COMP40901

Summer Industrial Placement
Project Report 2022/2023

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

Author: Rakha Djokosoetono

Abstract

This report summarizes the mandatory project work undertaken throughout the writer's placement during summer of 2022, at Cambridge-based ARM Ltd within their Debugging Team. The project's deliverable enables support of on-chip trace generation on a specific AARCH64 Armv8-A development board by way of kernel and filesystem flashing through trial and error, and provided an internal knowledge-based article upon completion.

Acknowledgements

I am deeply grateful to my line manager, Stuart Shepherd, and my work buddy, Stuart Hirons, for their invaluable guidance and unwavering support during my summer placement. Their expertise and encouragement have been instrumental in shaping my growth and development within the team.

I also extend my heartfelt appreciation to the entire ARM Partner Enablement Group, whose knowledge and expertise have broadened my understanding of the industry and inspired me to pursue excellence in my field.

Lastly, I would like to express my heartfelt gratitude to my family in Jakarta, Indonesia, for their steadfast support and encouragement throughout my journey in pursuing education abroad. Their unwavering love and belief in me have been a constant source of inspiration and motivation, and I am forever grateful for their unconditional support.

Contents

List of Figures

Chapter 1

Introduction

1.1 ARM Ltd.

ARM Ltd. is a UK-based company founded in 1991 based in Cambridge, England. Its primary focus is the design of ARM family Processors, and the licensing of IP. As of 2016, ARM is owned by SoftBank Group, a Japanese conglomerate company.

1.2 Project Motivation

The ARM Debugging Team within their Partner Enablement Group has deemed it necessary to understand how to perform on-chip instruction tracing without the use of an external debugger probe. Theoretically, this could be achieved by the use of the **CoreSight Access Library**, an open-source API used for on-chip tracing that interacts with the CoreSight Debug and Trace components directly. However, the lack of clear documentation and ambiguity in architecture support for this tool has placed this objective in an indefinite backlog. Therefore, the objective at hand was to provide support for their tool, the CoreSight Access Library to perform debugging on one of their development boards, part of the Armv8-A architecture family.

Chapter 2

Design and Background Information

This chapter outlines the initially conjectured design of each component of the project. Each component that is not trivial will be discussed as background theory and internal tools and terms used during the undertaking of the project.

This section will briefly scratch the surface regarding the overarching machinations of each utilities in the section following, and describe how they (or their functionalities) relate to the main project at hand.

2.1 Debugging

Debugging in essence is the method of detecting and removing potential errors within in a product. Within the ARM debugging Team, there are two types of debugging, that is **invasive** and **non-invasive** debugging.

2.1.1 Invasive Debugging

Invasive debugging in hardware observes the system while it is running. Invasive debugging typically makes the use of an external debugger probe. An example of a debugger probe that is commonly used within the ARM debugging team is the DSTREAM-5 unit.

The debug probe directly interacts with the processor such that it has R/W (Read/Write) access to the debug registers. Debugger access to the registers are given by connecting the debug control subsystems to the functional interconnect, as shown below:

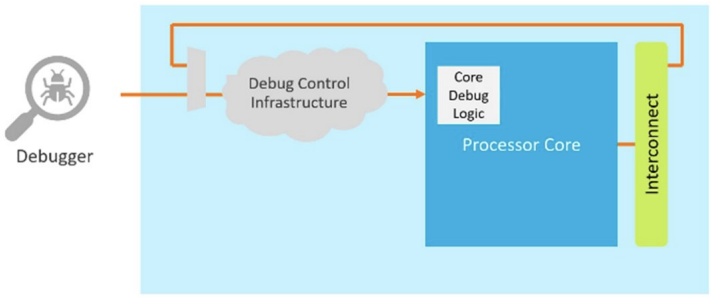

Figure 2.1: Invasive Debugging Illustration

Additionally, invasive debugging typically involves halting the cores to enter a 'debug state' midway throughout an operation, and exiting when finished. Invasive debugging will not be used in this project as trace generation (tracing) is part of non-invasive debugging, as described in the next section.

2.1.2 Non-Invasive Debugging

Contrary to invasive debugging, non-invasive debugging does not involve granting R/W access to registers to any external debugger. Non-invasive debugging focuses strictly on software behavior observation, not changing its behavior.

This is achieved by building program-tracking logic into the core of the processor that captures execution data. Data captured by the program-tracking logic may go to an external trace-port analyser or some internal logic for analysis. No direct connections to the functional interconnect is made, as shown in the diagram below:

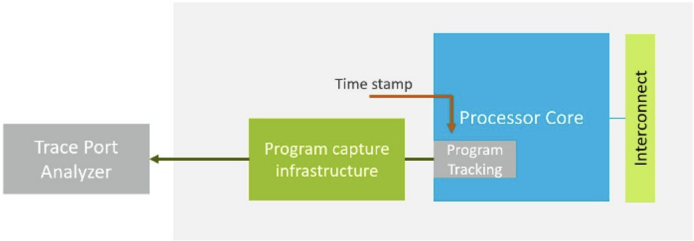

Figure 2.2: Non-Invasive Debugging Illustration

Tracing is part of non-invasive debugging as it only collects data on running processes, without any halting or changing any inherent behavior of the program. Tracing may also be used for performance measuring using a CoreSight PMU (Performance Measuring Unit).

2.2 Kernel

A kernel is part of the operating system. It resides between user processes and computer hardware, and is in charge of allocating hardware resources to accommodate all user processes. A kernel can have a variety of features or very little, depending on the usage intent of the device it lives in. Below is a visualization of where a kernel is in a system:

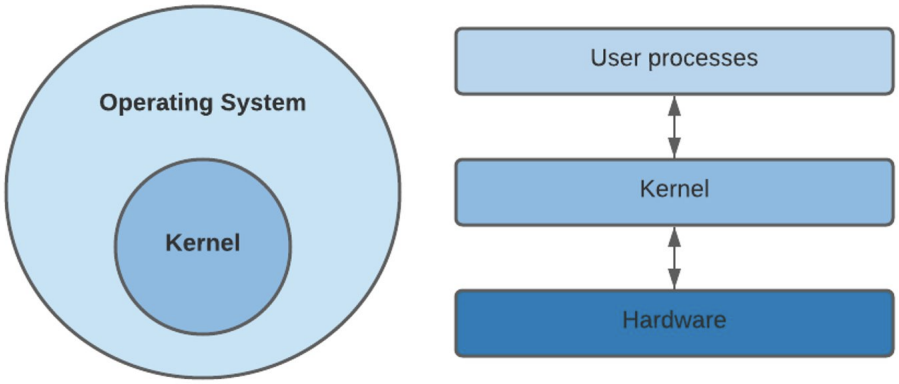

Figure 2.3: Kernel relationship between OS, User Processes and Hardware

2.2.1 Linux Kernel Building

In Linux, kernel building is driven primarily by a file named `.config`. This `.config` dictates the target device, the features it wants present or absent from the end kernel, and is supplied to a kernel tree. Below is what can commonly be found within a kernel tree directory, obtained from one of the repositories of `git.kernel.org`:

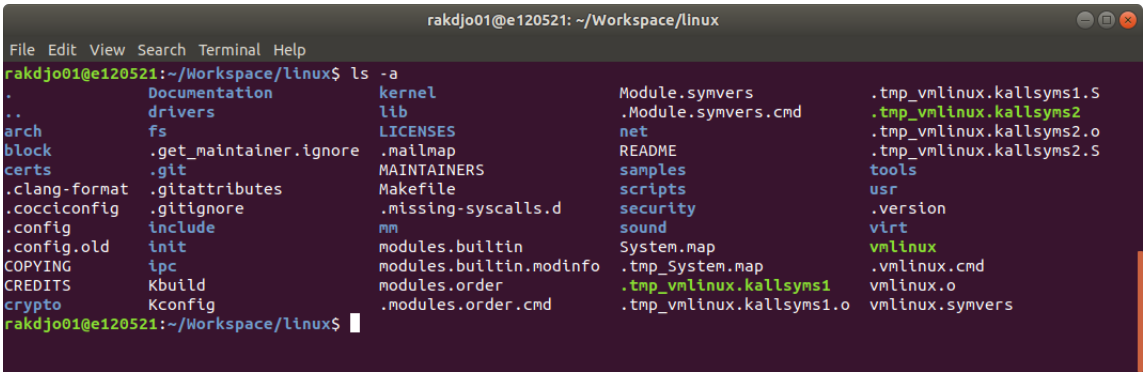

Figure 2.4: Kernel Tree Directory

Notice the `.config` file mentioned previously. If a `.config` file is not present, it can be built with the `make config` command at the root of the kernel tree. A new `.config` file can also be generated if one from an older kernel is present. In any case, setting the `CROSS_COMPILE` environment variable to equal a cross-compiler installation directory will target a device of a different architecture. Additionally, in order to build the kernel, the `make` command will need to be invoked at the kernel tree root directory.

