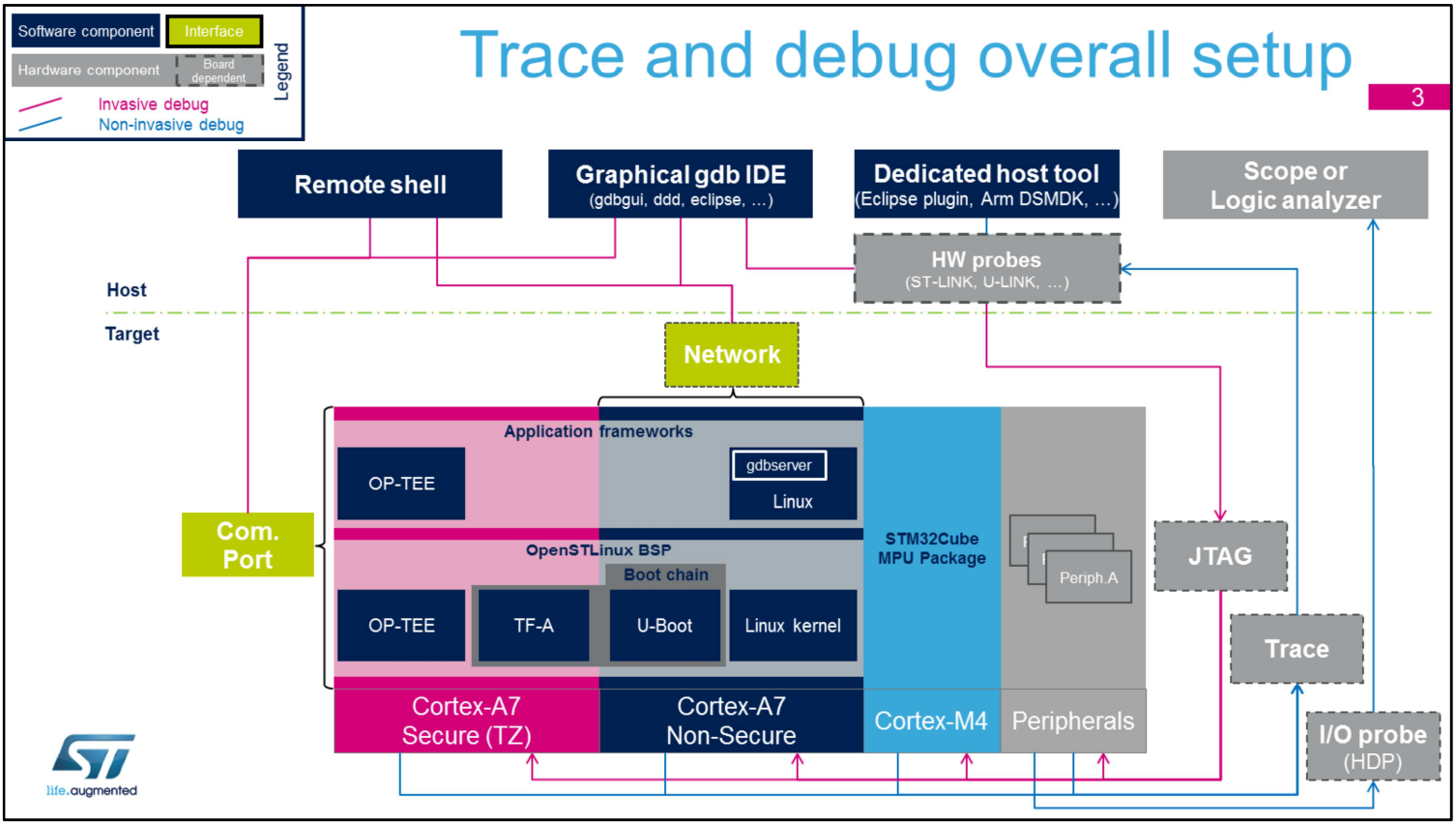
life.augmented

The STM32MP1 platform offers various trace and debug paths and solutions, depending on the part to focus on. Both hardware and software interfaces are proposed on STM32MP1 platforms.

