arm

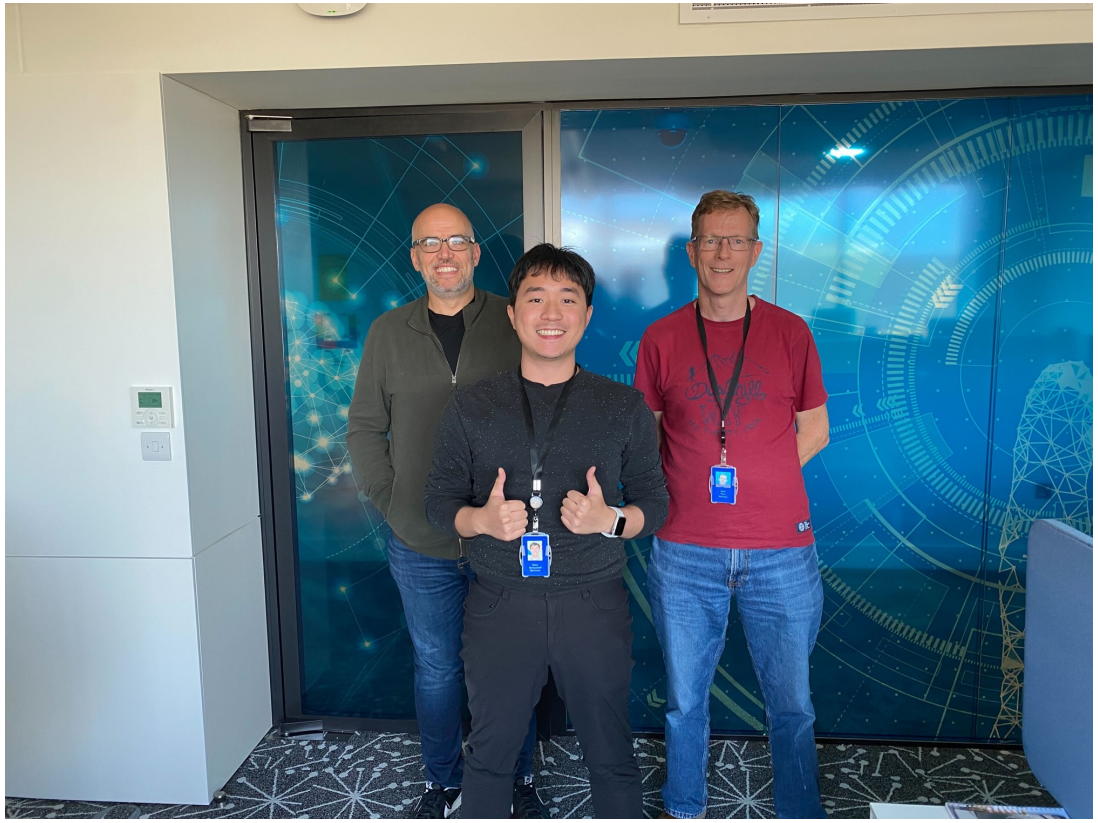
On-Chip **Tracing** using ARM **CoreSight Access Library** on AARCH64 Armv8-A

rakha.djokosoetono@arm.com

My Internship Experience

Placed at the **Debugging Team,**

July 4 – September 16 2022



Project Objectives (layman's)

- A **Debug-and-Trace tool's usage** is not well-understood within the Arm Debugging Team.

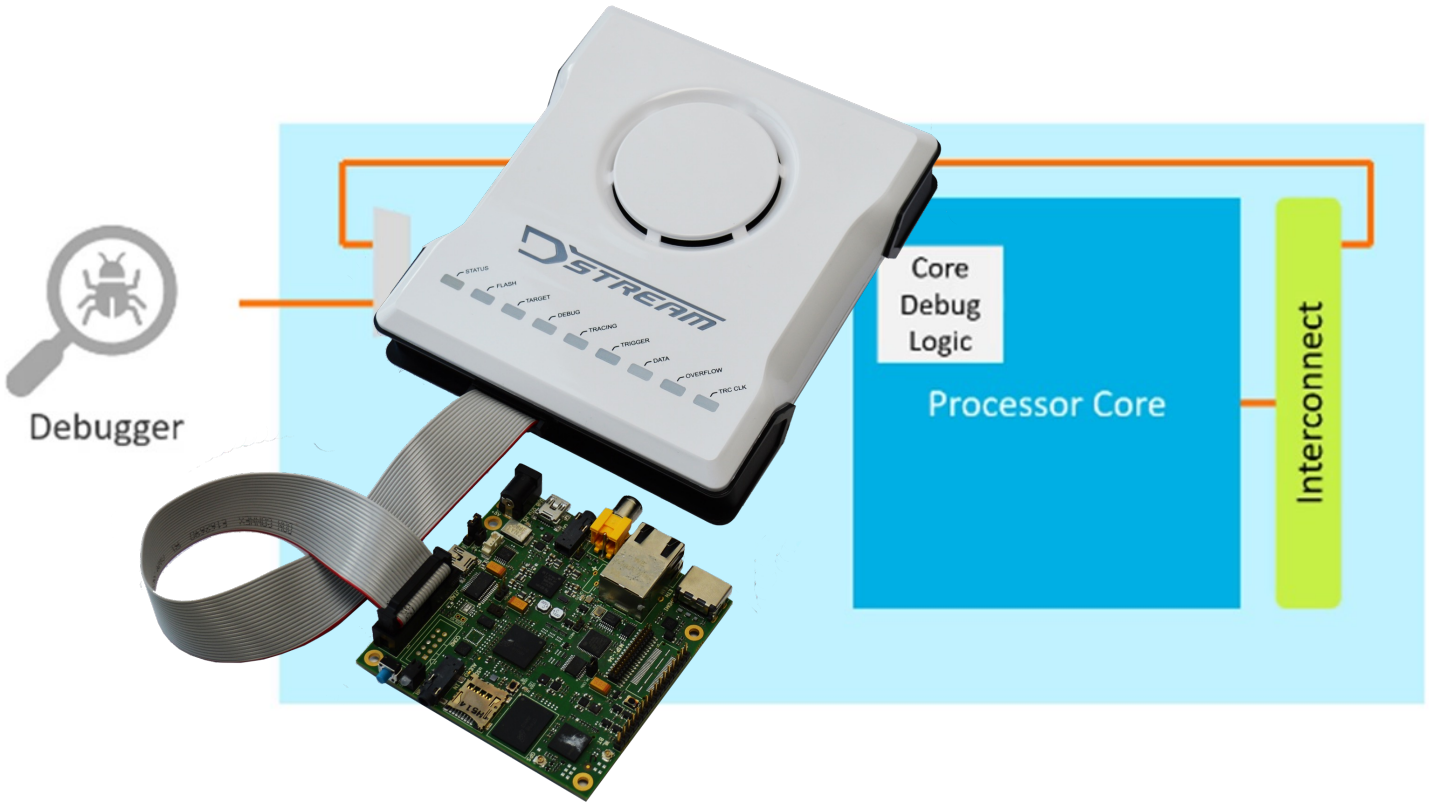
- Provide support for the tool to a **development board, document the process, communicate findings.**

Step 0:
All Relevant Concepts

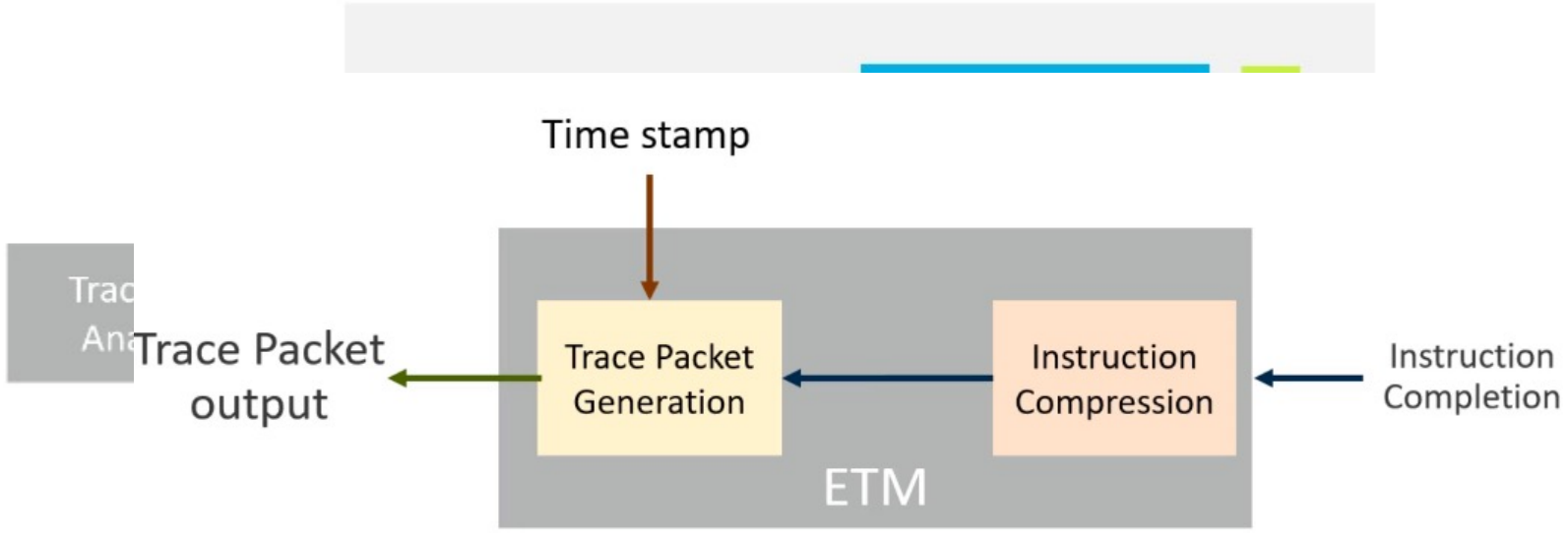
Fundamentals of ***Debug*** *and* ***Trace*** *on a* ***System On-Chip***

Types of Debugging (1) – Invasive Debugging

- Halts the Processor while running
- R/W access to register values
- Instructions run at the processor core level



Types of Debugging (2) – Non-Invasive Debugging



- Consists of detailed software behaviour observation
- Captures execution information
- No R/W Access to Registers

ARM Juno r2 Development Board

About the Juno Development Board

- The Juno Development Board is an Armv8-A AARCH64 development board
- Supports development and testing for next-generation SoC designs
- Runs on an ARM Processor

- Essentially like a Raspberry Pi

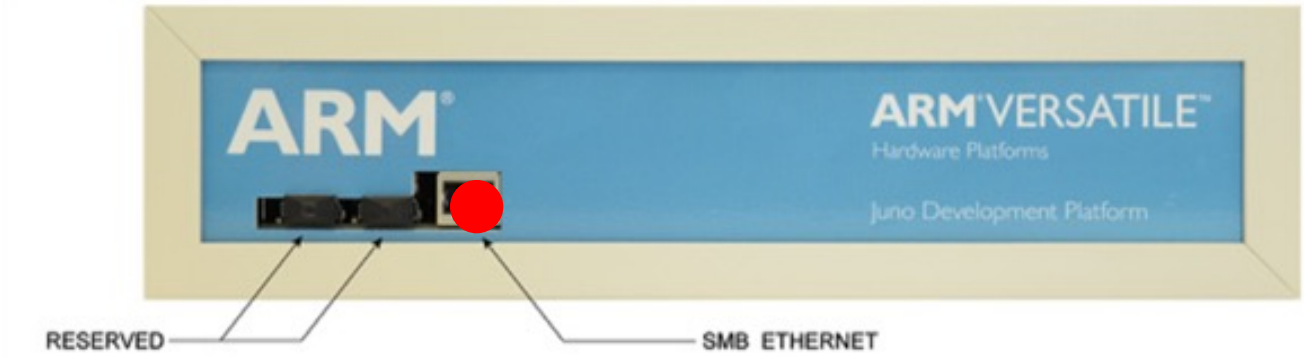
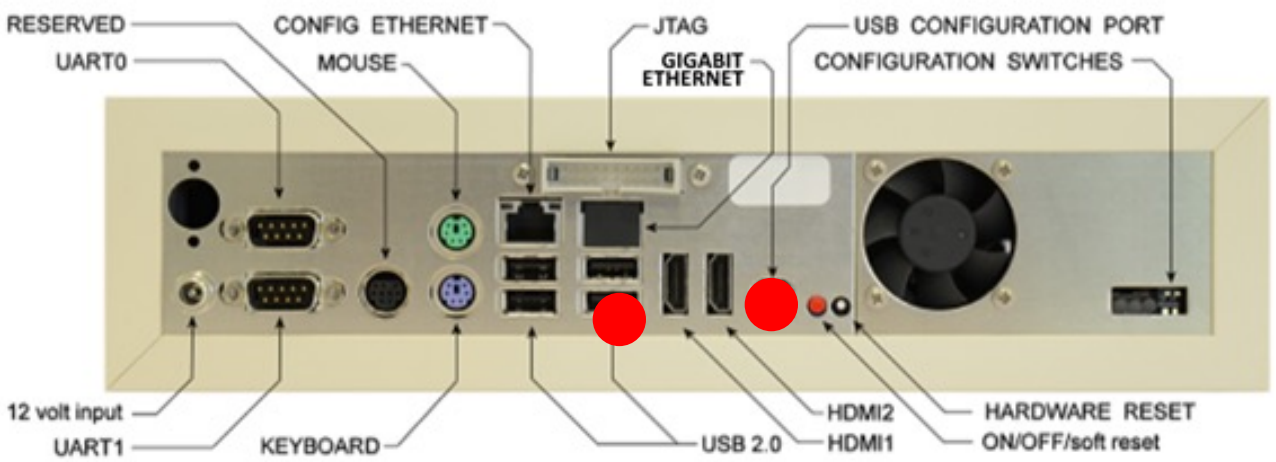


Figure 1-1 V2M-Juno motherboard front panel

Figure 1-2 shows the rear panel of the V2M-Juno motherboard.