This manual procedure however will be streamlined mostly with the use of the Linaro ARM Reference Platforms (see section 2.5). However, this method is revisited during the first implementation of the project (fast forward 3.2.1)

2.3 File System

The file system is part of the kernel. File systems describe how files are stored and is essentially part of the kernel that facilitates the interaction between storage media and other resources. The ARM Juno r2 development board (described in the next section) supports the BusyBox, OpenEmbedded, and Android file systems. Android was not explored during this project.

As this project deals with two different file systems of a Linux kernel, it is important to understand the underlying difference between the two, and how one successfully achieves the intended outcome of the project of on-chip trace generation, and the other failed to achieve it.

2.3.1 BusyBox

The BusyBox file system provides basic root file system support. It is designed for embedded systems with limited resources. In the context of this project, BusyBox only provides simple terminal commands, is missing a variety of programming tools, and it furthermore does not

support ethernet, USB booting.

In the next chapter, the BusyBox file system is explored and implemented, as its limitations procedurally hindered achieving the intended outcome of this project.

2.3.2 OpenEmbedded

The OpenEmbedded (henceforth OE) file system, when compared to the BusyBox file system, presents radically more features than BusyBox. Ultimately, this was the file system adopted by the kernel that helped achieve the project's intended outcome. A basic description of the features that are present in OE that helped achieve the project's goals are as follows:

- Git
- Internet through Ethernet
- Booting from USB
- Makefile support
- Python Interpreter
- GCC Compiler

The merits of these features are described further in 3.3.

2.4 Juno r2 Development Board

The Juno r2 Development Board is a 64-bit (i.e. AARCH64) software development platform. The architecture is based from the ARMv8-A family. Additionally, the 'r2' variant of the board (used in this project) hosts Dual core Cortex-A72 MPcore processors.

Out-of-the-box, storage is partitioned into two parts for the Juno board:

- The first partition is the main file system of the embedded Linux Kernel. This is accessed through a SerialPort connection plugged into the rear **UART0** port (see below)
- The second partition is for the board bootloader, to boot the kernel. This is accessed as a simple USB mounted storage to the computer through the **USB Configuration Port** (see below)

Below is a visual representation of the front and back sides of the Juno r2 Development Board:

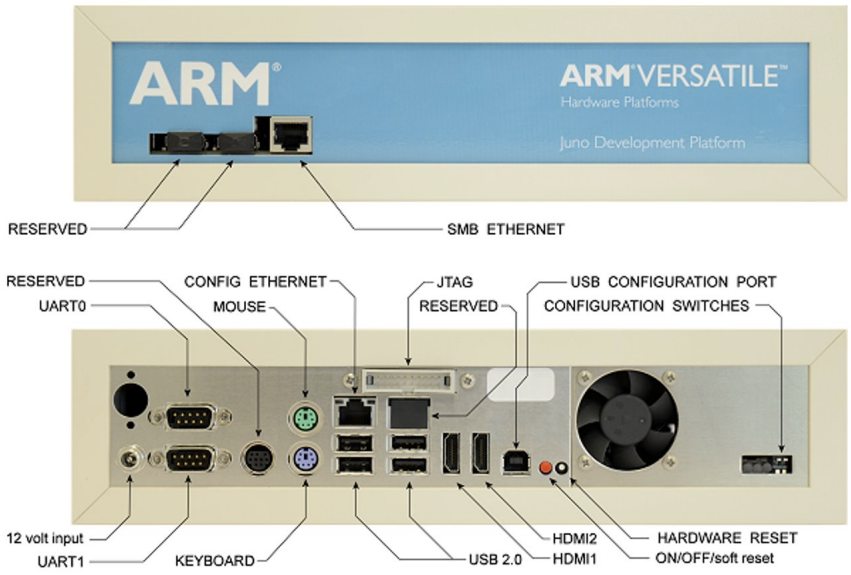

Figure 2.5: Front (up) and back (down) visualization of ARM Juno Boards

This project will in general only use the following ports of the ARM board from above:

- **SMB Ethernet** for internet connectivity
- **12 Volt Input** for power
- **UART0** for SerialPort Terminal Connection
- **USB 2.0 ports** for attempt at file transfer and booting from flash
- **USB Configuration Port** for mounted storage access, for kernel and firmware transfer

2.5 Linaro ARM Reference Platforms

The ARM Reference Platforms (henceforth ARP) is a library created by the Linaro Group, an organization that works on free and open-source technologies such as the Linux Kernel. Upon obtaining, these are the following files provided at first:

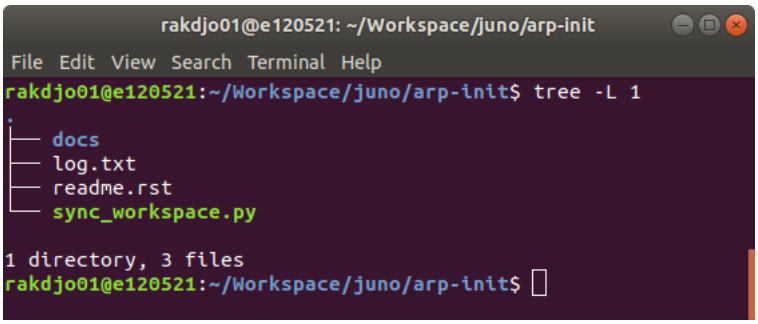

Figure 2.6: ARP initial directory

The workspace needs to be setup by running the provided `sync_workspace.py` script and installing the required dependencies as outlined by it if required. Once everything is setup, the directory should be populated similarly to below:

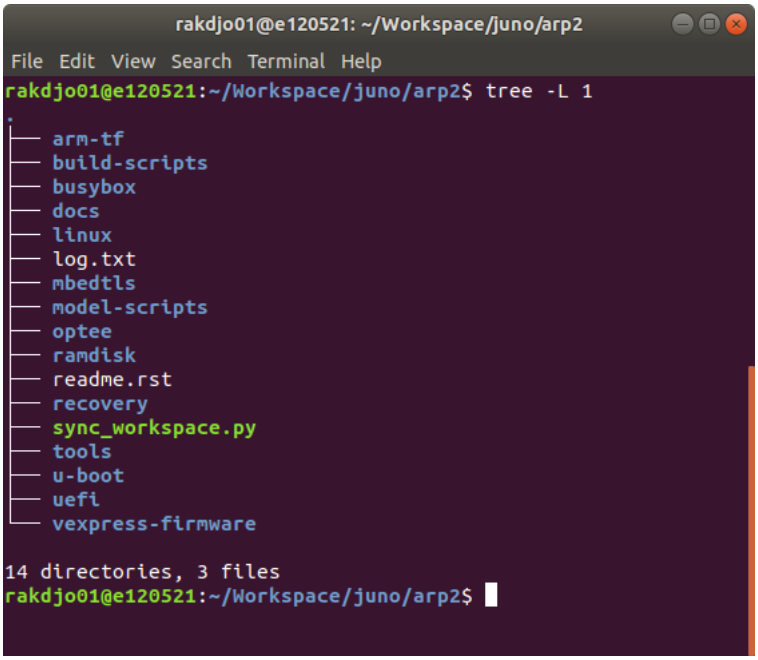

Figure 2.7: ARP populated directory after running the `python3` script

The main use of this library is to build software stacks for a variety of ARM platforms. In the context of this project, the ARP library will streamline the Kernel Build Process of the Juno Board. As outlined briefly in section 2.2.1 previously on the role of `.config` files in kernel building, the required `.config` files are provided in the directory after setup. Therefore, the only step that needs to be taken for kernel building would be only to run the necessary build scripts contained within the `build-scripts/` directory (see figure above).