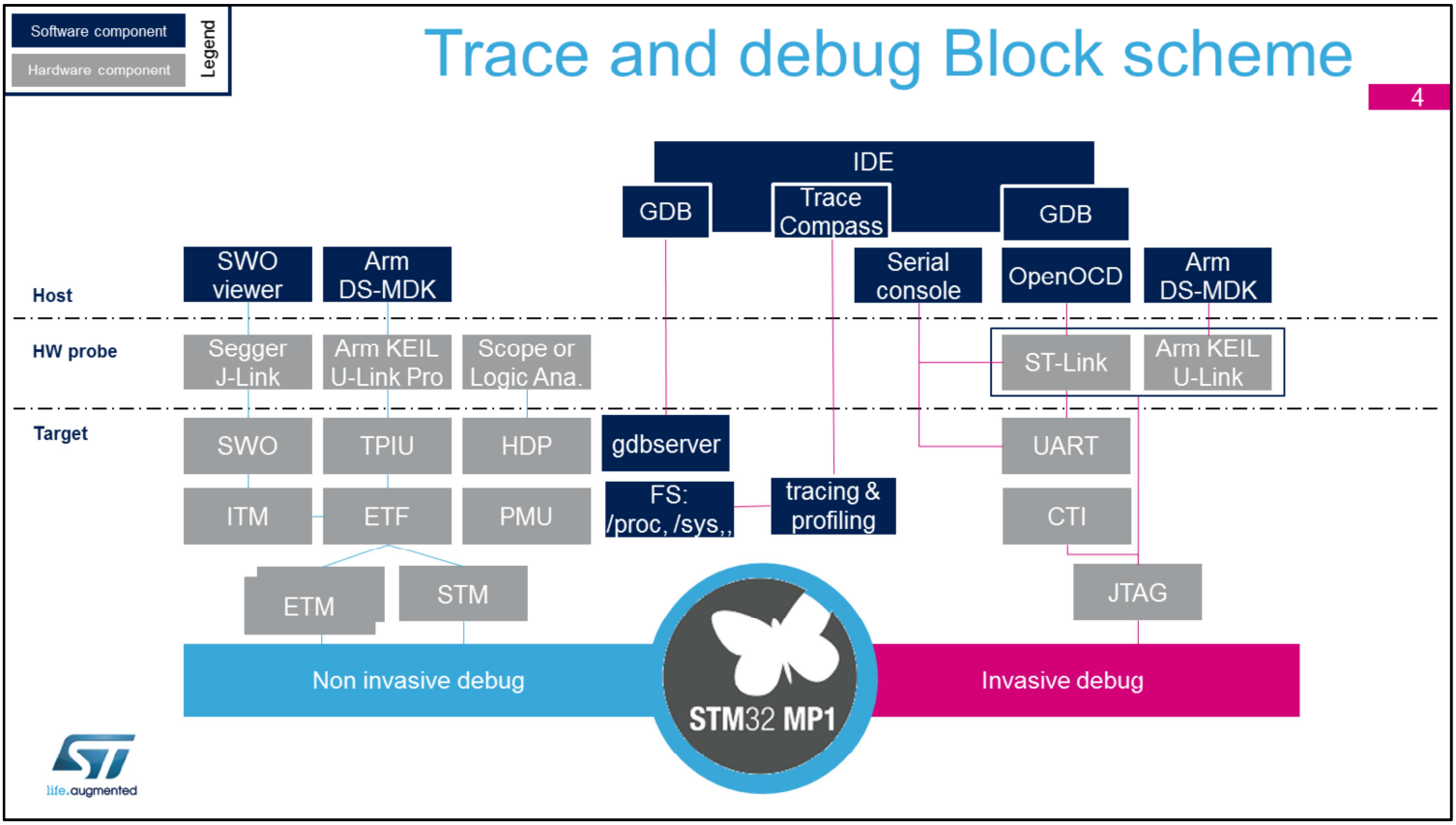
- The hardware path enables hardware signals for debugging Arm cores (Cortex-A7 and/or Cortex-M4), or for watching internal signal between peripherals by using the Hardware Debug Port (HDP). Be careful, these interfaces are not present by default on all boards (For example: only the JTAG port is available on Disco board)
- The software path allows the benefits of the Linux trace and debug environment and tools suite. Some of the tools are embedded in the Linux kernel (for example: strace, perf, ftrace, kmemleak…), not necessarily enabled by default. Some others, interfacing at user space level, can be integrated from outside.