ARM CoreSight and CoreSight Access Library

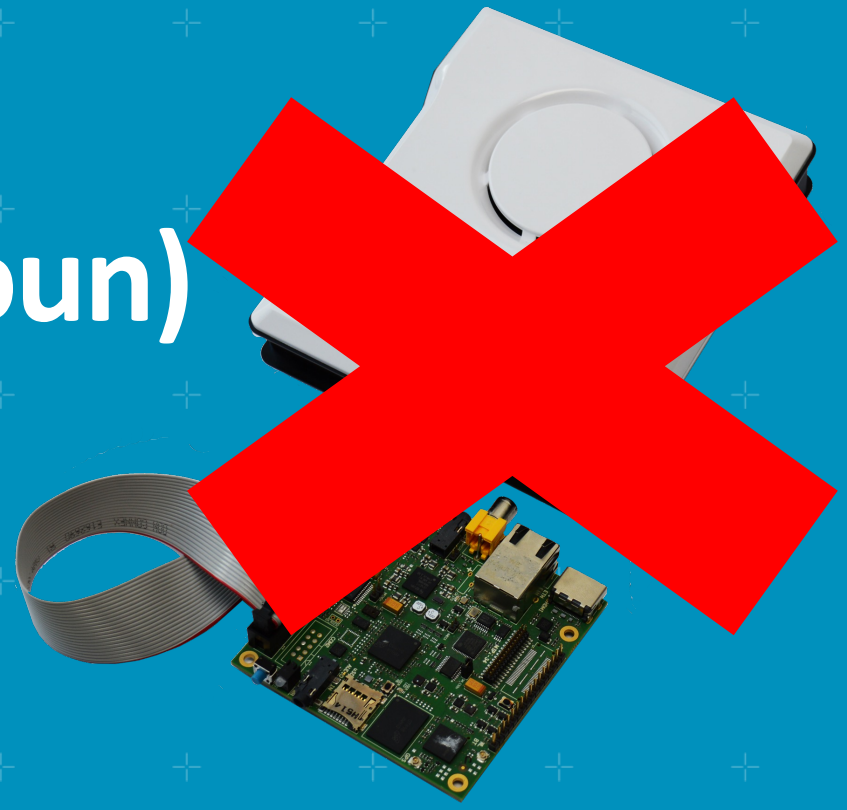
arm

CoreSight **(noun)**

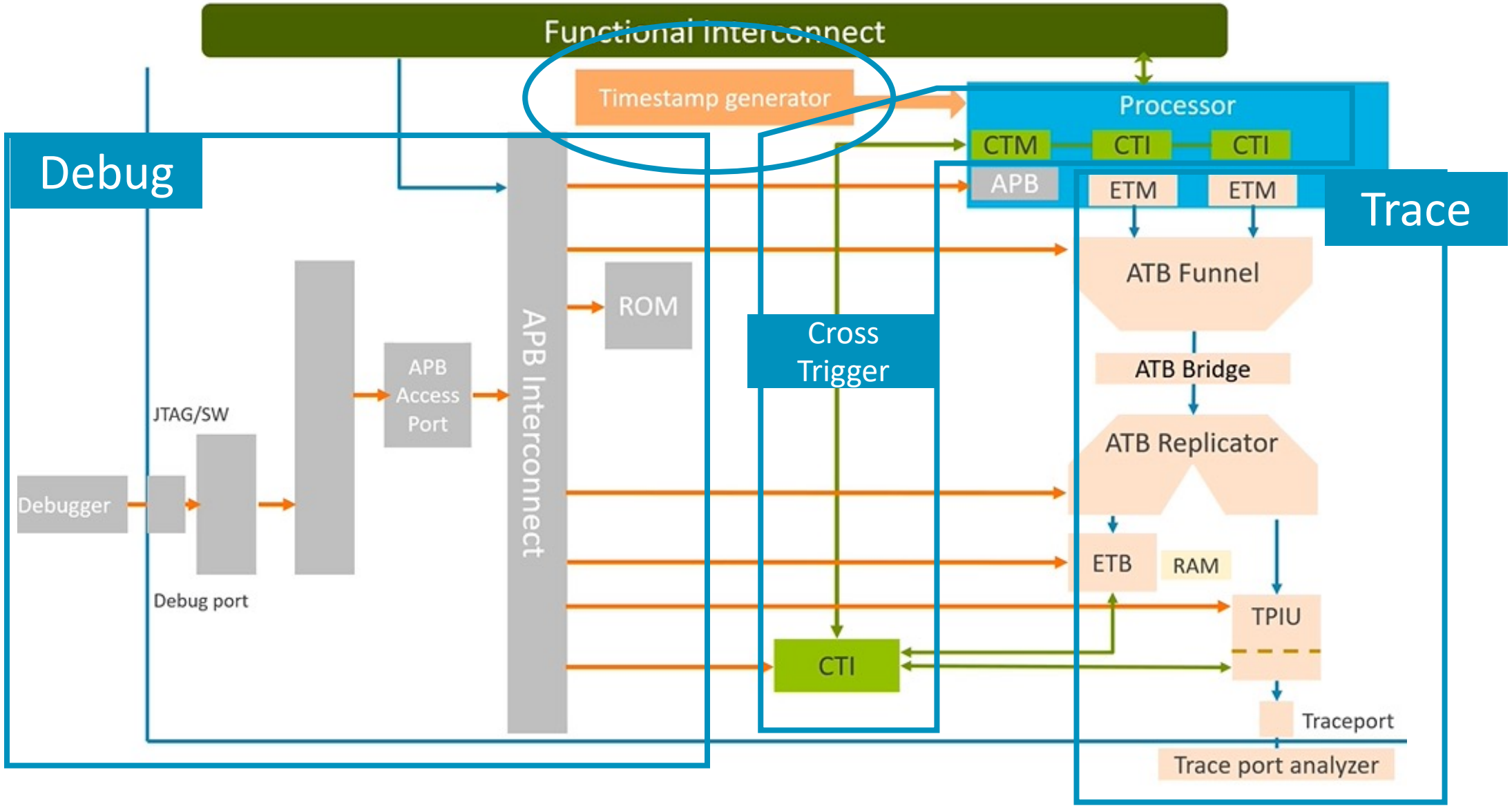
part of the ARM *SoC* architecture,
allows debug and trace functionality

CoreSight Access Library **(noun)**

an API written in C, enabling user code to
directly interact with *CoreSight*
components without a debug probe



The CoreSight™ Architecture & Components



arm

Project Objectives (explicit)

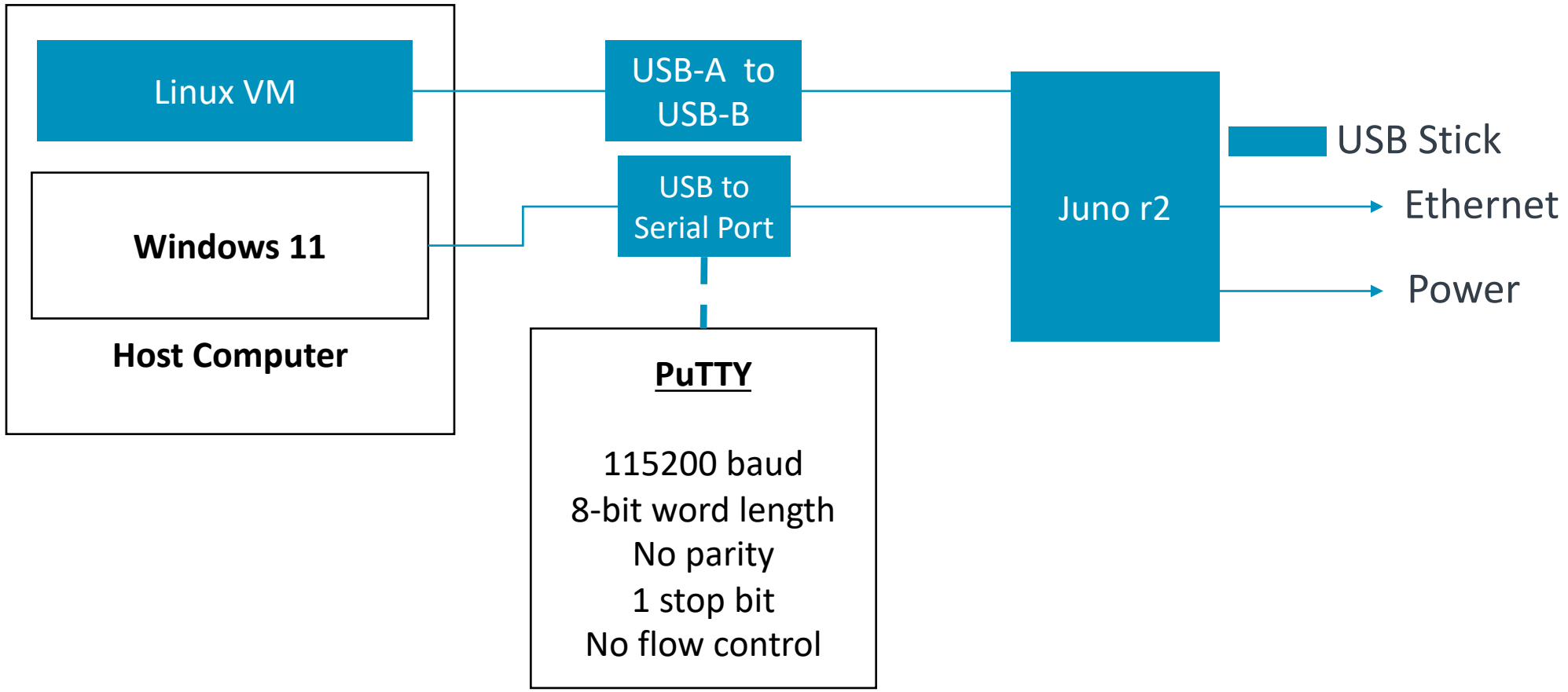
- The features and machinations of the **CSAL** (CoreSight Access Library) are esoteric within the ARM **debugging** team.

- By utilizing **CSAL** on the **Juno**, this project aims to **increase the understanding** of CSAL within the **debugging team**, so **CSAL** can be used in debugging cases

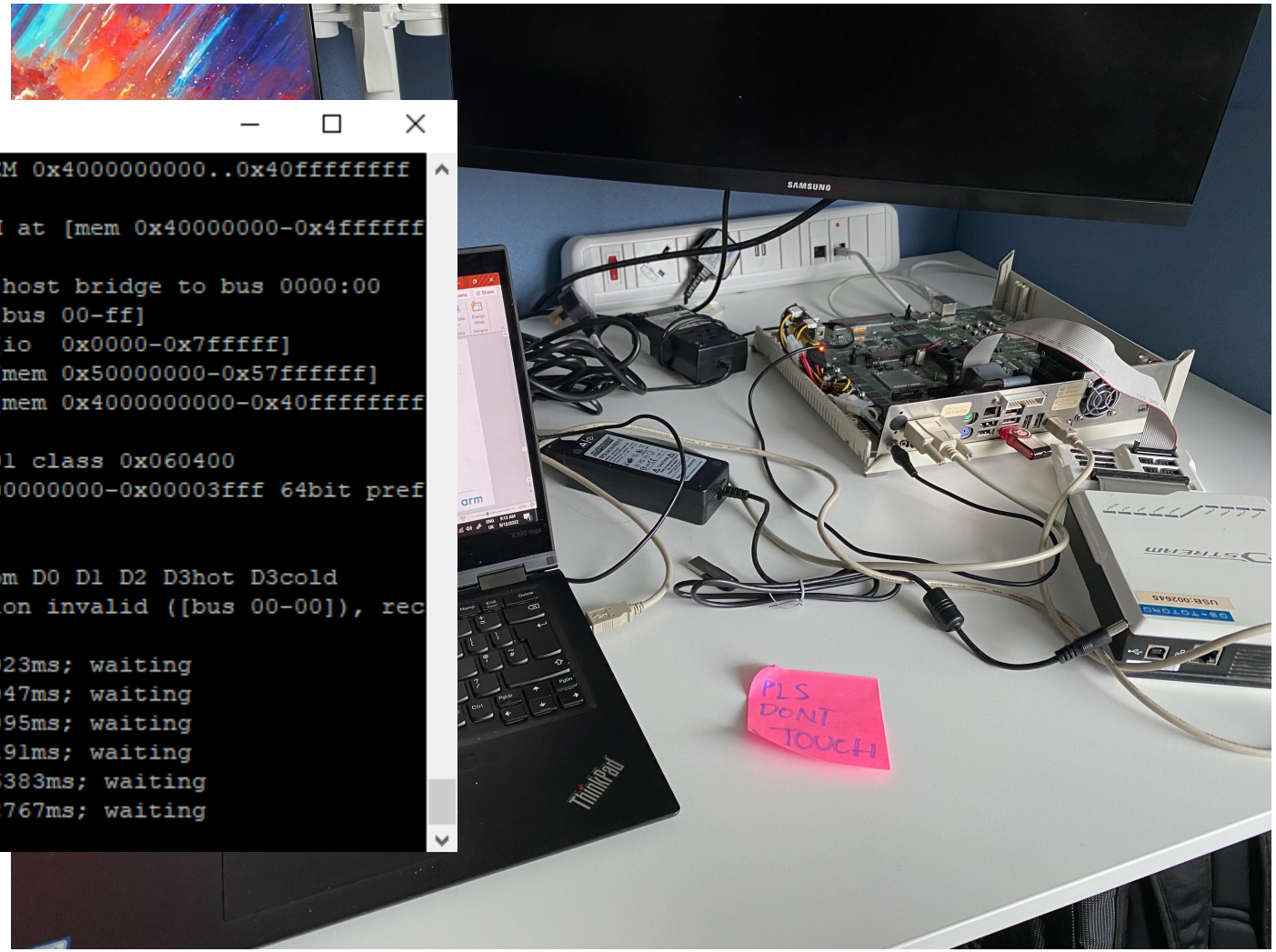
arm

Step 1:
Interfacing with the Board from your Host PC

My Workspace Setup:



In the real world:



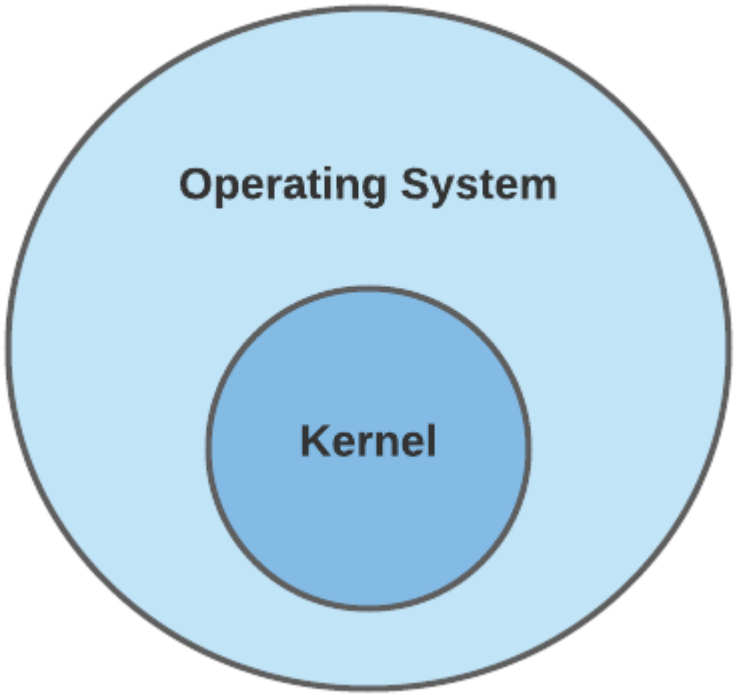
arm

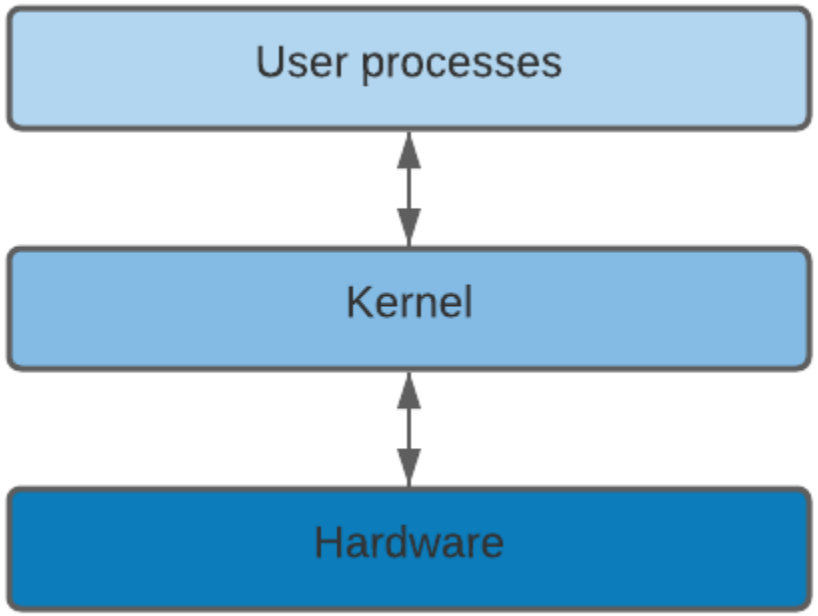
Step 2:
Flash a supported **Kernel** with Appropriate **Filesystem** and **Config**

Notice: *From here on out, there was little to no documentation guiding me, so a lot of the steps were based on trial and error that in the end worked*

Operating Systems *and* ***File Systems*** *in the Context of this Project*

Operating System (OS) and Kernel





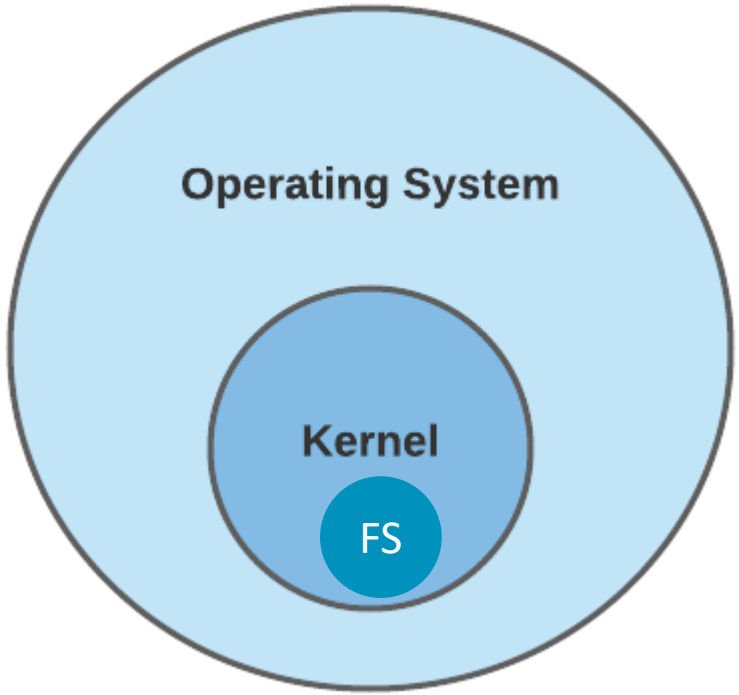
Kernel is part of the **Operating System**

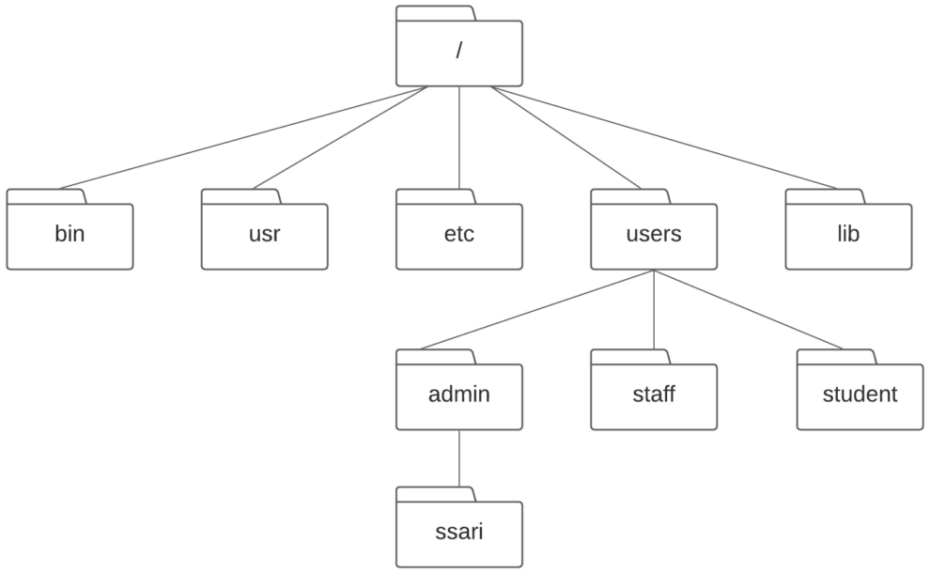
Resides between **User Processes** and **Computer Hardware**

Kernel allocates **hardware resources** to user process

Kernel and File System (fs)

- File System is part of the **Kernel**
- FS describes how files are stored in the system
- FS enables interaction between storage media & other resources





Supported Linux File Systems

On the Juno r2 Development Board

BusyBox

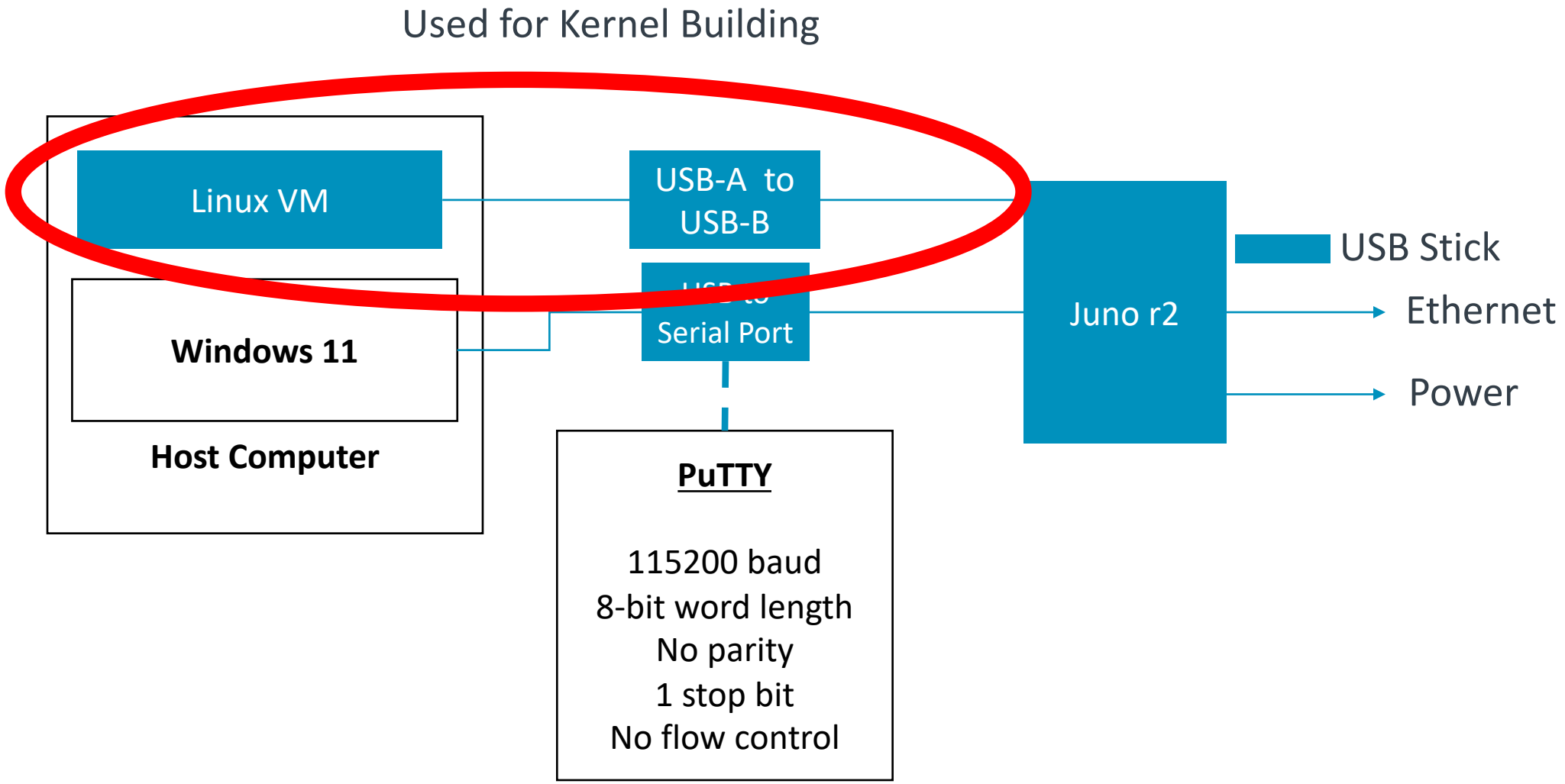
- Basic root file system support
- Simple terminal commands, e.g.
  `cat ls vi help ping`
- No eth0 Support
- No boot from USB support
- No compiler