The full ARP library can be obtained by cloning the git repository below through HTTPS `git.linaro.org/landing-teams/working/arm/arm-reference-platforms.git`.

2.6 ARM CoreSight

In essence, CoreSight is part of the SoC (System on Chip) infrastructure that provides support for debugging and trace capabilities. It is fully built in the SoC.

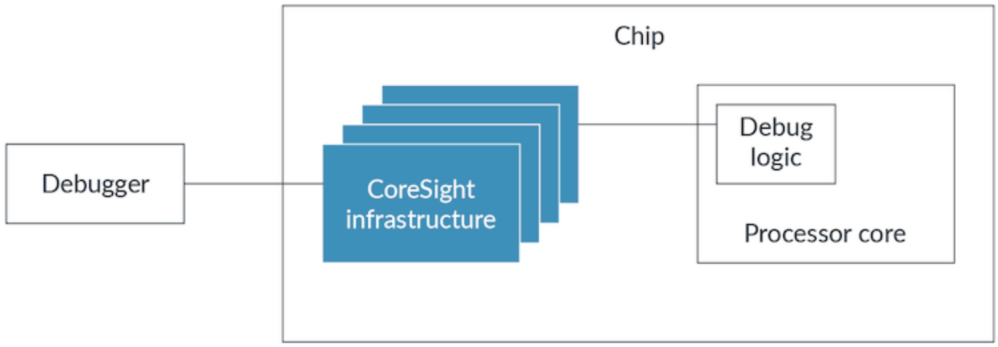

Figure 2.8: CoreSight Infrastructure in SoC

In terms of invasive and non-invasive debugging, CoreSight contains infrastructures for Program Capture (see fig. 2.4) and Debug Control (see fig. 2.1). Taking a deeper look into the infrastructure, ARM CoreSight has four types of components as illustrated by the different colors in the diagram below:

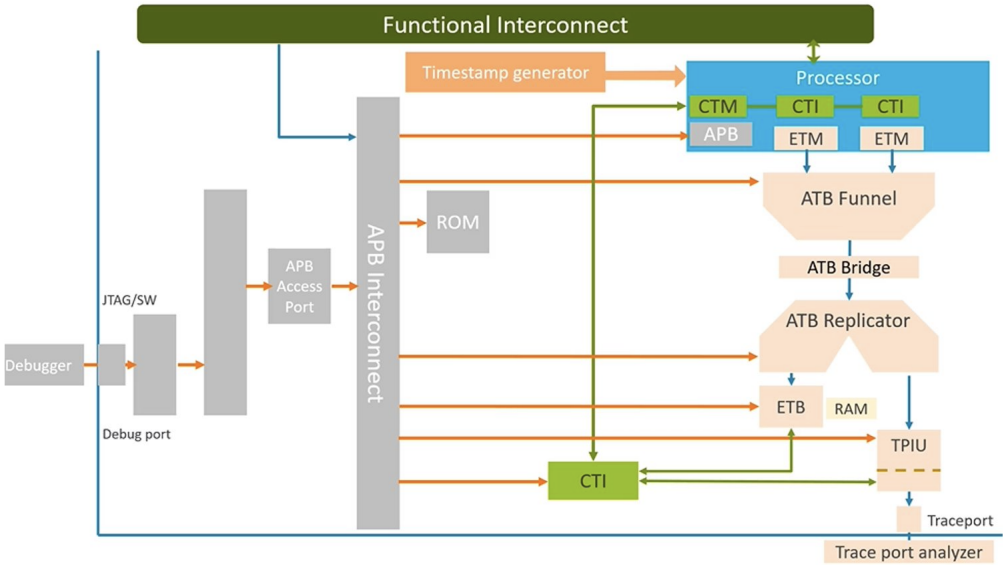

Figure 2.9: CoreSight Infrastructure - deeper look

Following are the descriptions of each type of component:

- **Debug Components**
  Debug components are represented in **grey**, they are components create the debug control path and act as the main point of entry to the system when using a debug probe. Conclusively, this component mainly supports invasive debug objectives.

- **Trace Components**
  Trace Components are represented in **cream** color. They support non-invasive debugging objectives. The CoreSight Access Library `tracedemo` program (see section 2.7) will interact with these trace components and allow trace generation.

- **Cross-Trigger Components**
  The cross-trigger interfaces are represented in green. Essentially, these components send signals throughout the system to halt and resume components. However, it can send other types of signals as well, such as notifying if something has happened during runtime.

- **Timestamp Component**
  The timestamp component represented in **orange** simply link events to a timestamp, so debuggers know when something happens.

2.7 CoreSight Access Library

The CoreSight Access Library (henceforth CSAL) is a library that allows user code to directly interact with the CoreSight debug and trace components hosted within a target device. CSAL formally supports earlier revisions of the ARM Juno Board (revisions r0 and r1 in particular) among other devices.

Most of the code contained within CSAL is written in C. This project will only use the subset of tools that is written in C of CSAL:

- **`csls`**
  `csls` simply lists the CoreSight components on the device by listing said components from the ROM table address. In the context of the Juno r2, it was discovered that the ROM table starting address is at `0x20000000` according to the Juno r2 Technical Reference Manual.

- **`tracedemo`**
  `tracedemo` is one of the programs within CSAL that captures trace data on a running Linux system. This project's end goal is to provide support for `tracedemo` to run on the Juno r2 development board.

- **`tracedemo_eti_stop`**
  `tracedemo_eti_stop` operates similarly like `tracedemo`. However, it injects a trigger-packet into each core's trace using the CTI channels and ultimately halts the capture by triggering a memory flush to the ETB (Embedded Trace Buffer).

- **`cs_etm_config_demo`**
  `cs_etm_config_demo` is a program that returns the configuration of the target device's ETM (Embedded Trace Macrocell) or PTM (Program Trace Macrocell) components . Like the `tracedemo_eti_stop` above, this program will not be used as knowing their configurations are irrelevant as we are only interested in generating trace.

In the context of this project, the CSAL **`tracedemo`** program will be used to generate instruction trace from user-defined applications from the Juno development board. Additionally, the basic **`csls`** program will be used to list all the CoreSight components available on the board.

In order to run the aforementioned four programs, building (compiling) them is essential beforehand using the provided `makefile`. This compilation process can either be done on directly on the target device, or through the means of a **Cross-Compiler** (see 2.8). The same `makefile` also has rules that discards any generated compiled files.

Furthermore, aside from the programs mentioned previous, CSAL also holds Discovery Toolkits written in Python that detects any ARM device's CoreSight topology, i.e. the ROM Table base addresses containing CoreSight components. Theoretically, these should ease the process of providing support for trace generation through the **`tracedemo`** program for formally unsupported devices. In the context of this project; the Juno r2, such was this project's objective.

The original CSAL open-source repository can be accessed at `github.com/ARM-software/CSAL`.

2.8 Cross-Compilation

Cross-compilation is to compile code for one computer system (called the target machine) from a different computer system (called the host machine) by the use of a cross compiler. An illustration of a cross-compiler workflow can be seen below:

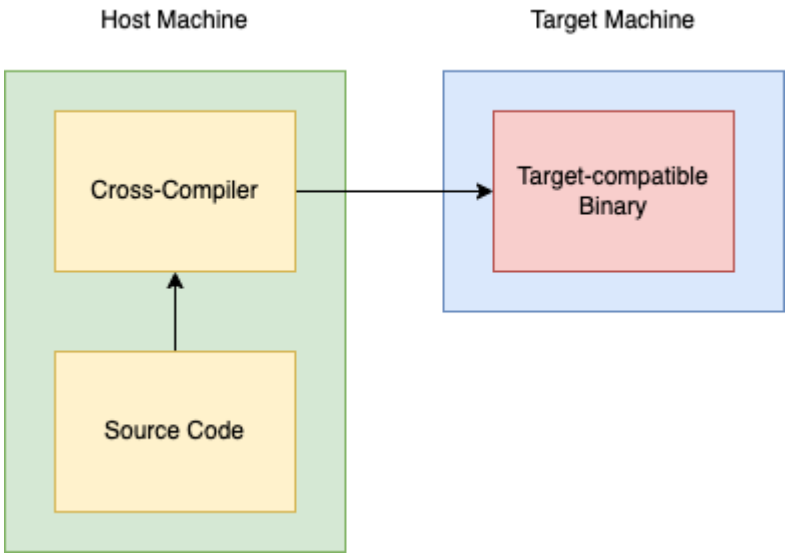

Figure 2.10: Cross-Compilation Workflow

Cross-compilation can be useful when the target device for example is too small to host a compiler, or even if a compiler is deliberately omitted. In the case of ARM, this allows ARM developers and engineers to compile software for ARM-architecture target devices on a machine that for example has an x86 processor.