For a debug path, Arm also introduces notions of invasive

and non-invasive debug modes.
Non-invasive debug environment reads or collects information without modifying at all the behavior of the system. It means that it does not impact the system setup and execution. On the contrary, in the invasive debug environment, the system behavior is impacted. This impact can be important, and this has to be known and taken into account by the user when setting up the system.



This scheme is a representation of the trace and debug environment on STM32MP1 microprocessors, showing the different hardware and software paths which interface with cores and hardware peripherals on the target side.
Host side represents different group of tools to interface with the user, and able to manage and /or display the trace and debug data.
We can mention here that this is a general trace and debug setup view. Some external target interfaces may not necessarily be present on all boards. As an example Trace and I/O probe interface is not present on the Disco board.



This scheme is a hardware and software block representation, which allows a parallel with the detailed debug block diagram available in the STM32MP1 reference manual. It highlights the main Arm Coresight block IP which is embedded in the STM32MP1 chip.

Embedded kernel observations (1/3)

- Looking at kernel observations is the first step before using any others debug methods
- Linux kernel logs its activity in a buffer
  - 'printk' is equivalent to printf, and is used to store a message in Linux kernel log buffer
  - 'dmesg' is the command to use to get information about kernel activity, by printing on the console the content of the log buffer (different level of information)
- Linux is a file oriented Operating System
  - Many information about current configs, running processes and apps, and more are stored as file
    - Those files are located in /dev, /proc, /sys and /var directories
  - /sys/kernel/debug (named *debugfs*) is probably one of the most important of them
- Peripheral registers are easily printable
  - Exposed registers are a "software" representation of hardware ones

life.augmented

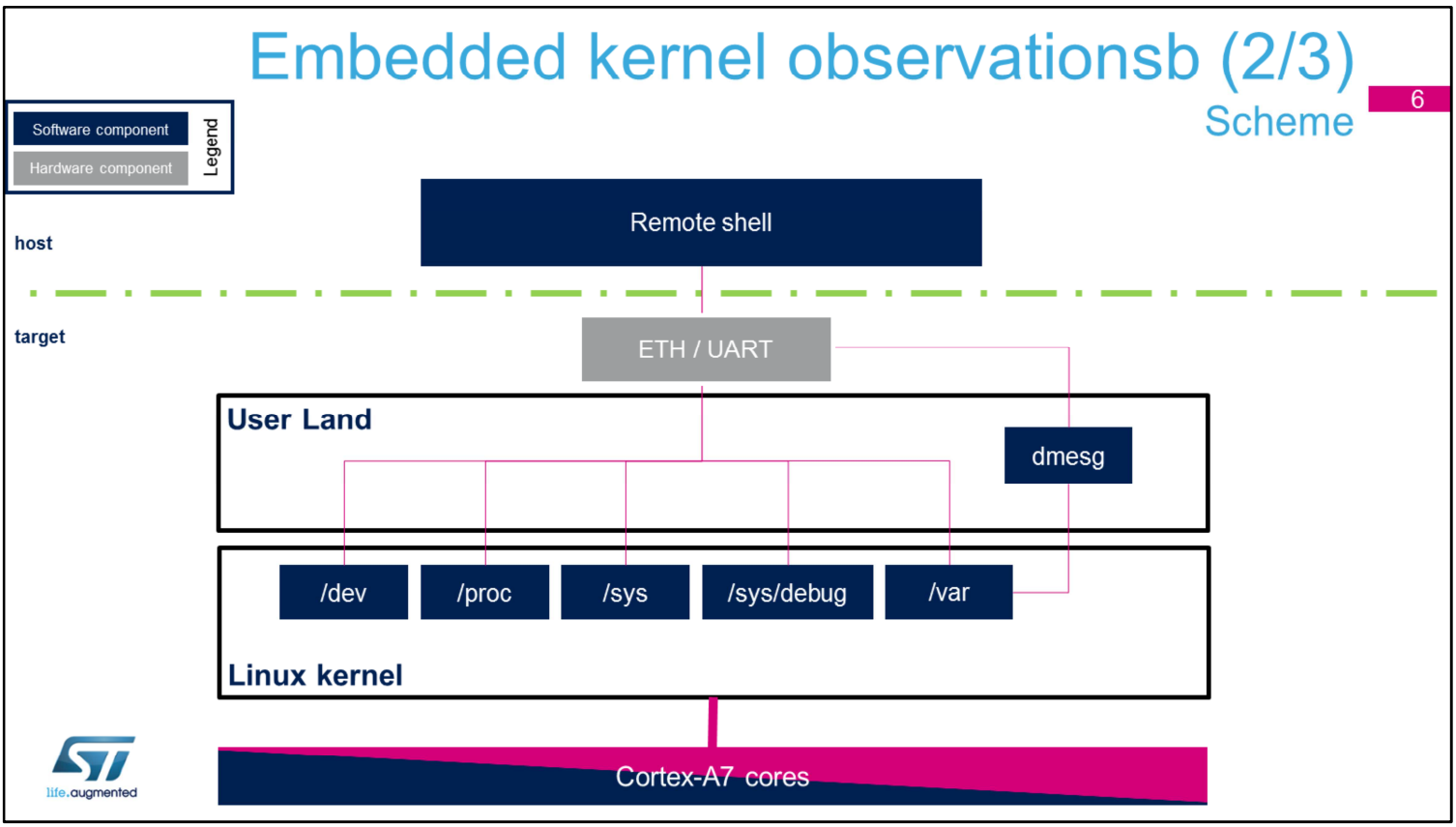
Now let's look at basic embedded kernel observations, illustrated by an example around UART configuration.
Kernel observation is the first step before using any other debug method.
Linux kernel is using a circular log buffer to store information coming from core components or drivers.