OpenEmbedded

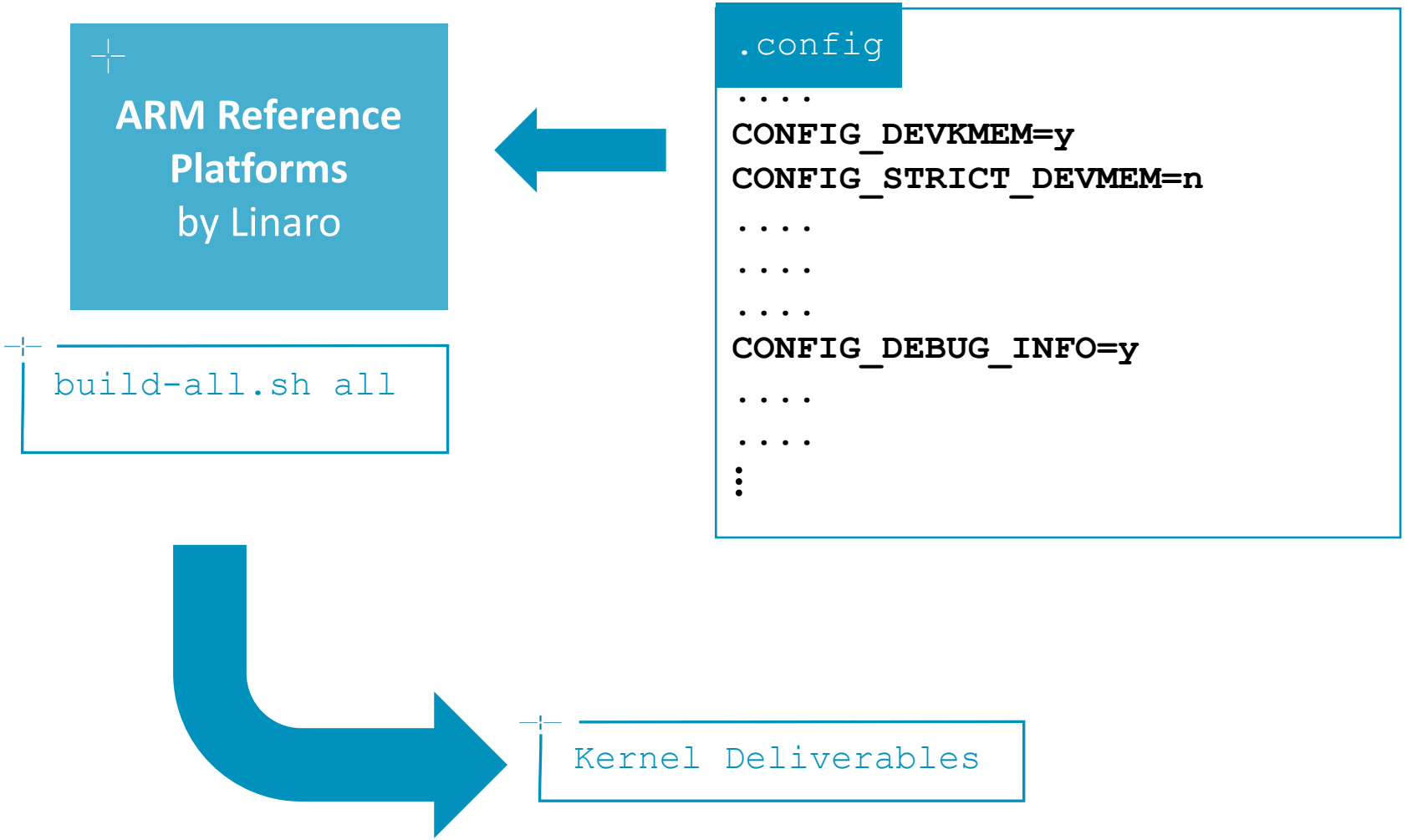
- Has more features than BusyBox FS
- Support for eth0, boot from USB, C Compiler
- Python Interpreter Support
- Git and SSH Support
- **Essentially, everything that was needed to generate trace**

Android

- (not explored in this project)



Building a Kernel with OE Filesystem

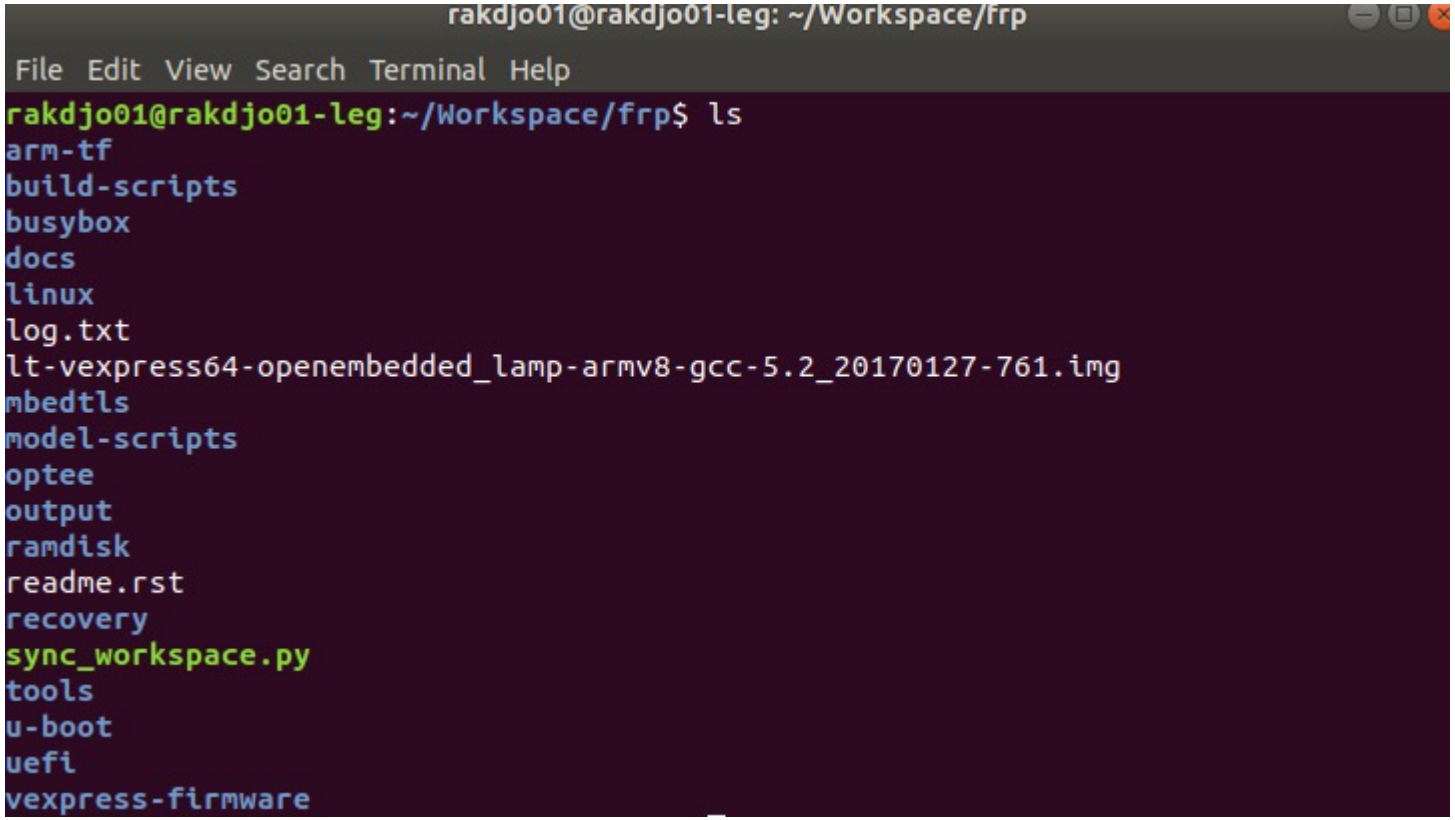


Clone & Setup

```
$ git clone https://git.linaro.org/landing-teams/working/arm/arm-reference-platforms.git
$ cd arm-reference-platforms/
$ python3 sync_workspace.py

## Please select a Platform: 1 - Development Boards
## Please select a Platform: 1 - Juno
## Please select a Platform: 1 - Juno with 64 bit software-stack
## Please specify: 1 - Build from source
## Please select an environment: 1 - Linux Kernel & userspace FS
## Please select your kernel: 2 - Linaro/ArmLT Latest Stable Kernel
## Please select your filesystem: 2 - OpenEmbedded
## Please select your filesystem: 1 - LAMP
```

Resulting Directory



Change **STRICT_DEVMEM** and **DEVKMEM** flags before building

```
$ gedit ./linux/linaro/configs/distribution.conf
```

Change the flags accordingly, will need later

Build Kernel

```
$ chmod a+x build-scripts/build-all.sh
$ build-scripts/build-all.sh all
```

Move Firmware & Kernel to Juno Mounted Storage

```
$ sudo mv output/juno/juno/firmware/juno-oe-uboot/* /media/user/JUNO/
$ sudo mv output/juno/components/linux/* /media/user/JUNO/SOFTWARE/
```

arm

Step 3:
Format USB Stick to **ext4**, then burn the OE Image

Format to ext4

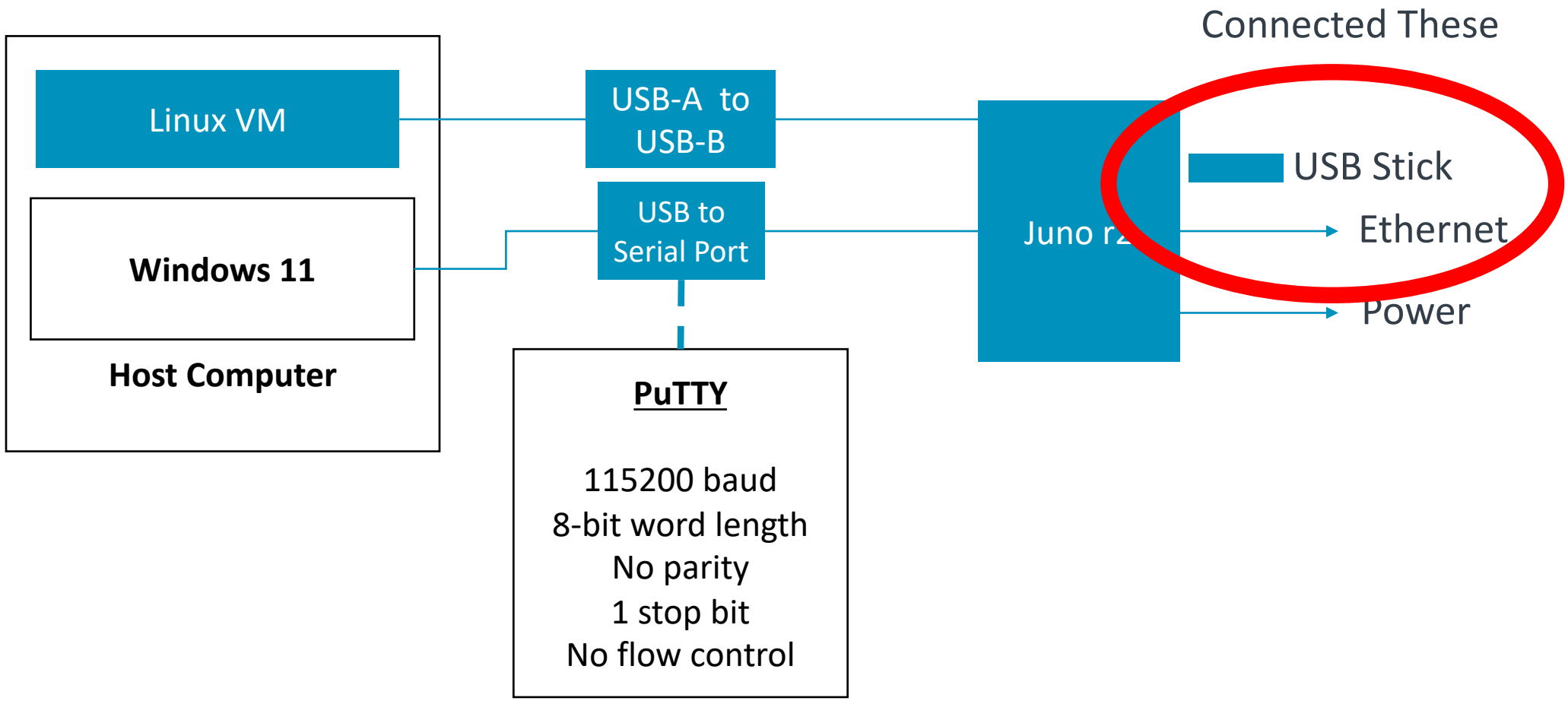
```
$ sudo mkfs -t ext4 /dev/sdX
```

Burn Image file

```
$ sudo dd if=/dev/zero of=/dev/sdX bs=1M
$ sudo dd if=lt-vexpress64-openembedded ... .img of=/dev/sdX bs=1M
```