In the case of this project, the programs in CSAL have the option to be cross-compiled (in addition to being compiled directly on the target device) to the target device as the host computer used in the undertaking of this project has an x86_64 processor. Cross-compilation is also used in this project when building a custom kernel. Additionally, the traditional way of kernel building described in section 2.2.1 makes use of a cross compiler as the target device (Juno development board) is ARM-based.

2.9 ARM Development Studio

ARM Development Studio (ArmDS henceforth) is an Eclipse-based C/C++ Integrated Development Environment (IDE) designed for the development of ARM-based SoCs. Though its features are plentiful, this project will only focus on its debugging capabilities, to be precise its snapshot viewer feature to read application instruction trace generated by the `tracedemo` program of the CSAL.

Arm Development Studio, which was only relatively recently released, is the successor of the Eclipse for DS-5 IDE. A lot of the documentation may reference DS-5 instead of ArmDS, however they are essentially the same.

2.10 Initial Design Verdict

The initially intended outcome was to achieve trace generation with the Linux kernel having a BusyBox file system, though the next chapter details the outcome of the implementation attempt.

Chapter 3

Implementation

This chapter mainly discusses the methodologies and steps taken to implement the initially conjectured design to generate trace on-chip, and the failure of said design. This is then followed by an alternative implementation approach to rectify the initial design's failures, and the resulting deliverable from the successful implementation.

3.1 Planning and Work Management

To keep good track of the project's progress in providing support, Confluence by Atlassian Corp. was used at ARM as an internal documentation tool for Applications Engineers. Furthermore, the final project documentation is hosted there, where ARM engineers have access.

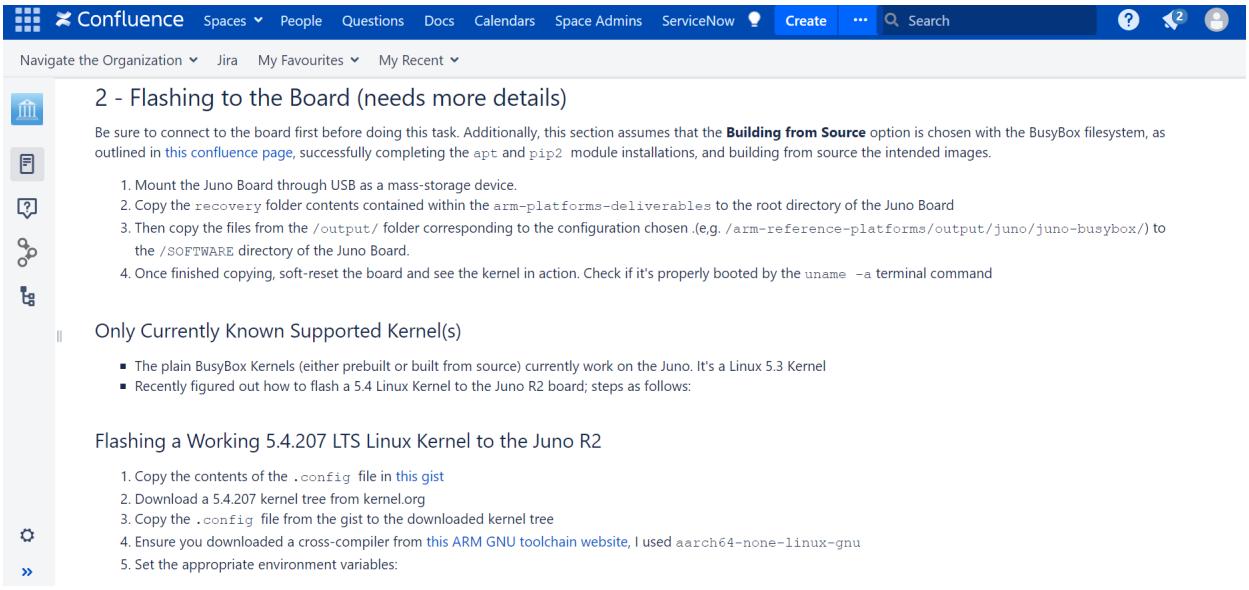

Figure 3.1: Confluence Page Sample

3.2 Initial Design Implementation

As outlined in the previous chapter, the initial design had a general goal of generating trace on the Juno r2 development board that has the BusyBox Kernel. Following were the steps taken to achieve the goal using the initial design:

1. **Flash a Working Kernel**

   Flashing a working Linux kernel is key to proceed as it allows interaction of the host computer with the board by means of a serial terminal. The ARP library will be used to create the kernel that will be flashed on to the Juno Board's mounted storage. The following settings were selected upon initialization of the application:

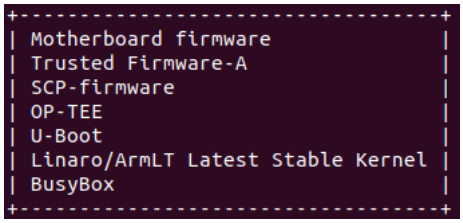

Figure 3.2: ARP Library initialization settings

   Once the build command is issued, the kernel deliverables should now be copied into the Juno r2 mounted storage. If the kernel flashes correctly, then the following should be seen over the serial terminal, where keyboard input can be read (ignore errors and warnings):

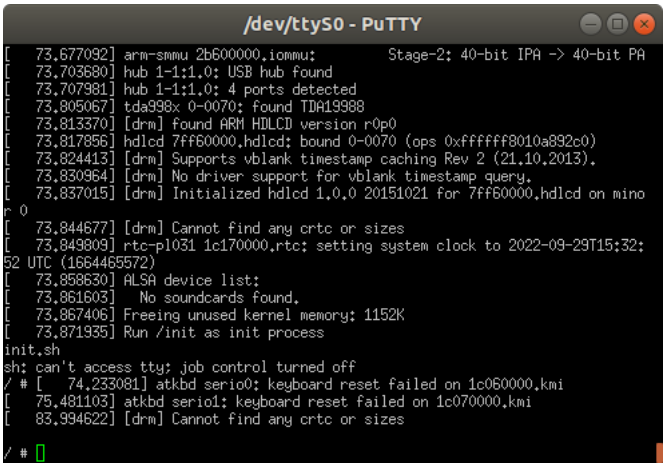

Figure 3.3: Successful Kernel Flashing with BusyBox filesystem

   Additionally, this means the Juno Board is finally running a Linux Kernel with the BusyBox filesystem. In the context of this project, the Kernel was running Linux 5.3, as shown in the figure below, with the built-in commands:

2. **Transfer CSAL to the Board**

   There are three ways of moving the CSAL source code to the board, as outlined below:

Figure 3.4: Linux 5.3 Kernel with built-in BusyBox commands

(a) **USB Thumb Drive**

The Juno Board has four USB ports at the rear panel. Therefore, the obvious first attempt to transfer the CSAL source code to the board was by means of a USB Drive. The source code was copied to the root of a USB drive, then inserted into one of the USB ports while the Juno Board was on.

However, it was found that the kernel prevented the device from properly mounting. This is because although their block information and device id can be detected by the kernel in `/dev/sdX` and `lsusb` command respectively, as shown below in the following figure, this BusyBox kernel prevented access to that aforementioned `/dev/sdX` directory by not including it in the file system to begin with.