Printk is a basic function used to store some messages in this buffer. The simple way to read the content of this buffer is to use the "dmesg" command.
Note that all log messages are classified in different log levels – from 0 to 7 – This classification enables the filtering of those messages when reading them.

Please remember that the Linux kernel is a file system oriented operating system. This means that all information about current configurations, running processes and applications are stored as files, which can be accessible and

read by trace and debug tools. In this file system, you can get a status about STM32MP1 hardware registers settings, which is key to manage the system configuration and monitor the system behavior.

Debugfs is also an important mounted point on the file system which provides more debug information. We will see some usage examples during this training.



This scheme is an extract from the Trace and debug architecture scheme. It focuses on the Linux kernel observation path and summarizes the information given in the previous slide.

Embedded kernel observations (3/3)

Practice - Debug scenario – UART issue

- To put into practice the embedded kernel observations detailed in previous slides of this presentation, you can refer to the wiki article
  - [Trace_and_debug_scenario_-_UART_issue#Embedded_kernel_observations]
- Pre-requisites for debugging any issue on a hardware device
  - Hardware board schematics for the pinout and { STM32MP1 Datasheet } for the alternate functions for pins configuration
  - { STM32MP1 Reference manual } for the description of the related registers, …

To put into practice embedded kernel observations detailed in the previous slides of this presentation, you can refer to the wiki article [Trace_and_debug_scenario_-_UART_issue#Embedded_kernel_observations]

As pre-requisites, It is recommended to get the reference manual, and the data sheet of the STM32MP1 chip, and also the hardware schematics of your board.
The above listed documents enable you to :

- Have full description and configuration information about every related registers,
- know the exact hardware configuration settings.

Trace and debug tools (1/3)

8

- Offer of trace and debug tools in Linux community is huged
  - Most used and useful sum-up on wiki page: [Linux_tracing,_monitoring_and_debugging]
- OpenSTLinux official image embeds most of useful tools
  - strace, ip, ethtools, perf, netdata …
- Two main trace and debug features embedded in Linux kernel
  - Dynamic debug (dyndbg) to dynamically select a piece of kernel to trace
    - Displays trace messages present in the source code, dynamically on demand
    - Information of dyndbg is stored in /sys/kernel/debug/dynamic_debug
    - See Wiki page: [How_to_use_the_kernel_dynamic_debug]
  - Linux kernel tracing feature for kernel deep tracing
    - Core trace framework which enables the tracking of events and function calls, including execution timing
    - Tools like ftrace collect information from Kernel Tracing feature
    - Tracing feature is not set by default in OpenSTLinux Weston image (**need to recompile kernel**)

life.augmented

Here is some information about trace and debug tools by giving an overview about solutions available on the STM32MP1 microprocessor, and by continuing the example on UART issue to illustrate.
Linux kernel embeds a set of tools for deeper kernel observations.