arm

Step 4: Boot Board & Connect USB Stick + Ethernet Cable to Board



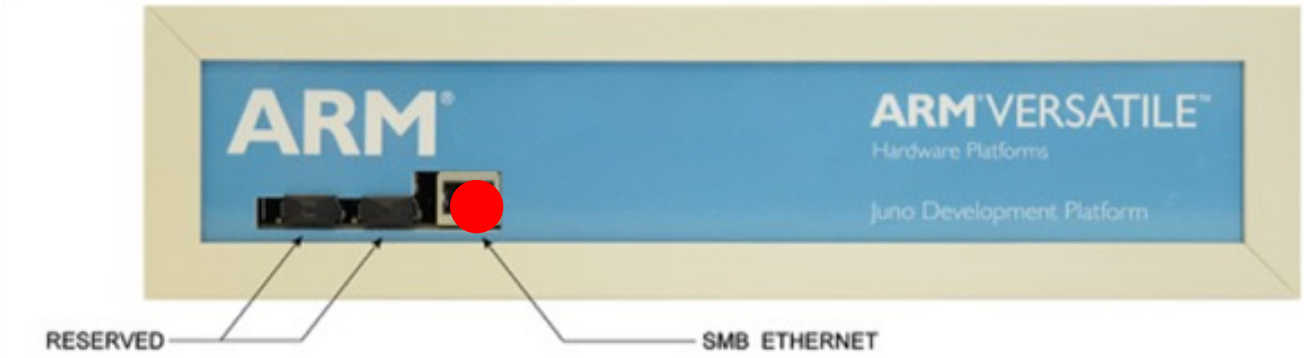
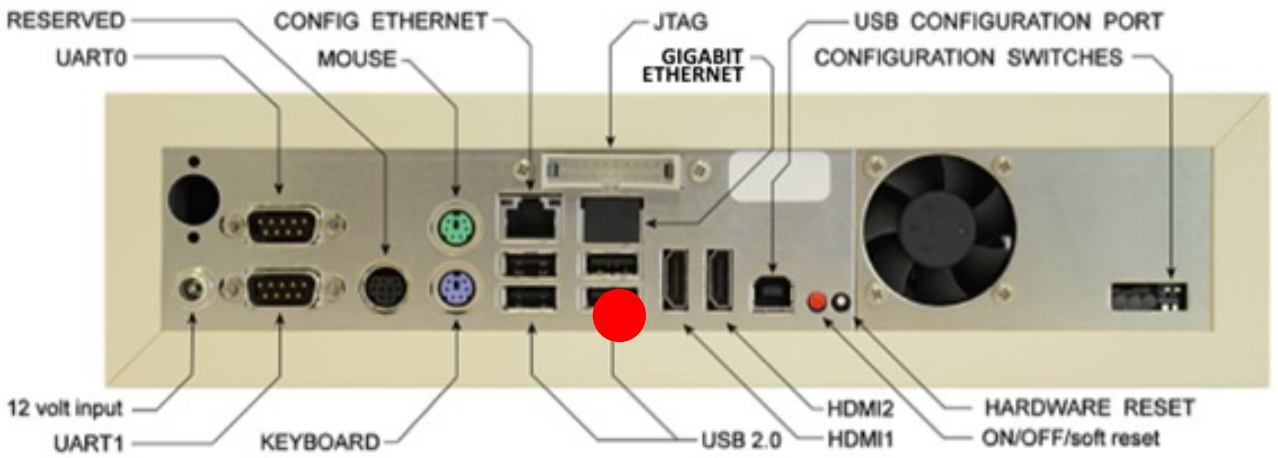


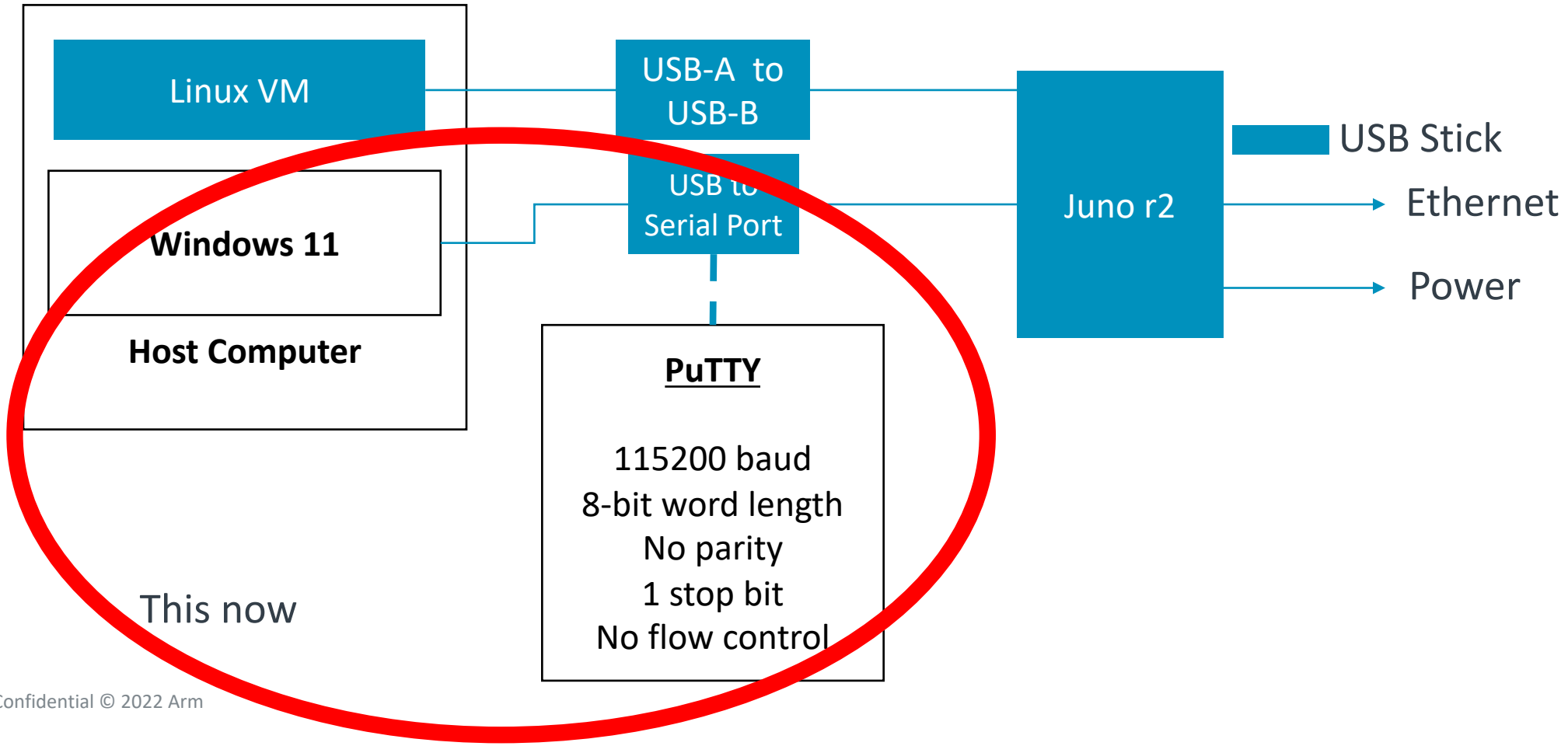
Figure 1-1 V2M-Juno motherboard front panel

Figure 1-2 shows the rear panel of the V2M-Juno motherboard.



arm

Step 5:
Clone, and Build the CoreSight Access Library for the Board



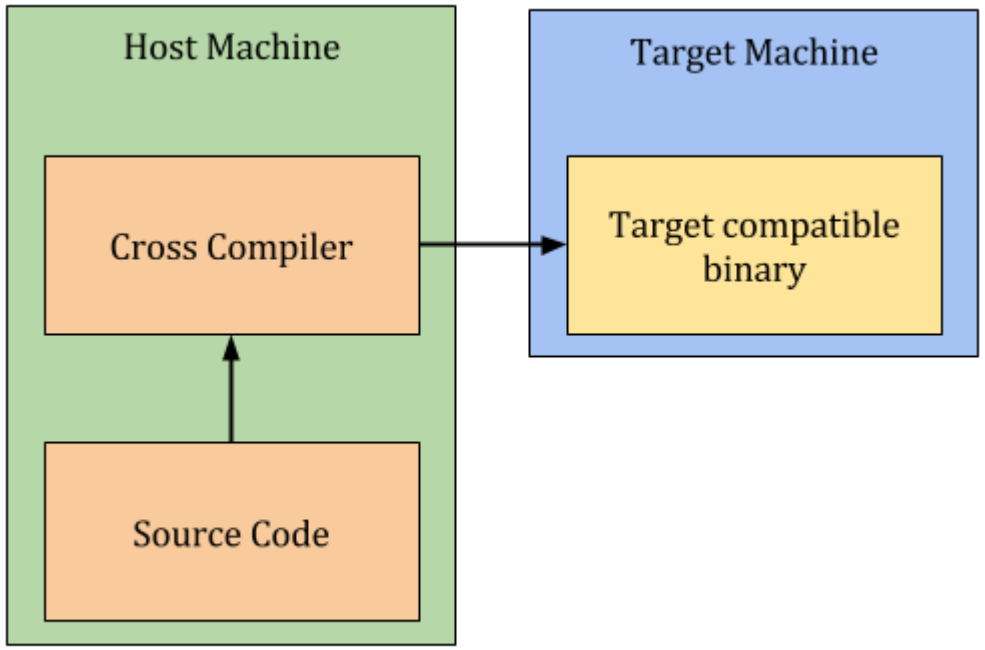
Clone & Build

```
root@genericarmv8:~# git clone https://gi[redacted].com/rakhadjo/CSAL.git
root@genericarmv8:~# cd CSAL && make
root@genericarmv8:~/CSAL# sh demos/juno_demo_setup/no_cpu_idle.sh
```

```
root@genericarmv8:~/CSAL# ls bin/arm64/rel/
  clean-snapshot.bash    etmdemo          save-snapshot.bash     tracedemo
  csls                   makefile         trace_cti_stop
```

Side Note (1) – Cross Compilation

i.e. what if your target device doesn't have a compiler on its own?



Cross Compiler operation

Building CSAL on the host PC

- CSAL can be built on x86 devices (e.g. i7) targeting ARM devices
- Needs a **Cross Compiler**
- Result on the Juno Board:
  - Missing `lib6` library to run compiled program
  - Needs `apt` to install `lib6`
  - Package `apt` not present on this distro
- CSAL **should** be built on the board as the FS came with a Compiler

Side Note (2) – Idle CPUs

The Double-Edged Sword that is Linux

- Good Power saving principles adopted by Linux
- But unused CPUs and Peripherals will switch off
- Idle CPUs mean losing connection to debug/trace

Solution:

```
#!/bin/sh
for i in 0 1 2 3 4 5 ; do
    for j in 0 1 2 ; do
        echo 1 > /sys/devices/system/cpu/cpu$i/cpuidle/state$j/disable;
    done;
done
```

arm

Step 6:
Run CSAL on the Board

arm

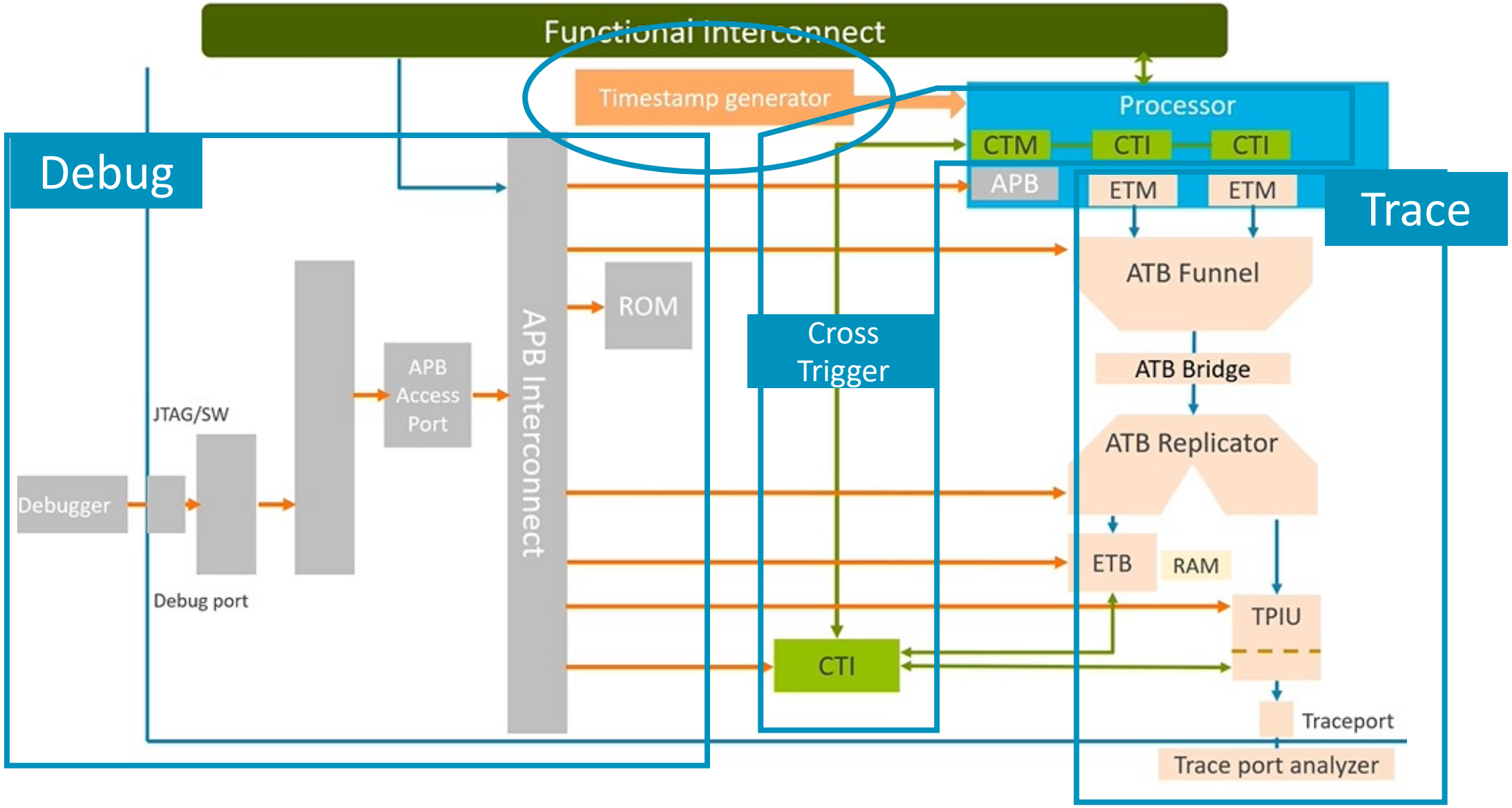
CoreSight **(noun)**

part of the ARM *SoC* architecture,
allows debug and trace functionality

CoreSight Access Library **(noun)**

an API enabling user code to
directly interact with *CoreSight*
components without a debug probe