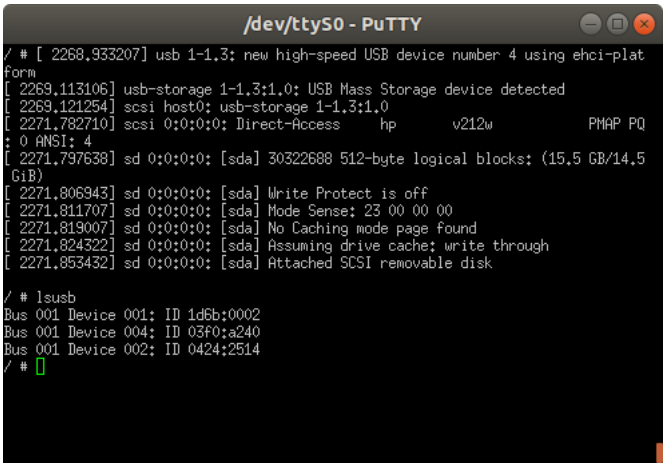

Figure 3.5: USB detected

In the figure above, `sdX` was `sda`.

Another method that could potentially support the transfer of files was by means of an ethernet cable.

(b) **FTP through Ethernet**

In addition to USB ports, the Juno Board also has multiple ethernet ports. The ethernet cable was inserted to the front port, as advised by the TRM (Technical Reference Manual), but an IP address was never assigned. It was discovered that in addition to not providing support for USB mounting, BusyBox also does not support networking out-of-the-box. This means an alternative method must be found to transfer the CSAL code to the board, for the terminal to access.

The final approach involved the use of a mounted SD Card.

(c) **Mounted Micro-SD Card**

A Micro-SD Card slot was also provided by the ARM Juno Board for additional storage space. Like the process with the USB Drive, the source code was copied into the root directory of the SD card, and then inserted into the provided slot.

Fortunately, BusyBox supports SD-card mounting. Its block information can be accessed at the point `/dev/mmcblk0p1`. This point can be mounted by means of the Linux `mount` command, and its contents can therefore be accessed. Additionally, the `umount` command had to also be used to properly unmount the SD-card.

As the BusyBox kernel only supports the transfer of data through a Micro-SD card, this will be the standard used when moving the source code (CSAL included) from the host computer to the Juno board.

3. **Building and Running CSAL Programs**

The final step in generating trace would finally be to run the CSAL programs. As mentioned in section 2.7, CSAL can either be built on the device, or cross-compiled to the appropriate target. Both methods make use of the `make` command with the provided `makefile` in the CSAL root directory.

(a) **Building On-Target**

Following the steps outlined in the documentation to build on-chip, the only command that was needed is the `make` command, to be invoked at the serial terminal at the root of the CSAL directory of the board. However, it was soon found that BusyBox did not provide support for running `makefile`.

Additionally, it was also found that GCC was not present in the kernel. Python was also not supported, as shown in the figure below.

Therefore a cross-compiler had to be utilized to run the appropriate CSAL programs on the board, to attempt to generate trace data, as building on the board turned to be an impossible endeavour.

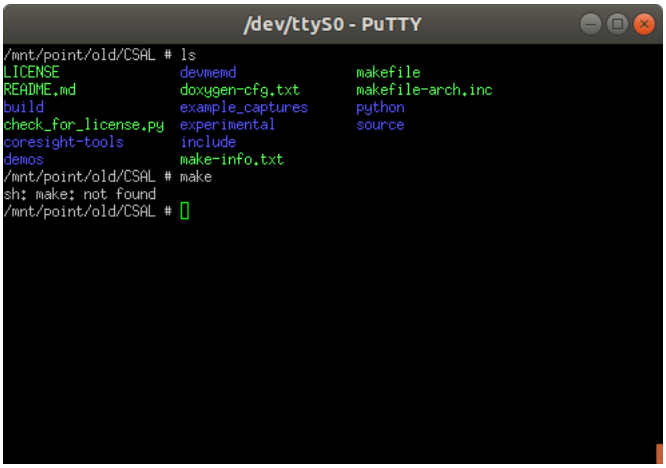

Figure 3.6: BusyBox failed to run `makefile`

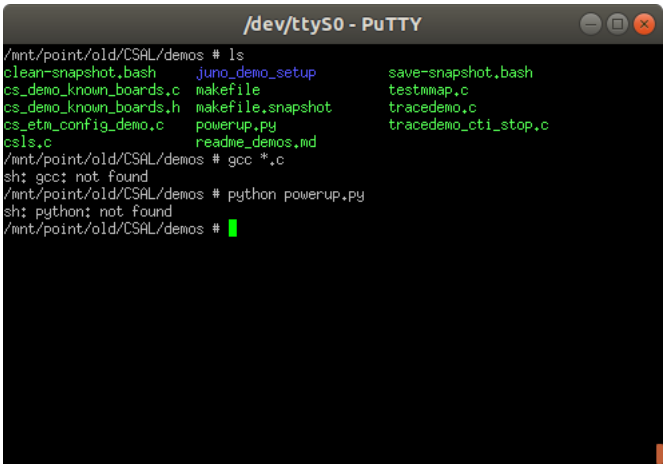

Figure 3.7: GCC and Python not present in the deployment

(b) **Cross-Compilation**

Cross-compiling CSAL to the Juno r2 development board requires the installation of a certain GCC cross-compiler toolchain. The `aarch64-none-linux-gnu` cross-compiler was selected and installed to the host PC, as the Juno r2 was an AARCH64 Armv8-a architecture running a Linux Kernel.

Additionally, the local terminal variable `CROSS_COMPILE` had to be set equal to the installation directory of the aforementioned cross-compiler, before invoking the `make` command at CSAL folder root.

After invoking `make` on the host computer, the resulting compiled binary files were then placed in a new directory of `./bin/arm64/rel/` of the CSAL directory. Figure 3.8 shows the list of binary files present in the directory, imported to the Juno Board through Micro-SD card, to be run from the terminal.

However, the board still could not run the resulting binary files generated by the cross-compiler, as shown in the figure below.

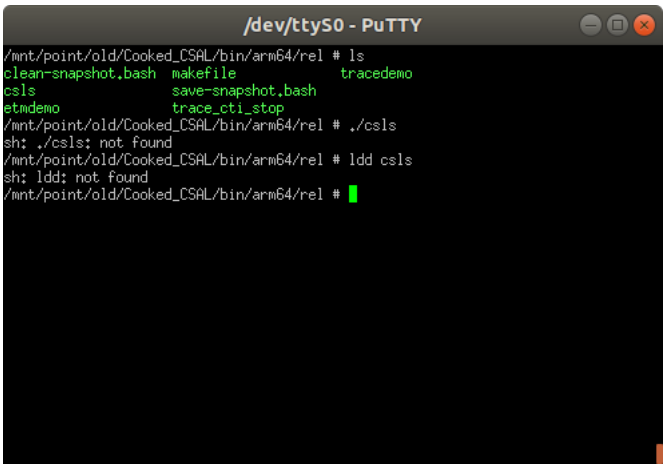

Figure 3.8: Files list in `./bin/arm64/rel/` and failure of running binary files

An attempt at using the `ldd` command to list any dependencies of running binary files was met with failure, as shown above.

The same procedure of cross-compilation and transferring binaries was applied to a simple "Hello World" program written in C, but the same failure was met:

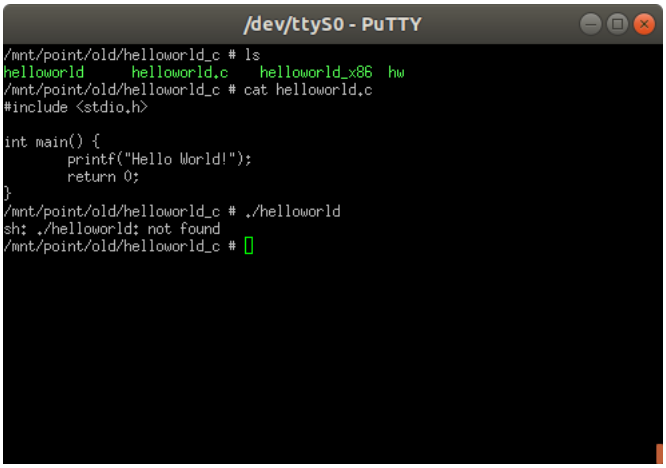

Figure 3.9: Fail to run cross-compiled Hello-World program

There should be a way of installing more tools to the library to run the cross-compiled programs. However, there was little to no guidance found to achieve such a task without the `apt` package manager, which was also not present in the kernel.