The Linux community also propose some tools mainly at user space level, which can get multiple information from the file system, and especially from the debugfs.

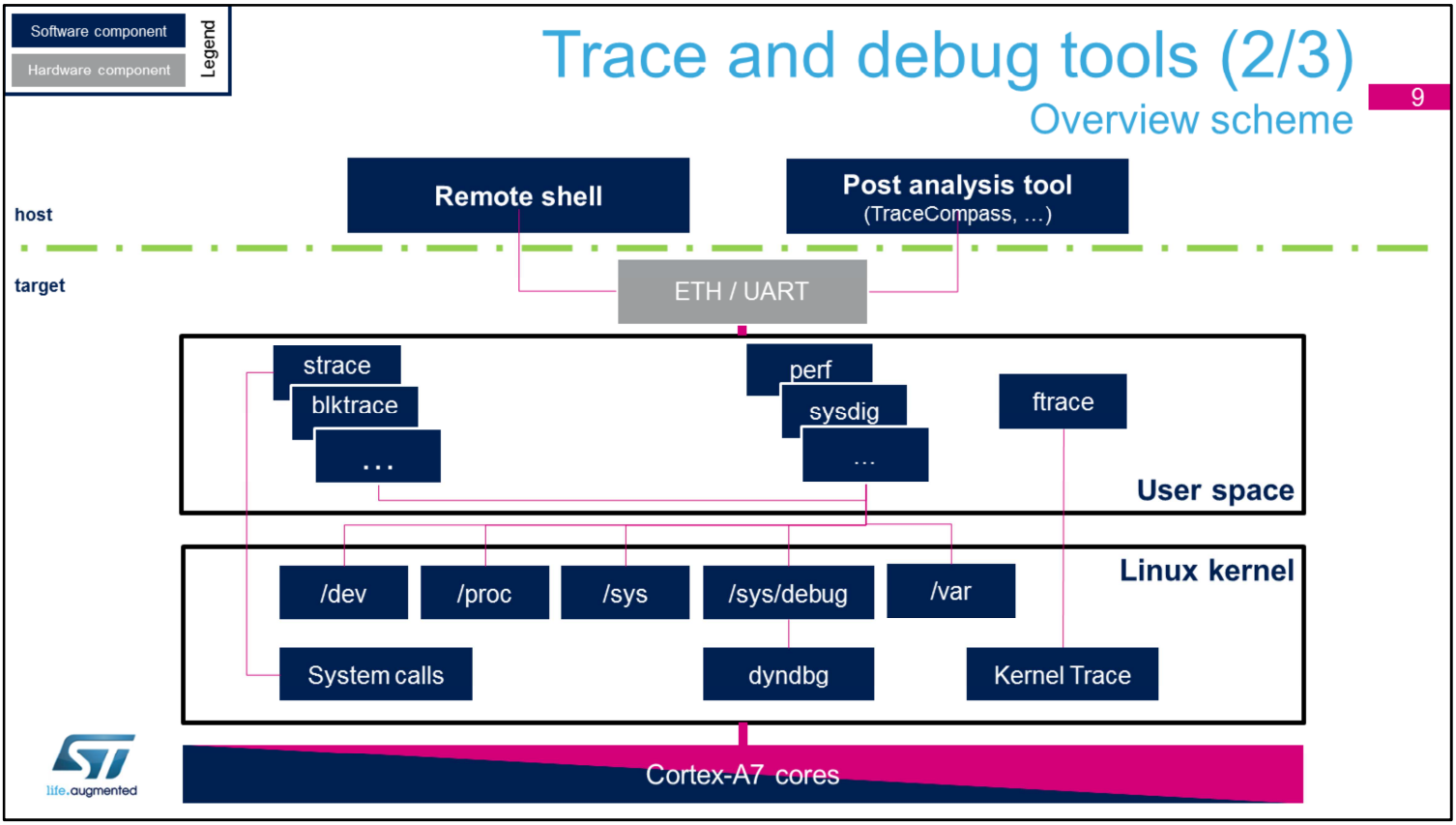
The Linux tracing, monitoring and debugging wiki page is a good entry point about trace and debug solutions for Linux kernel ported on STM32MP1.