The CoreSight™ Architecture & Components



arm

CoreSight Access Library (CSAL) Programs

csls

Lists all CoreSight Components found on the Device and the ROM Table

cs_etm_config_demo

Outputs the configuration of ETM or PTM components

tracedemo

Capture trace on a running Linux System

tracedemo_cti_stop

Capture trace on a running Linux System, but injects trigger-packet into trace for each core, using Cross-Trigger Interfaces. Triggers ETB to flush and halt capture

Refresher on Tracing

- A form of non-invasive debugging
- Capturing data illustrating how components are working
- Can:
  - Diagnose runtime problems
  - Performance Measuring
  - View operation on a system-level
- Types of Tracing:
  - Instruction Trace
  - Data Trace
  - Instrumentation Trace
  - System Trace

Recall the Output when Building the Library…

```
root@genericarmv8:~# git clone https://github.com/rakhadjo/CSAL.git
root@genericarmv8:~# cd CSAL && make
root@genericarmv8:~/CSAL# sh demos/juno_demo_setup/no_cpu_idle.sh
```

```
root@genericarmv8:~/CSAL# ls bin/arm64/rel/
  clean-snapshot.bash    etmdemo          save-snapshot.bash    tracedemo
  csls                   makefile         trace_cti_stop
```

…running `tracedemo` creates `.ini` and `.bin` files

```
root@genericarmv8:~/CSAL# cd bin/arm64/rel/
root@genericarmv8:~/CSAL/bin/arm64/rel# ./tracedemo -board-name Juno
```

```
root@genericarmv8:~/Workspace/WS2/CSAL/bin/arm64/rel# ls
clean-snapshot.bash  cpu_5.ini      device_7.ini     save-snapshot.bash
cpu_0.ini            csls           device_8.ini     snapshot.ini
cpu_1.ini            cstrace.bin    device_9.ini     trace.ini
cpu_2.ini            device_10.ini  etmdemo          trace_cti_stop
cpu_3.ini            device_11.ini  kernel_dump.bin  tracedemo
cpu_4.ini            device_6.ini   makefile
```

…used for tracing in ArmDS

arm

Side Note – Importance of disabling `STRICT_DEVMEM` Flag

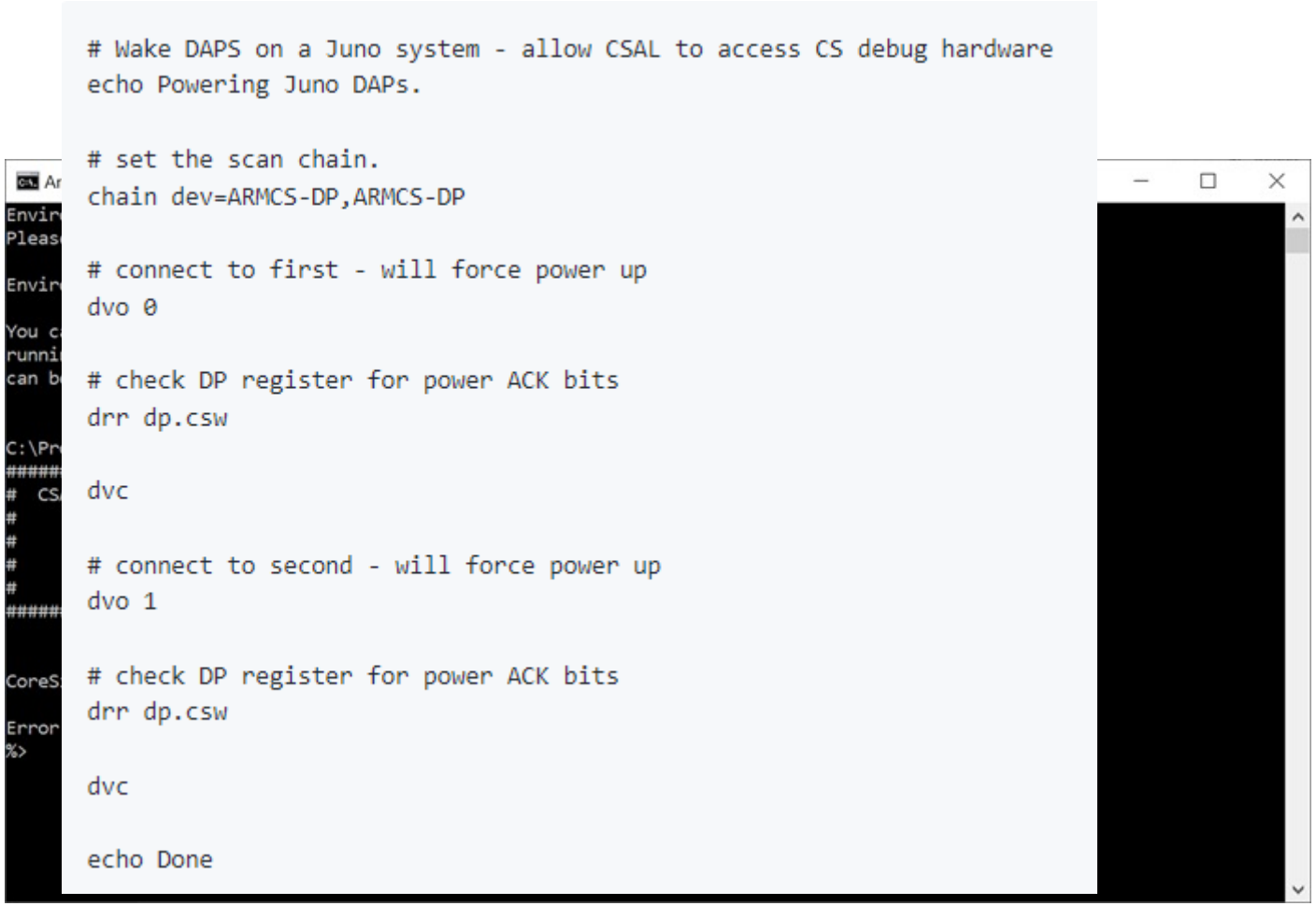
```
CSDEMO: Enabling trace...
CSDEMO: CTI settings....
CSDEMO: Configured and enabled trace.
CSDEMO: Trace configured
dumping config with No ITM
Attempting to dump kernel memory from 0xFFFFFF8010081000 to 0xFFFFFF8010781000
can't open /dev/kmem, trying /dev/mem instead...
Kernel code entry in /proc/iomem starts at 0x80080000
Found kernel symbol @ 0xFFFFFF8010081000 as kernel VA
Using 0x80080000 as mapping address, 0x700000 size
Cannot mmap device at 0xffffff8010081000, errno=1
so cannot dump kernel memory automatically to the file kernel_dump.bin.
This may be because the kernel has been built with CONFIG_STRICT_DEVMEM set.
Try rebuilding the kernel without this flag, alternatively:
1) Use DS-5 Debugger to dump this kernel memory from the target
2) Extract this kernel memory from the kernel Image
Program execution will now continue to generate the other necessary files...
```

TLDR: Kernel Memory Dump is Required for to read snapshot trace data

arm

Side Note (2) – Dumping Kernel Memory using DS-5 & DSTREAM

- Used the CSAT tool by the DSTREAM
  - To dump kernel memory
- Ran the following commands to dump the kernel memory through a source file
- …But resulting kernel dump file couldn't be found anywhere
- So I rebuilt the kernel



```
# Wake DAPS on a Juno system - allow CSAL to access CS debug hardware
echo Powering Juno DAPs.

# set the scan chain.
chain dev=ARMCS-DP,ARMCS-DP

# connect to first - will force power up
dvo 0

# check DP register for power ACK bits
drr dp.csw

dvc

# connect to second - will force power up
dvo 1

# check DP register for power ACK bits
drr dp.csw

dvc

echo Done
```

Anyways… here's the `tracedemo` program output

```
1734     Buffer RAM size: 65536
1735     Bytes to read in buffer: 65536
1736     Buffer has wrapped: 1
1737   ** 65536 bytes of trace
1738   The first 256 bytes of trace are:
1739    25 10 FC F8 FA D7 D6 D7 21 D6 04 FD FC DB 9A 54  5A 6E 52 10 F6 9A 25 1F DA 95 3C FE 21 09 54 AC
1740    10 F8 25 95 40 DF FC D7 D6 D7 D6 F9 F6 91 FE 40  21 FB DA 95 25 4B 90 FF F6 90 DA 95 FA EA DA 9E
1741    9A 04 21 6B F4 FE 94 0F F6 95 9E 27 FC FF F6 DE  9A 54 66 53 10 F7 94 E6 25 E9 12 10 DE C3 94 BA
1742    82 5E 21 FB FA F9 FC 95 25 E0 DA 9A 3C 0B 9E CF  10 DA 9A 7E 5A 12 21 89 EA F8 F6 9A 7C 00 54 A4
1743    25 10 DA 95 21 93 10 F9 FE 9A 12 6A 52 10 25 64  5E F8 E0 FB DE F7 9A 1E 21 E1 F6 95 EA E7 F6 E0
1744    94 97 25 1A 9E 10 E2 DA 94 14 F6 9A 4E 57 12 79  21 EA FA E0 25 F7 10 FB FE F5 94 0F FC DB 94 E6
1745    04 F7 21 95 94 FD E8 E0 25 1C F8 F5 D4 F8 F6 E6  9A 1E 21 E1 F6 9A 25 50 78 80 10 DB 9A 27 4C 9C
1746    12 10 FE 95 9C E4 F6 95 B0 57 F6 95 58 DB 94 AF  AC 5E FC F7 94 30 FC 95 66 DA 21 52 0C 10 D6 1E
1747   CSDEMO: shutdown...
```

Debugging, without Debug Probe!

```
root@genericarmv8:~/Workspace/WS2/CSAL/bin/arm64/rel# ls
clean-snapshot.bash  cpu_5.ini       device_7.ini      save-snapshot.bash
cpu_0.ini            csls            device_8.ini      snapshot.ini
cpu_1.ini            cstrace.bin     device_9.ini      trace.ini
cpu_2.ini            device_10.ini   etmdemo           trace_cti_stop
cpu_3.ini            device_11.ini   kernel_dump.bin   tracedemo
cpu_4.ini            device_6.ini    makefile
```