Furthermore, the kernel did not support ethernet connectivity. Evidently, this greatly inconvenienced the potential installation of other modules integral to trace generation.

In retrospect, the BusyBox Kernel prevented the project from moving forward, from running the CSAL programs to generate trace data. It was seen as well that running the simplest C compiled binaries were near impossible. Importing a variety of utilities through the Micro-SD card could potentially nudge the project forward.

3.2.1 Potential Kernel Issue

Given the failure and challenges met thus far, a possible issue could be the Linux Kernel version itself, in the way that it may not have the required features to execute the binaries or compile programs. A later kernel version was rebuilt using the `.config` file of the previous 5.3 version flashed into the system, in hopes of potentially rectifying the missing library issue.

Rebuilding the kernel involved cloning a kernel tree of a later Linux Kernel. In this case, version 5.4.207 was used as a trial installation run. By setting the `CROSS_COMPILE` local variable to the cross-compiler install directory and building the kernel with the supplied configuration file and adding all new features accordingly, a newer Linux kernel was successfully flashed, albeit the built-in commands were the same as the previously flashed version:

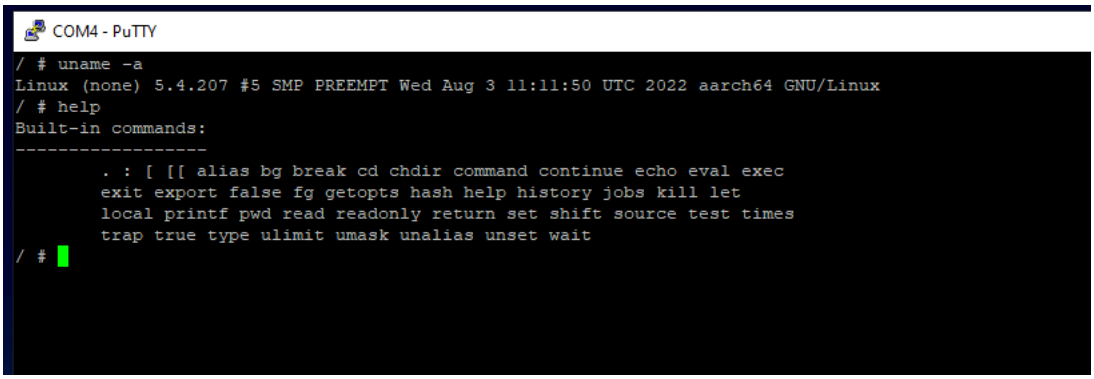

Figure 3.10: Successful 5.4.207 Linux Kernel flash

Unfortunately, all of the same issues mentioned previous were met again. A higher Linux kernel version of 5.15.59 was also flashed to the board, only again to be met with identical issues. Conclusively, for now, it turns out that it is not a kernel version issue. Furthermore, it was then at this point in the project, the decision of abandoning the BusyBox filesystem was made, to revise the approach. These revisions are discussed in the section following.

3.3 Revised Design Implementation

This revised design came after the realization that BusyBox presented a lot of limitations, a notion unbeknownst to us before the undertaking of the project. Its lack of features, was a source of a handful of the issues throughout the first attempt at implementation. Therefore, an alternative Linux file system was chosen for the board kernel. The OpenEmbedded file system was chosen as the Juno board also supports the file system.

A great deal of the steps hold the same purpose and remain the same as seen above. However, this revised design implementation contains a few changes to the indicated steps:

1. **Flash a Working Kernel**

   The ARP library also supports building Linux Kernels with the OpenEmbedded (OE) file system. However, this required a complete rebuild of the ARP library. The following settings indicated in the figure below were selected to reinitialize the library, to build Linux kernels for the Juno r2 with the OpenEmbedded filesystem:

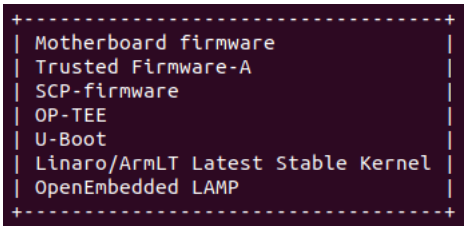

Figure 3.11: ARP Library initialization settings for OE

   Upon initialization finish, below is the resulting directory for the new ARP:

Figure 3.12: ARP directory with OE initialization

   The kernel must be built with the `CONFIG_STRICT_DEVMEM` flag set to `n` in order to access the kernel memory dump for instruction trace. This can be done by modifying `distribution.conf` located within `./linux/linaro/configs/`.

   After the kernel is built, move again the deliverables to the mounted Juno storage, just like the initial design.

- **Format a USB Stick**

  This implementation of the OE filesystem boots from a USB stick. However, the USB filesystem needs to support R/W support for the kernel.

  The **`ext4`** journaling filesystem will be used as the USB stick filesystem. This is because `ext4` is the latest in the series of journaling filesystems for Linux. Alternative filesystems such as NTFS are more suitable for internal hard drives, hence they will not be used in this project.

- **Burn the `.img` to the USB Stick**

  The provided `.img` file mentioned earlier must be burned into the USB stick. This will create two partitions, `boot` and `root`, which is essential for the booting process of the board (boot) and file storage (root).

  The command used for burning the image is `dd`.

After burning the image to the USB stick, the stick should be inserted in to one of the rear USB ports to boot it from there. In the figure below, it can be seen that the Kernel is waiting for a USB stick to be inserted in order to boot:

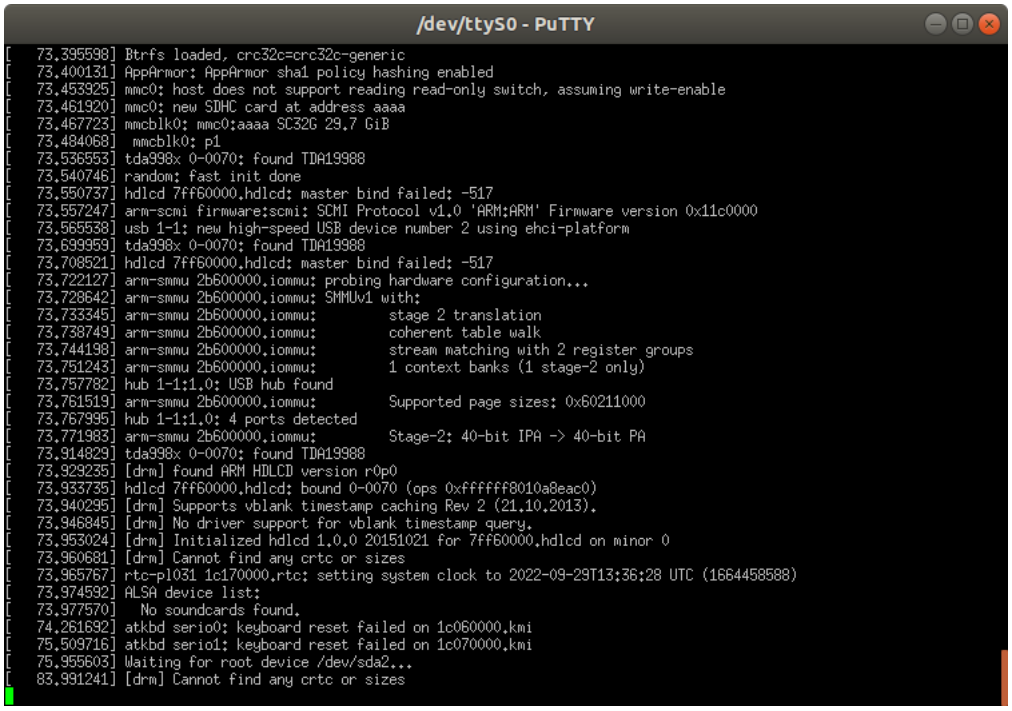

Figure 3.13: Kernel waits for USB for Boot

Once the kernel boots, a big difference that can be spotted is that root access is now granted upon login, unlike the previous BusyBox implementation that lacks support for the `sudo` keyword.