Three main axis are proposed to get started with tracing and monitoring setup:

- Some tools present by default on OpenSTLinux

distribution which provide useful information. Details for those tools are described in corresponding wiki article

- We highlight here two main trace and debug features embedded with the Linux kernel, so present on OpenSTLinux:
    - Dynamic debug which is an important tracing feature. It triggers trace on demand, to not overload the trace buffer. Those traces are present on the source code with specific syntax (dev_dbg or pr_debug). Corresponding wiki article is available.
    - Linux kernel tracing feature is also a powerful trace environment, which provides information about Linux kernel code execution. It can be used to trace functions calls to make performance analysis. Please note that this tracing feature is not enable by default, as it is intrusive in code execution. All details are given in the ftrace tool wiki article.



This scheme is a block view representation of the Linux kernel tracing and monitoring links, including some of tools available at user space level.

Trace and debug tools (3/3)

Practice - Debug scenario – UART issue

- To put into practice trace and debug tools information detailed in previous slide of this presentation chapter, you can refer to the wiki article
  - [Trace_and_debug_scenario_-_UART_issue#Trace_and_debug_tools]
- Pre-requisites for using those tools
  - Know how to recompile a Linux kernel with new configuration
  - Have kernel source code especially the related driver source code

life.augmented

To put into practice trace and debug tools information detailed in previous slides of this presentation, you can refer to the wiki article [Trace_and_debug_scenario_-_UART_issue#Trace_and_debug_tools]

Following pre-requisites are recommended:

- STM32MPU Embedded software packages available (Developer package or Distribution package) to be able to modify the kernel configuration options, and recompile the kernel image
- Linux kernel source code in order to be able to check for the debug traces available for a given driver

Advanced trace and debug tools (1/3)

Overview details

- Take advantage of hardware traces
  - Hardware traces are low level traces using Hardware Debug Port (HDP) or JTAG/Trace connectors. Note that it depends of board configuration
  - Hardware probes are needed : ST-Link, U-Link, …, scope or logic analyzer
- Do active debug directly in source code
  - Control source code running with breakpoints, step by step execution…
  - gdb is a Linux tools to do active debug

life.augmented

Let's see some advance trace and debug tools which take advantage of hardware traces, and active debugging directly on code execution on cores.
Depending on the STM32MP1 board you are using, you can access fully or partially to some hardware debug ports for getting traces, or signals.

Three main hardware debug ports are proposed and could be available on board:

- JTAG and Serial wire port (SWD) through hardware probes such as ST-Link or through JTAG connector; this helps drive the software execution and access memory (read and/or write)
- Hardware Trace connector which can provide Arm Coresight signals also linked to software execution on cores
- Hardware Debug Port (HDP) which allows the observation of internal signals. This is an internal

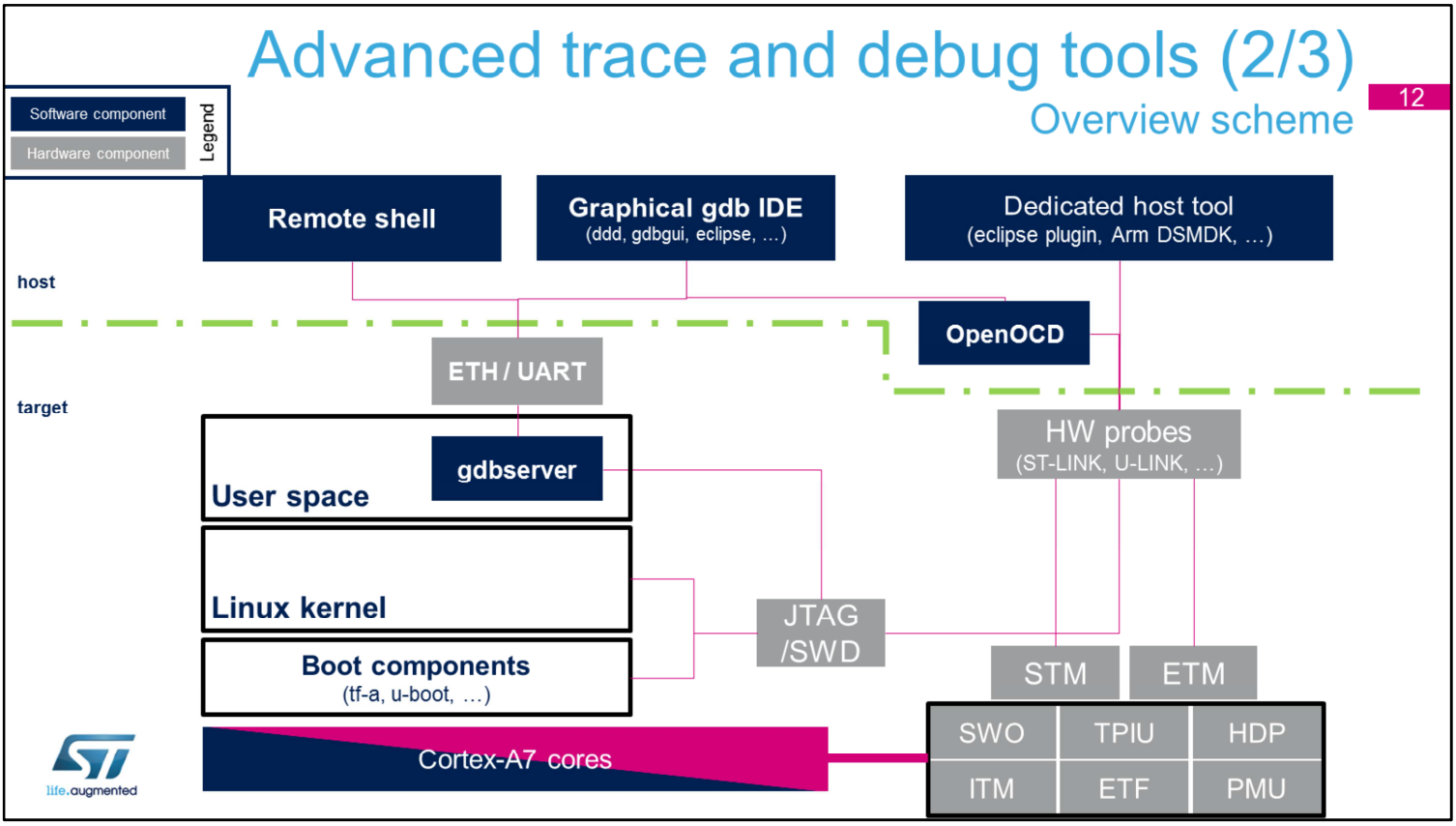
peripheral, configured by a Linux kernel driver, and for which output signals are mapped on GPIO

An important usage of this debug port is the possibility to do active debug directly on the source code running.
It is possible to manage software execution by adding breakpoints, perform step by step execution, tracing variables, etc…

For this environment, use the ST-Link probe to connect the target to the host PC, and as debug tool, use the GDB debugger interface, linked to OpenOCD interface, which enables the control of the JTAG/SWD port.
For more detail, you can refer to the [GDB] wiki page.

Advanced trace and debug tools (2/3)

Overview scheme



You can find here a representation of the hardware debug configurations detailed in the previous slide.

Advanced trace and debug tools (3/3)

Practice - Debug scenario – UART issue

- To put into practice advance trace and debug tools information detailed in previous slides of this presentation, you can refer to the wiki article
  - [Trace_and_debug_scenario_-_UART_issue#Control_execution_of_code_with_gdb]
- Pre-requisites for using GDB
  - Developer package for GDB/OpenOCD environment

To put into practice advance trace and debug tools information detailed in previous slides of this presentation, you can refer to the wiki article [Trace_and_debug_scenario_-_UART_issue#Control_execution_of_code_with_gdb]

Following pre-requisites are recommended:

- STM32MPU Embedded software Developer package available for GDB and OpenOCD environment

Useful wiki related pages

- **https://wiki.st.com/stm32mpu**
- Trace and debug environment overview
  - [STM32MP1_Platform_trace_and_debug_environment_overview]
- Trace and debug tools for Linux
  - [Linux_tracing,_monitoring_and_debugging]
- Linux kernel file system
  - [File Hierarchy Standard (FHS)]
- Advanced debug tools
  - [GDB]

life.augmented

STM32MP1 User guide provides useful wiki articles to support developers on several aspects:

- How to use of the platform
- How to trace, monitor and debug the platform.

To go further on trace and debug solutions for STM32MP1, simply go to wiki.st.com/stm32mpu and search for the articles listed in this presentation.

Thank you