But needs all these generated files, supplied to the ArmDS Snapshot viewer

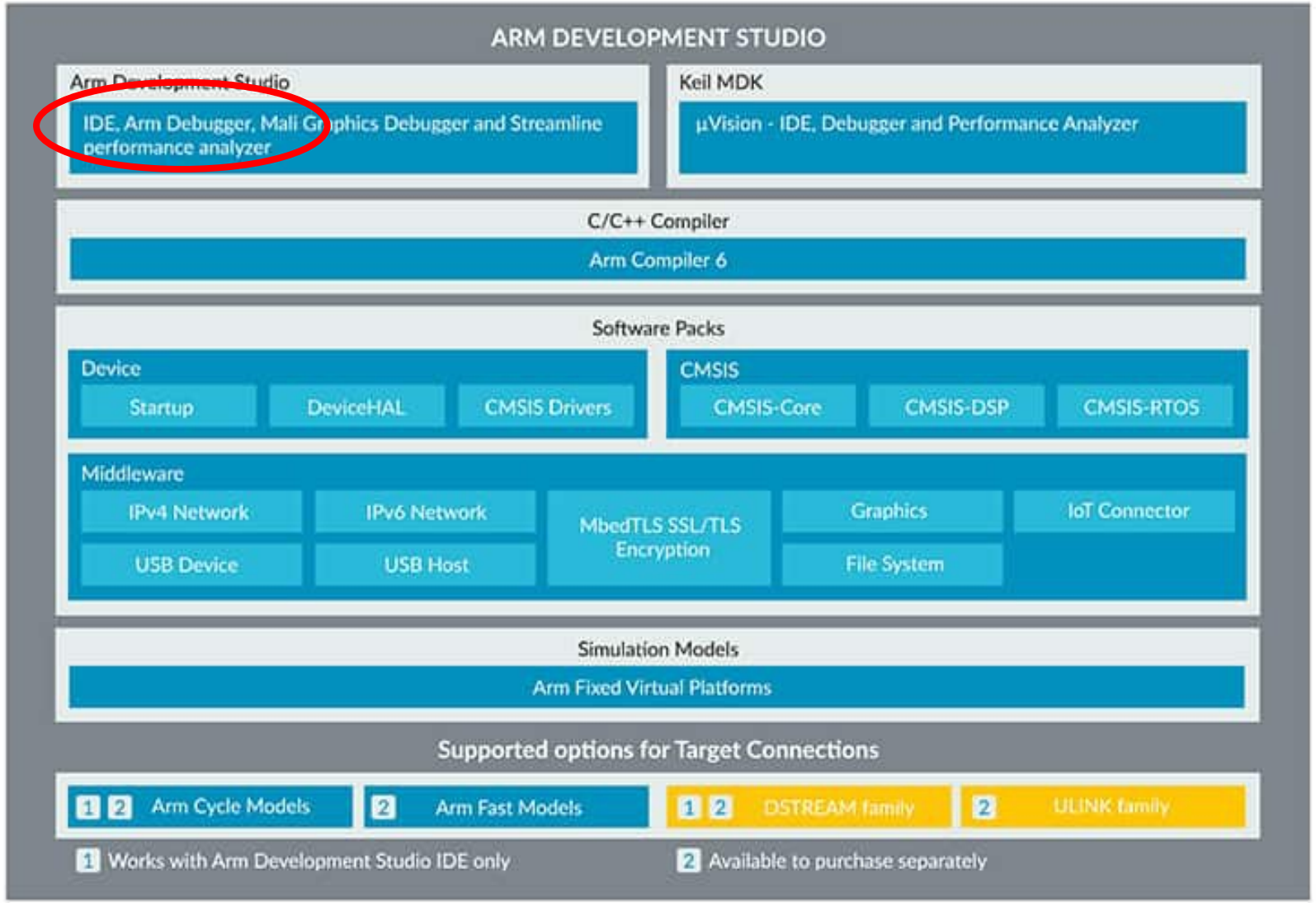
arm

arm

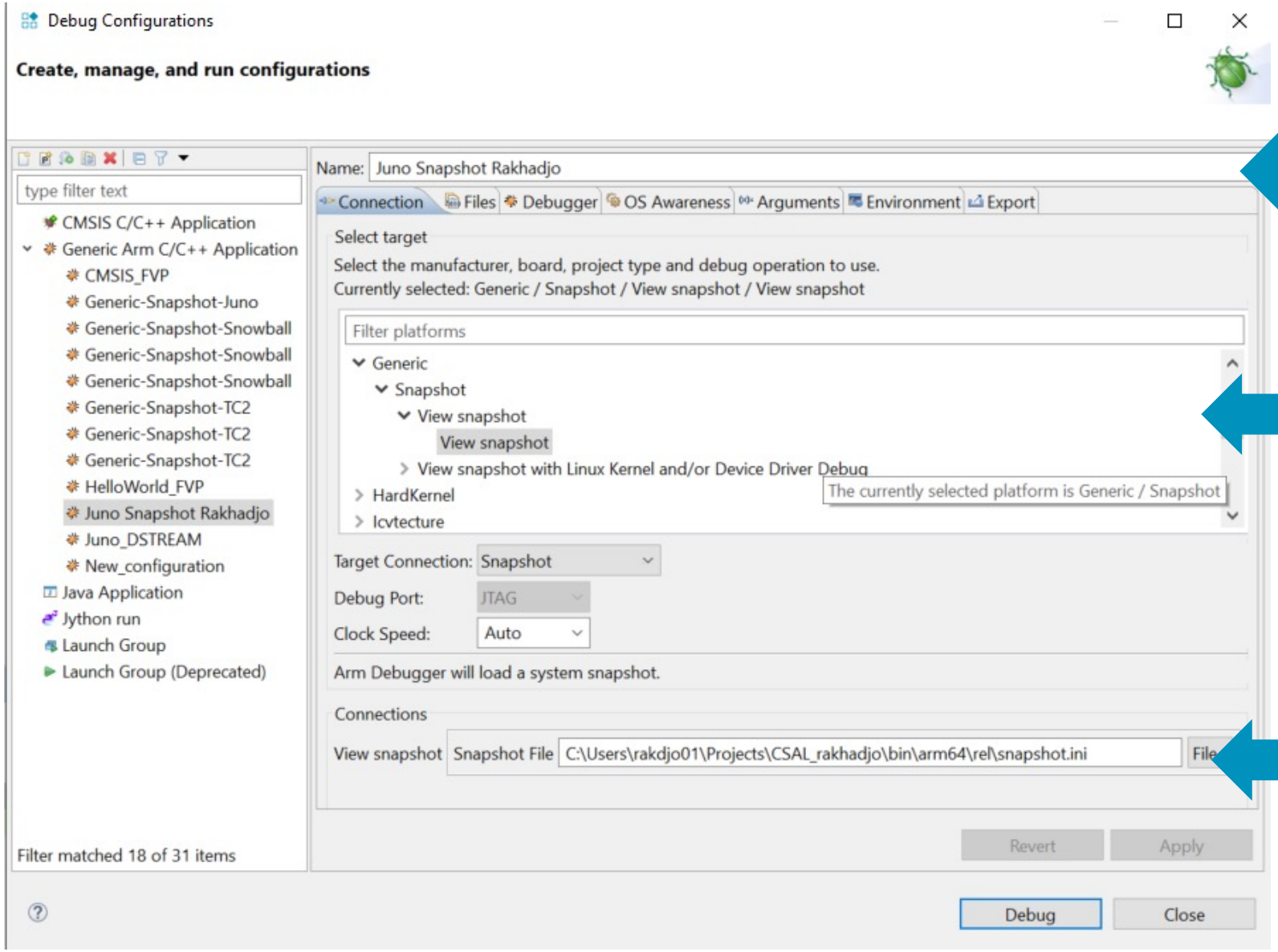
Step 7:
Analyze Trace using ArmDS

ArmDS (Arm Development Studio)

- Eclipse-based C/C++ IDE
- Used to develop ARM SoCs
- Only debugger used in this project



Using Snapshot Viewer for Trace Information (1)



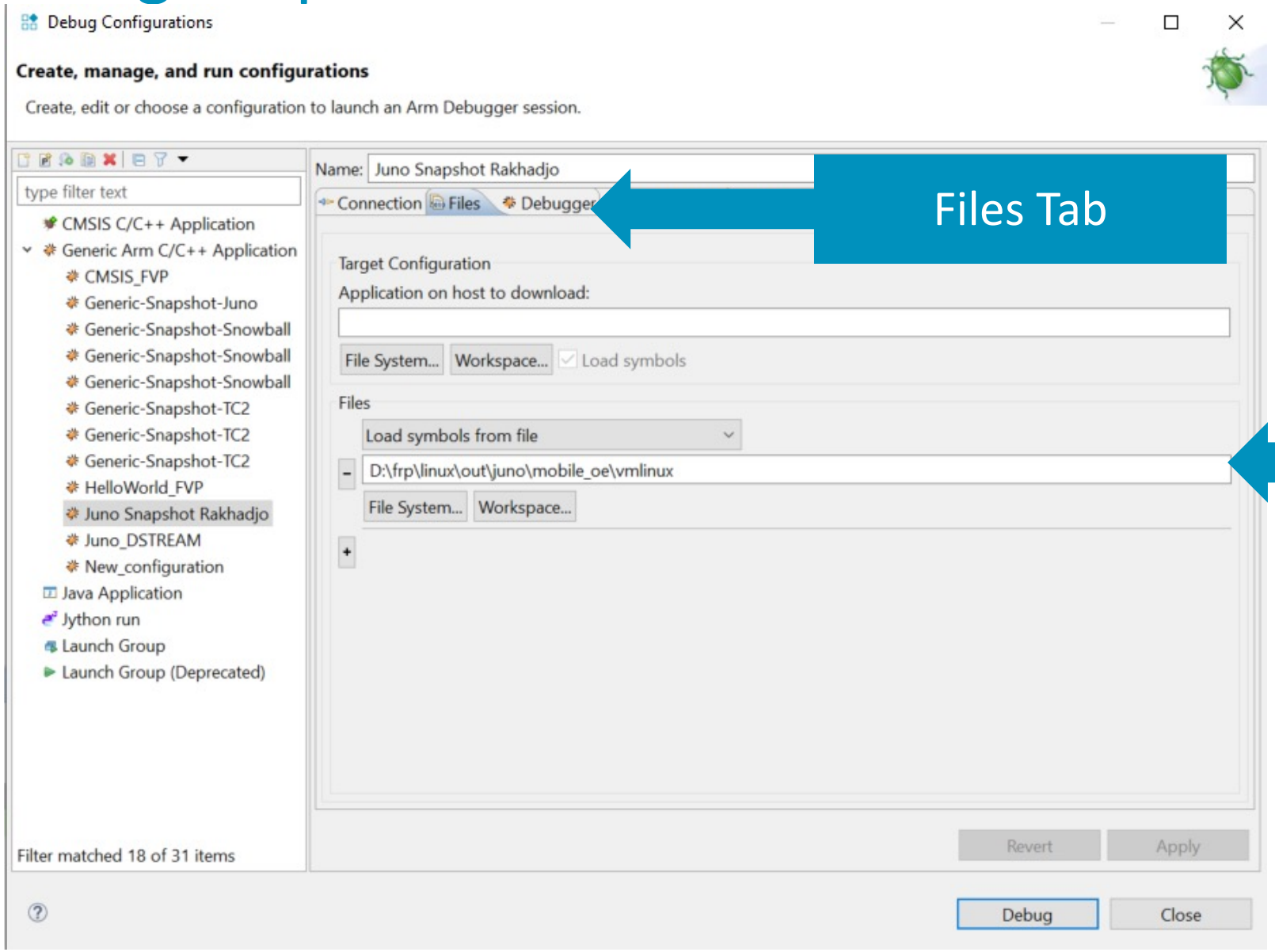
1) Provide Suitable Name

2) General → Snapshot → View Snapshot → View Snapshot

3) Path to generated snapshot.ini

arm

Using Snapshot Viewer for Trace Information (2)