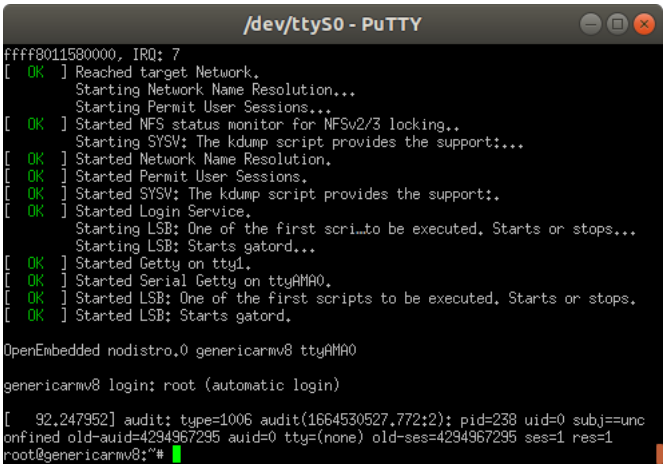

Figure 3.14: OpenEmbedded ready to accept input

2. **Transfer CSAL to the Board**

   Transferring files using a Micro-SD card is still supported with the OE filesystem. However, the OE filesystem supports the use of the ethernet port. Additionally, the `git` command is available for use on the serial terminal.

   Therefore, any changes to any user-code (including CSAL) is done from a code editor of the Host Computer. These changes are then pushed to a `GitHub` repository, and pulled into the Juno Development Board, where they can be built or run.

3. **Prevent CPUs from turning Idle**

   Linux adopts an exemplary power-saving principle. When cores or peripherals are not going to be used (or have not been used for a while), Linux will switch such peripherals or cores off. However, from a debugger perspective, CPUs shutting down breaks the debug and trace connection. Additionally, this ensures that all resources were available when the CSAL programs were run.

   To prevent the CPUs from shutting down, the script `no_cpu_idle.sh` is run at every startup, before performing anything else. This script is provided in the CSAL repository's demo directory.

4. **Building and Running CSAL Programs**

   Less errors were met in this approach compared to the last one. Like previously, CSAL can be built either on the target device or using a cross-compiler. Following are the outcomes in doing so.

   - **Building On-Target**

     Unlike the previous BusyBox Kernel, the OE version has a built-in `gcc` compiler feature and support for running `makefiles`. This means the `makefile` provided at the root of the CSAL directory can now be used to build (compile) the CSAL

programs. Below can be seen the terminal after running the `make` command:

Figure 3.15: Successful termination after `make` command

In the `./bin/arm64/rel/` directory, the files are shown below:

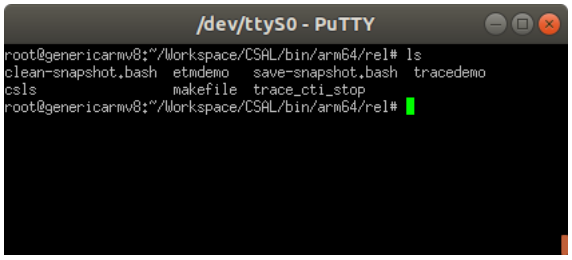

Figure 3.16: `./bin/arm64/rel` directory after running `make`

The next step would then be to run the compiled programs, namely `tracedemo` by issuing the `./tracedemo` command where it is located:

Figure 3.17: new `.bin` and `.ini` files generated by `./tracedemo`

- **Cross-Compilation**

Despite the success of building the code directly on the target after, running the cross-compiled programs was attempted again out of sheer curiosity. However, the same library `lib6` was missing and prevented the compiled binary to be properly executed. This is shown when listing dependencies of a certain binary file using the `ldd` command, as shown in the figure below:

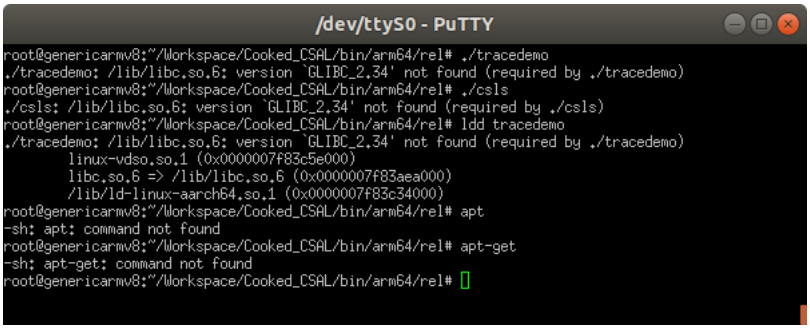

Figure 3.18: Failure to run Cross-Compiled binaries

In the figure above, `./tracedemo` and `./csls` attempted to run, however, some libraries that were required were missing. The `apt` and `apt-get` package managers were also not present in this deployment.

Theoreticaly, the `lib6` library could be installed more conveniently in the OE kernel rather than BusyBox as the board may now access the internet through the ethernet cable. However, the `apt` package manager was still missing in this deployment.

Instead of attempting to provide support for running the cross-compiled binaries, CSAL will be built on the Juno target device as the workflow will be faster whenever binary files need to be discarded and source code needs to be recompiled.

The final workflow looked like this:

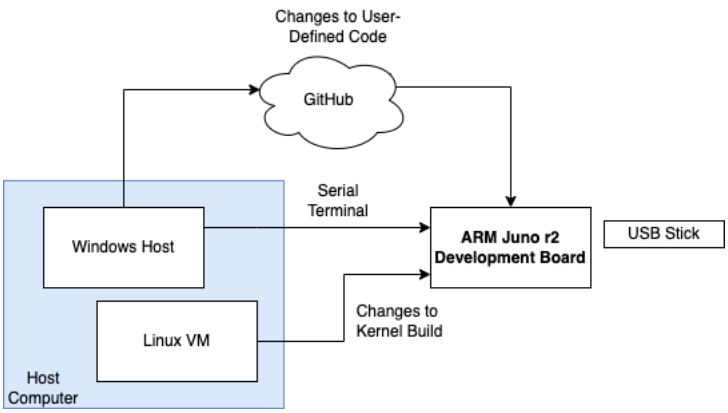

Figure 3.19: Final Workflow

3.3.1 Understanding the newly generated files

A variety of `.ini` and `.dump` files are generated when the `tracedemo` program is executed, as shown in figure 3.17. This is the trace data that has been successfully generated. What follows is a brief explanation of what the important ones are:

- **`snapshot.ini`**
  The `snapshot.ini` file provides a 'snapshot' (as the name suggests) of the kernel runtime. This and all `.ini` files will be imported to the ArmDS snapshot viewer for trace analysis.

- **`kernel_dump.bin`** The `kernel_dump.bin` file holds the kernel dump memory during the trace. This file is required for the ArmDS debugger to read the snapshot data.

Details of the other files are not relevant for the next step and chapter, that is importing the data into the ArmDS debugger.

Chapter 4

Results & Evaluation

This chapter follows after running `./tracedemo` with success, i.e. the `./bin/arm64/rel` directory is populated as in figure 3.17. This chapter will start by importing the new files to ArmDS and understanding how the trace works.

4.1 Importing Trace Results

The text output after running `./tracedemo` ends with 256 bytes of trace as can be seen as:

```
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...
```

Figure 4.1: Last lines of `./tracedemo` text output

These addresses are not meant for users to manually trace, but they are used to reconstruct the code path in the ArmDS snapshot viewer.

4.1.1 Trace Analysis

The following perspective can be seen after importing the `.ini` and `kernel_dump.bin` file appropriately to the snapshot viewer, in addition to importing a `vmlinux` file generated by ARP after building the OE kernel.

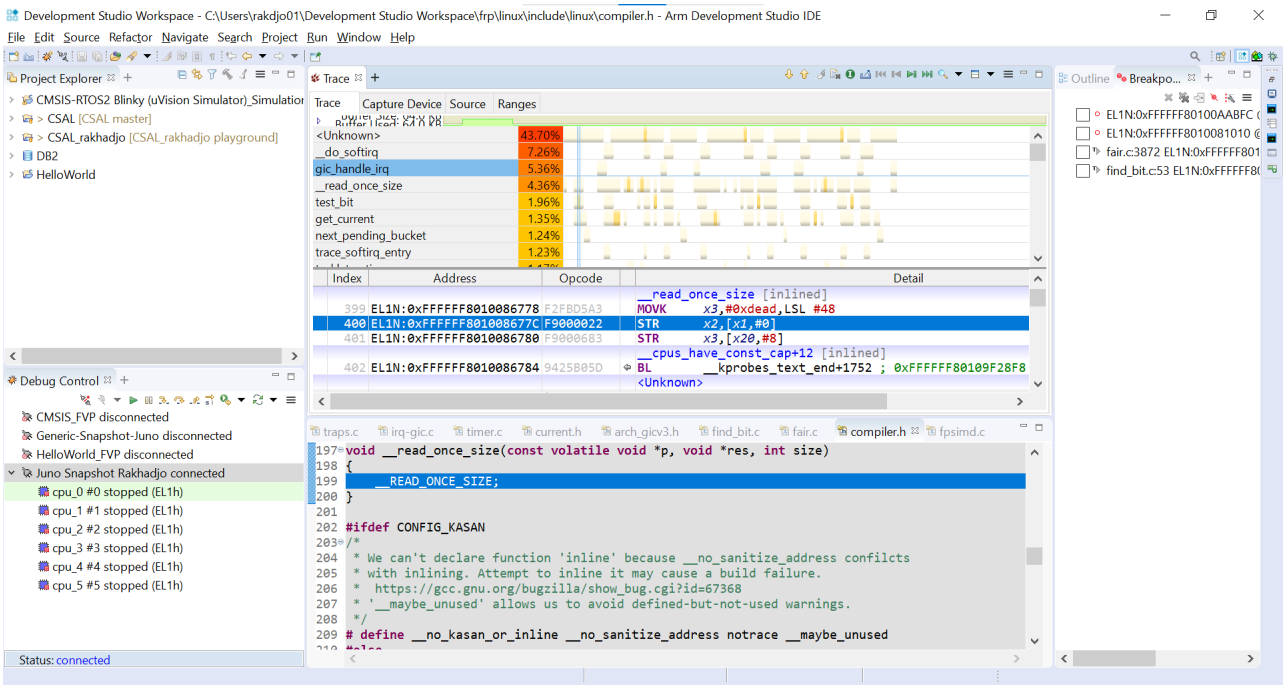

Figure 4.2: Tracing Perspective in ArmDS

There are three important windows in the figure above:

- **Heat Map**
  The heat map (located in the **trace** window in the center top) shows how much of the CPU process is given to each of the processes. This means that on the Juno board, the `_do_softirq` task as above consumed roughly 7% of the processing power.

  The heat map can be navigated by pressing the left and right arrows on the keyboard. Presently, the blue line represents where the viewer is currently at. Changes to the view by the keyboard arrows are reflected on the following Disassembly and Source Code windows.

- **Disassembly**
  The disassembly perspective located just under the heat map shows the assembly level code corresponding to where the blue line (current view) is at the heat map right above it. This low-level code is derived from the original position in the source code, shown just below it, with the method names stated too.

- **Source Code**
  The source code view shows the high-level code corresponding to the low-level code just above it. This view is achieved by importing the `vmlinux` file generated by ARP during the kernel build process.

4.2 Potential Security Issues

After utilizing a variety of tools throughout the undertaking of the project, there are a few security issues and risks that may be the cause for concern.

Firstly, the CSAL and ARP tools by ARM and Linaro respectively are available as open source. This means the general public has access to the code, and its inner machinations. No authentication was also requested when the tools were used, i.e. when performing kernel building or flashing by the ARP, and when performing tracing using CSAL. In essence, anyone can potentially access the CoreSight system, and/or build and flash kernels maliciously. Attackers could access large Armv8-A based data centers and run the same CSAL as I have done to steal some processor trace data.

Some form of authentication should be put in place whenever using low-level tools such as ARP or CSAL to prevent privilege escalation attacks on any CoreSight system. An improved kernel can be built that supports authentication, as the OE kernel used in the successful implementation of this project automatically granted root access to whoever could access the terminal. Additionally, some changes to CSAL can be made by only limiting its access to the kernel, instead of everything. All in all, a good (human) security practice should be conducted to prevent potential attackers from gaining physical access to an ARM system containing sensitive data.

4.3 Company Benefit: The Raspberry Pi Shortage

It was revealed that after the project was completed, there has been a Raspberry Pi shortage due to supply-chain issues this year. Raspberry Pis were commonly used within ARM to perform ARM-on-ARM debugging, in which this report is not at liberty to go into more details of. However, there were a handful of used Juno development boards that remained idle. The kernel build process done in this report can be replicated by ARM debuggers and perform the needed debugging that would have usually been done by Raspberry Pis.

Chapter 5

Reflection & Conclusion

This chapter focuses essentially on how this project could have been done differently, and any potential extension work that could be done. Furthermore, personal reflection on the overall learning achieved at ARM will be discussed here.

This chapter is written with a first person perspective to reflect personal learning.

5.1 Personal Learning

Before being placed in the debugging team, I had little to zero knowledge regarding Linux kernels, embedded systems, and the likes. The project that I was assigned to pushed me to learn at an exponential rate by getting my hands dirty, by means of kernel building and flashing.

As I was the only member of the team that was placed in Manchester (the rest were in Cambridge), I learned a lot about keeping myself accountable for the work that had to be done, that meant being in charge of my personal learning and not relying on a lot on others.

5.2 Main Challenges

One of the biggest challenges of undertaking this project was the relatively messy, outdated, contradicting and sometimes ambiguous documentation that was available to drive this project forward. This meant that almost everything had to go through a trial-and-error process, such was the case when BusyBox was first used before making the move to OE, as we simply did not know its limitations even when implementing. To prevent this from happening in the future, I have posted my overall documentation on a Confluence page so it can be referenced again in the future.

It was mentioned as well that the Juno r2 was formally unsupported by CSAL. Everything worked fine in general. However there were a few commands that had to manually receive function arguments (e.g. address values) to properly function. A pull (change) request will be made to indicate the board was in fact fully supported to the original repository.

5.3 Things I Would Do Differently

Although everything ended up in an internal documentation site, consistency was not my strong point in a lot of aspects of this project. Oftentimes I would forget to update the documentation. This resulted in potentially losing important pieces of information throughout the trial-and-error stage of the project.

There was very little compliance from my end in the workflow I had set up for myself when (for example) code had to be modified and sent to the Juno board. This resulted in a messy git workflow in the forked repository, and multiple branches (and even repository clones) had to be made in order to continue working.

5.4 Potential Extension Work

I believe I did not have as much time looking into greater detail about the other features provided by CSAL. I did not at all touch the python discovery kits.

This project ultimately hasn't succeeded in obtaining application trace outside the kernel. However, if I were to continue on with this task at hand, I would look into modifying the kernel memory address trace size within the CSAL program, to gain access to more than just the default address space.

Aside from this, I could have potentially invested in more time to look into installing modules starting from `apt` into the Juno board, just to see interesting trace data by third-party applications.

5.5 Conclusion

In conclusion, this project was extremely well-received by ARM Applications Engineers. Despite the shortcomings (if any) on this project, learning and understanding what worked and what didn't is still valuable information granted if it were saved, so that the same mistakes aren't made in the future. On a personal note, I thoroughly enjoyed the undertaking of this project am lucky to have gauged an interest in embedded Linux.

Bibliography

[1] ARM. Arm development studio getting started guide. *ARM Developer*.

[2] ARM. Arm versatile express juno development platform (v2m-juno) technical reference manual. *ARM Developer*.

[3] ARM. Coresight access library. *GitHub*.

[4] ARM. Introducing coresight debug and trace. *ARM Developer*.

[5] ARM. Juno arm development platform getting started guide. *ARM Developer*.

[6] ARM. Understanding trace. *ARM Developer*.

[7] BusyBox. Busybox: The swiss army knife of embedded linux. *BusyBox*.

[8] Linaro. Arm reference platforms. *Linaro GIT*.

[9] OpenEmbedded. Openembedded wiki. *OpenEmbedded*.

[10] Liam Tung. Raspberry pi: Why they are so hard to buy right now, and what you can do about it. *ZDNET*.