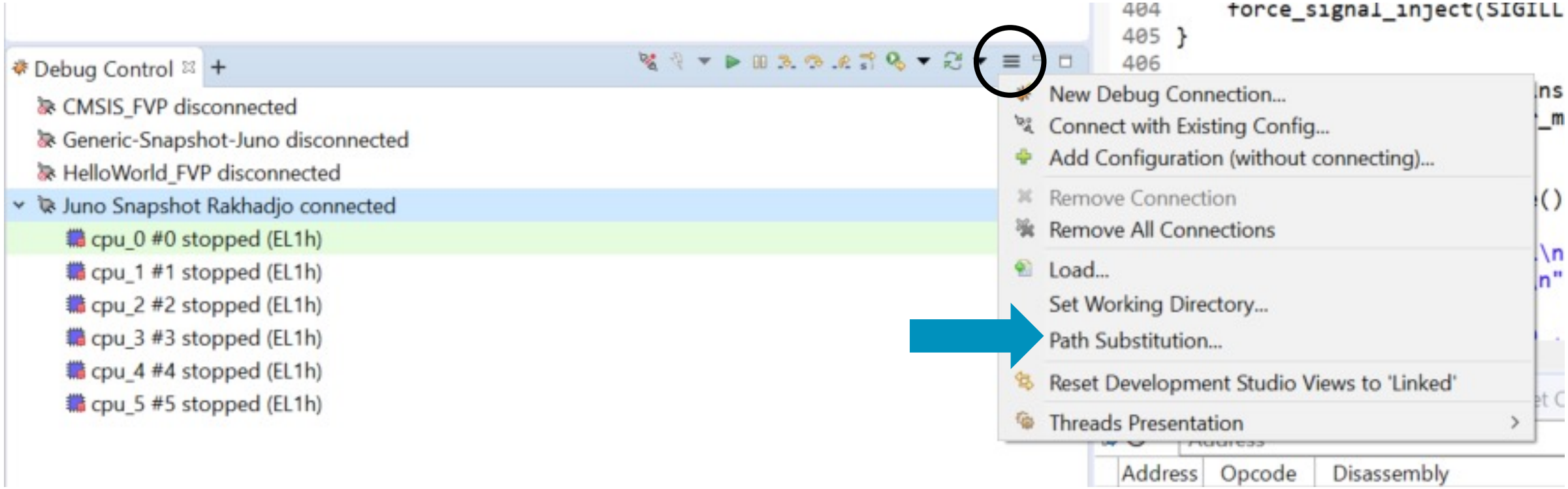
Files Tab

4) Path to vmlinux

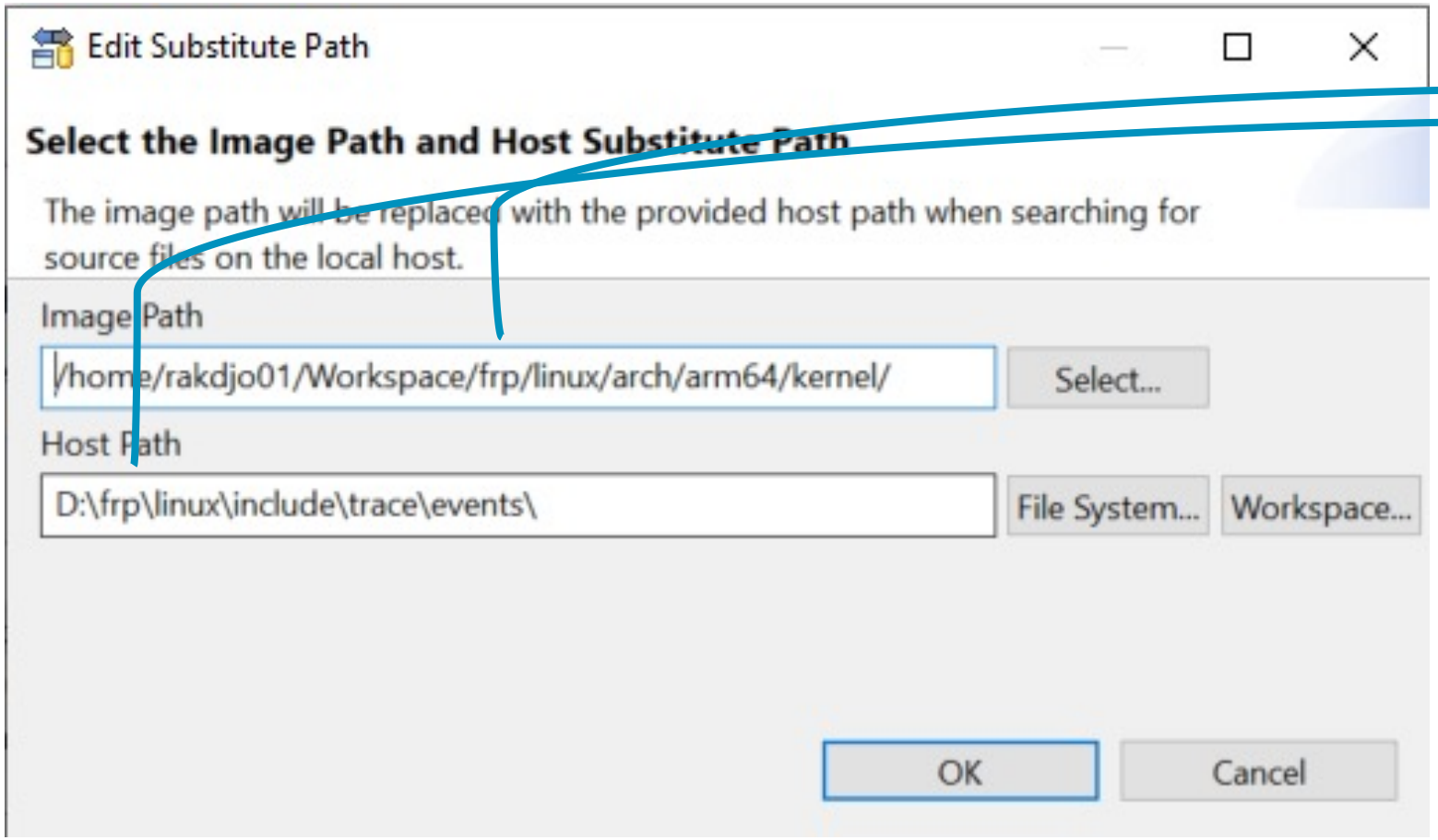
arm

*Next is **Path Substitution** for **Running on Windows** after **Building with Linux VM***

Path Substitution for Source Code (1)



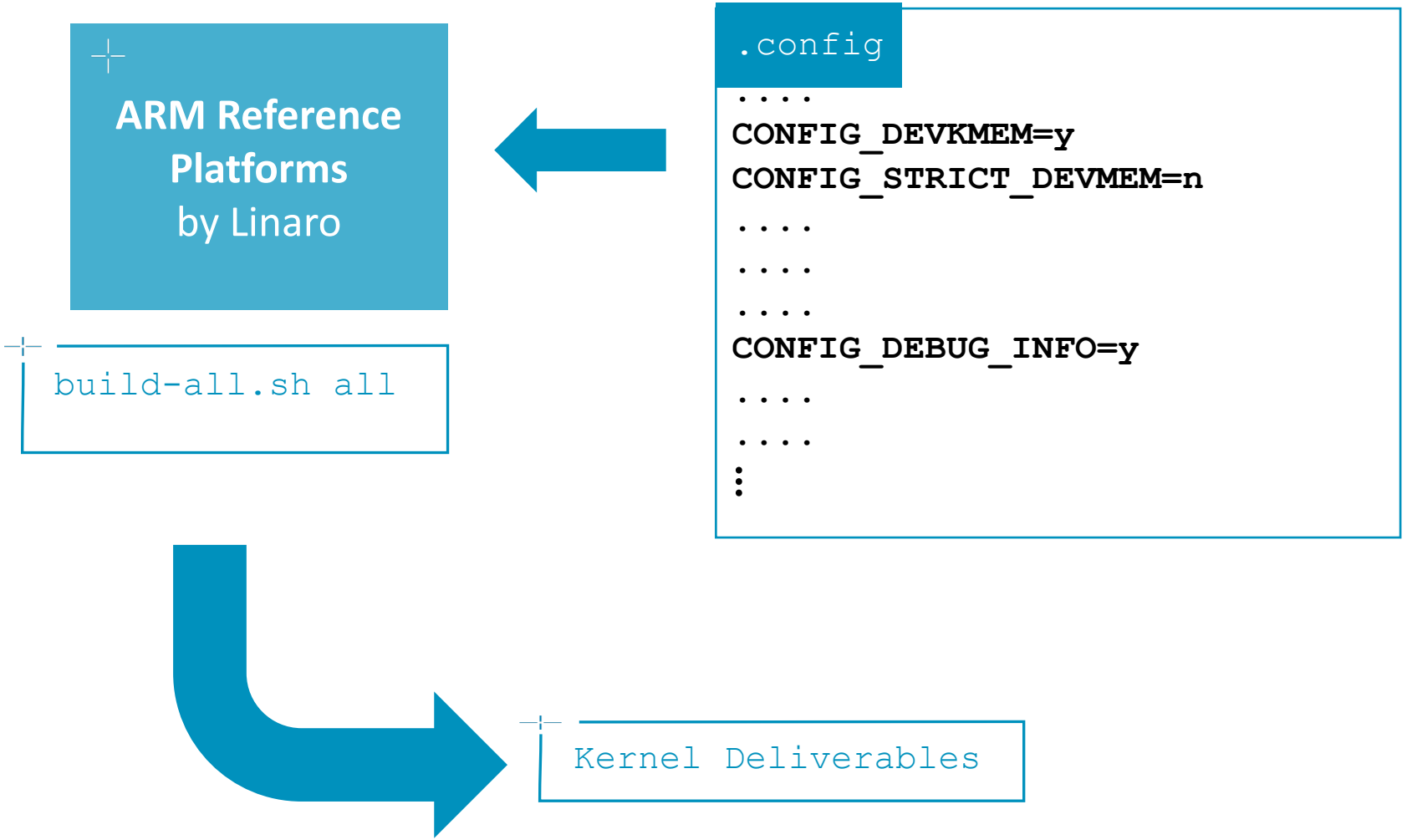
Set Image & Host Path



frp is a folder name I used for Linaro Arm Reference Platforms

arm

…they're the tools I mentioned to build the kernel on this slide

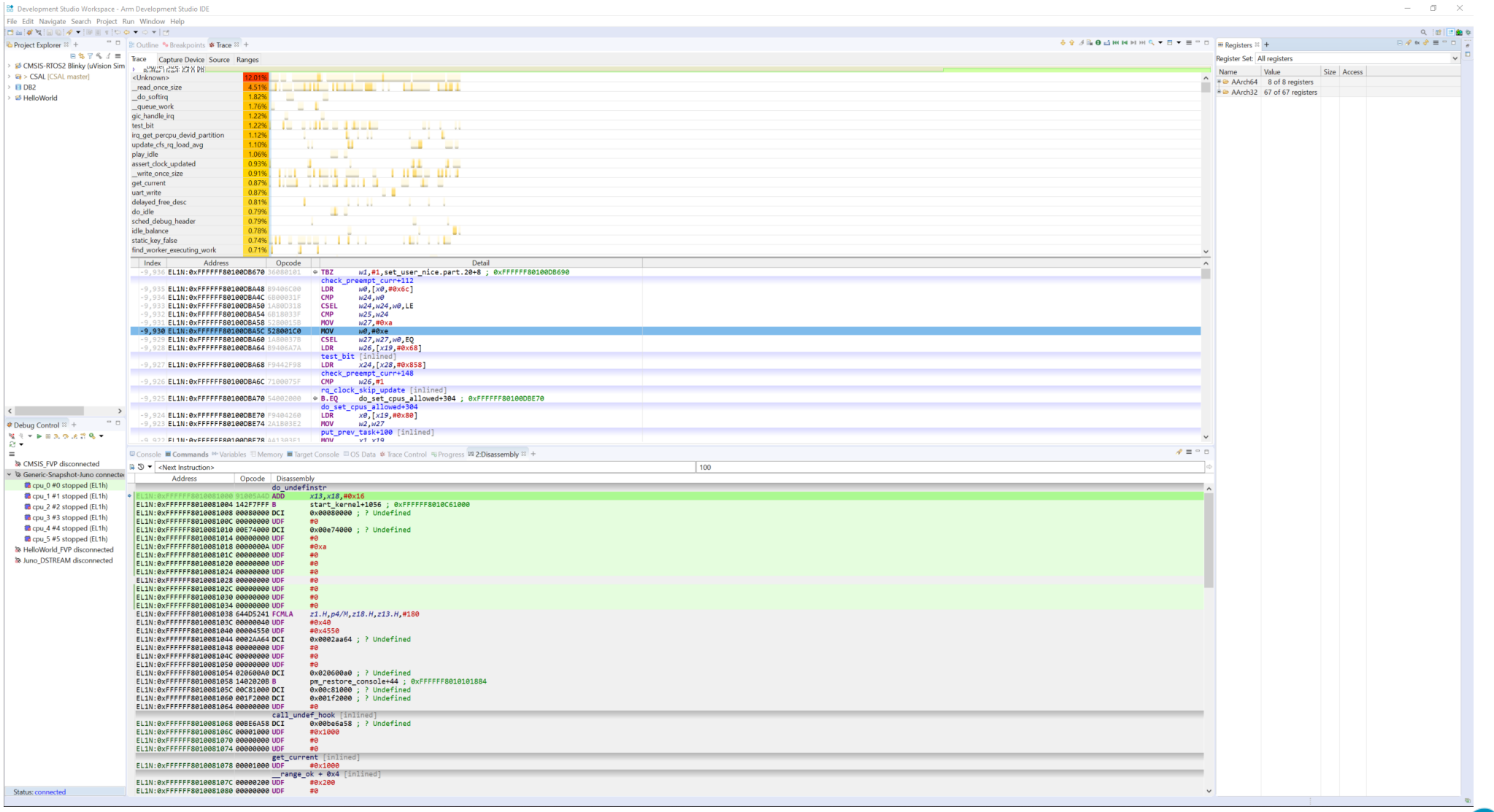


…with this clone command

```
$ git clone https://git.linaro.org/landing-teams/working/arm/arm-reference-platforms.git
$ cd arm-reference-platforms/
$ python3 sync_workspace.py

## Please select a Platform: 1 - Development Boards
## Please select a Platform: 1 - Juno
## Please select a Platform: 1 - Juno with 64 bit software-stack
## Please specify: 1 - Build from source
## Please select an environment: 1 - Linux Kernel & userspace FS
## Please select your kernel: 2 - Linaro/ArmLT Latest Stable Kernel
## Please select your filesystem: 2 - OpenEmbedded
## Please select your filesystem: 1 - LAMP
```

What everything should look like right now



Dangers of CSAL

Potential Security Risk

- CSAL and Linaro Kernel Builder Program is put as Open Source
  - Everyone has access to it
- No authentication needed running the program on the board
- No user authentication when flashing kernel and file system.
- TLDR: Anyone can access the CoreSight system – Escalation of Privileges!

Workarounds(?)

- Some form of authentication before running CSAL
- Linaro deliverables or Board to provide some form of user authentication before building kernels

Retrospectively…

i.e. main challenges

- Addresses not defined properly
  - Needed to look into SoC TRM to find CoreSight ROM Table
- Juno r2 formally unsupported, according to documentation
  - Will issue a pull request to state it supports the r2
  - Along with change requests to default addresses
- Relatively messy documentation
  - Docs either hosted on Arm Community, Linaro deliverables site, both or neither 🥵

From Actions to Reflection

Things I'd do Differently

- ! Maintain a good documentation habit
- Maintain a more streamlined, standardized workflow
  - No standardized procedure from modifying code to building & executing to Juno. Should create a procedure first

Potential Extension Work

- Deeper look into other CSAL components
  - CoreSight Discovery Toolkits, for addresses and everything else
- Could write & run own application, but not enough time to get trace data for it
  - Deeper look into the Juno SoC TRM, looking into PC register values
- Deeper look into installing modules

Project Components

used throughout the development

Hardware & Connectivity

- ARM Juno r2
- Serial Wire
- USB-B to USB-A
- 8GB USB Stick
- Ethernet Cable
- Juno Power Supply & Cable

Libraries

- ARM Platforms Deliverables by Linaro
- Linux Kernel
  - BusyBox
  - OpenEmbedded
- CoreSight Access Library

Tools

- VirtualBox
  - Ubuntu 18 Client
- PuTTy
- Git
- Eclipse DS-5
- ARM Development Studio

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה