DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Report

Praktikum L4 Microkernel - Porting Genode/Fiasco.OC to Odroid-C2

Authors: Tabea Haeseler, Lukas Graber, Johannes Walcher
Submission Date: 2020-08-07

Contents

1 Introduction

The practical course "L4-Microkernels" introduced basic tasks of modern operating systems, focusing on concepts applied in L4-based microkernels. The practical part of the course was the task of porting a L4-based operating system to an embedded hardware platform, in our case the Odroid-C2 from Hardkernel. Exercises were provided to give the teams a good starting point into the different platforms and tools required for the actual porting task. The overall practical phase can be divided into three different parts: In the beginning, there was a dedicated time slot for the participants to get acquainted with the hardware platform. The second part was about porting the Fiasco.OC kernel together with the L4Re or Genode OS Framework to the Odroid-C2. The last part focused on application development on the Genode OS framework.

1.1 L4 Microkernel

The microkernel concept is about eliminating logic from the operating system kernel, that does not absolutely need to run in a privileged mode. This especially means that driver logic is not build inside the kernel. but rather in userland.

This separation is implemented by abstracting into inter process communication, such as that an IRQ which is handled in-kernel for monolithic kernels is sent as an IPC callback previously registered to the interrupt. That way a userspace driver does not need to be run within the kernel.

This enables microkernels to minimize the trusted code base, for any userland process. The trusted code base is the code, that a userland process has to trust in order to be confident that it is running as desired.

1.2 Genode Framework

The Genode framework is especially focused on minimizing the trusted code base by minimizing every component within the critical chain to the userland process. This is achieved by managing process permissions using capabilities. Capabilities are used to describe hardware access, such as memory and CPU time as well as inter process communication to call other process' service.

2 Hardware overview

Our Hardware platform was the Odroid-C2, developed by Hardkernel in 2015. It is based on an Amlogic S905 running an ARM Cortex-A53 processor quad core accompanied by two gigabytes of DDR3 RAM and a triple core Mail-450 GPU. The processor is running an 64 bit arm-v8a instruction set. The board has several build in peripherals ranging from a gigabit ethernet port, four USB ports, and an infrared sender and receiver.

3 Division of tasks

For a large part of the practical course, every team member was more or less equally involved in the different sub-tasks. Only as the deadline was approaching and the project could be differentiated into the actual porting task and the *Genode* application development, it made sense to focus on different aspects and split up the work. Lukas Graber was responsible to ensure that the documentation in our *GitLab* Wiki was up-to-date and in as much detail as possible. Furthermore, he was involved in setting up U-Boot and experimenting with different versions of U-Boot for the Odroid-C2. The actual porting task was a shared effort together with Johannes Walcher. Johannes was also responsible for setting up remote access to the Odroid-C2 as well as porting the Genode OS Framework on seL4 as well as the Fiasco.OC kernel. As Tabea Haeseler did not have access to the Odroid-C2 until the remote setup was built, she was focusing on the Genode application development. Furthermore, she documented her experience with running Linux subsystem for Windows.

4 Collaboration Platform

The special situation in the summer semester 2020 due to the lockdown required us to avoid gatherings that can be avoided, such as team meetings In order to be able to work as if we were together we createed a remote collaboration platform using a spare laptop, a wifi-connected plug, a FTDI RS232 adapter and the Odroid-C2.

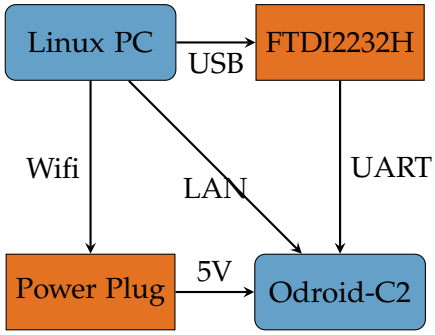


Figure 1: Hardwaresetup for remote access to the Odroid-C2

The general hardware setup is described in figure 1.

The Linux PC is connected via USB to a FTDI adapter for accessing the TTY of the odroid. The power to the odroid is controlled using a wifi power switch DeLock wifi power switch[1] Local network is connected with a router inbetween the router and the Linux PC. This router is configured to give the odroid a constant ip address.

4.1 Linux Machine

The Linux Machine is running Archlinux. For the course a systemd-nspawn container[NSPAWN20] with a Ubuntu 18.04 was installed. This container can be accessed by the team using SSH in order to work on the actual hardware. This Computer is also used as a shared development machine where the team can show each other progress and problems.

4.2 FTDI Adapter

The FTDI USB to FT2232H was used to connect the odroids serial output line to the USB port of the laptop. This USB adapter can be patched through to the `nspawn` container for it to be accessible by our team by allowing devices of type 188/0 in the container.

Unfortunately, this setup is quite fragile, because the usb device is lost to the container whenever it has to reconnect, for example because it unplugged.

So we decided to redirect the data from usb FTDI device to a network port which can easily be connected and disconnected.

The redirection was made using `socat`:

```
/usr/bin/socat \
  tcp-listen:9999,reuseaddr,fork \
  file:/dev/ttyUSB0,nonblock,waitlock=/var/run/tty0.lock,b115200,raw,echo=0
```

This command opens TCP port 9999 as a mirror of the serial port /dev/ttyUSB0 using UART baud rate 115200 encoding 8N1, no flow control and no echo. This allowed the container to access the serial data without needing access to the hardware device itself.

4.3 LAN, WAN and TFTP

The Odroid-C2 setup was installed at the home of Johannes Walcher. He did not like full access to his home network using this jump host, so a DMZ inside a tunneled machine was created.

The network of the nspawn container was masqueraded on the host in order to access the internet and was run in a virtualized network setup. The host was connected using a wireguard tunnel to a server instance running at `l4praktikum.jotweh.com`, which forwarded port 2222 to the host machine which in turn forwarded to the openssh server running inside the nspawn container.

[1] Compare https://delock.de/produkte

The tftp server was running on the host machine to avoid further complicating the network setup, handing out images from the `/srv/tftp` folder of the container. These images where then served to the Odroid when requested without needing to allow ports in the firewall.

In the beginning of the course, this setup was connected to the local wifi, but this setup proved to be insufficient for downloading larger packages such as full toolchain directories. Therefore the setup was moved to a wired connection wich improved the efficiency of the team a lot.

This setup required us to create the virtual network interfaces and bridges on the host and forward the created `tap`-device to the container.

4.4 Managementtool `odroidctl`

The setup should be intuitive to use, so we created a tool to control our interaction with the Odroid-C2. We called the tool `odroidctl` and installed it on the laptop. The tool made it especially convienient when the hardware needed to change, or the basic interaction with the hardware required different command line parameters, because only the tool neded to adopt to the new circumstances. This enabeld a smooth workflow for the whole team.

The source code of the management tool can be found in the repository.

Upon calling `odroidctl` with no arguments, the following usagee guidlines are printed:

```
Usage: odroidctl [command]

Commands:
on turn the device on
off turn the device off
restart the devce off and on
connect connect to the odroid
```

In more detail these perform the following actions:

ON Send a "ON"-Command to the wifi plug in order to turn on the odroid

OFF Send a "OFF"-Command to the wifi plug in order to turn of the odroid

RESTART send a "OFF" followed by ab "ON" command with one second delay.

CONNECT connect to the TTY adapter connected to the odroid using `nc`.

4.5 Options to scale this setup

This setup was working really well for one team to collaborate together. Especially the introduction of the `odroidctl` tool helped hide the complexity of the whole setup and allowed everybody to work on the tasks at hand.

To utilize such a setup for more than one group on one host a more complex firewall setup has to be done in order to isolate the different teams. If more than one development board shall be connected to the server, virtual networks have to be created. One option would be to utilize a managed swich and tag the interfaces into different vlans mapped through to the host, connected to the bridge interface introduced in exercise 5.

5 Challenges with Windows Subsystem for Linux

For this project Tabea Haeseler decided to work with Windows Subsystem for Linux (WSL) since she is familiar with Windows 10. The main goal was here to get a setup for building the different parts for our project. Which did work mostly and showed different challenges such as accessing the windows file system from WSL and from Windows 10 via network device.

WSL is a subsytem for Windows 10 which lets users run programs in a GNU/Linux environment. For that users can download a specific Ubuntu version and access most features, such as Linux-specific compilers, modules and more. Running ELF-binaries is supported for 32-bit and 64-bit and also some popular command line tools such as *grep*, which we used often for searching the linux kernel for porting our systems.
Graphical output is not supported, but was not required for our project. Although it is possible to add a server-based graphical output, which is largely experimental and not supported by Microsoft.

A major difficulty was the underlying Windows file system which does not allow certain characters like "äs file-path-name. With that not all files could be pulled from our repository. After trying many different workarounds, which did not lead to a working result, Johannes let Tabea develop the Genode Application on his remote Ubuntu-Laptop. Another problem here was the missing ability of Windows to be utilized as a DHCP server which can be done in Linux with *tap devices*.

As a conclusion here, WSL can be used for developing non-complex applications on Windows, but not for large applications like our project. Especially the possibility of connecting a development environment such as Visual Studio Code to WSL improves the development process.

5.1 Mounting Partitions with WSL

It is possible to mount different drives in Windows Subsystem for Linux (WSL). We require SD-card mounting only. Officially WSL offers no support for accessing hardware devices. It is not directly possible to access proc/partitions with WSL. The command fdisk is not supported to access proc/partitions which displays available drives in the file system.

However, there exists a workaround for that problem explained in the official Microsoft Blog Post Archive. Since WSL 1 is not a virtual machine, it is possible to access the windows file system DrvFS from within the WSL command line [Gro17].
WSL supports some file systems such as NTFS or FAT and also mounting network locations. With that we are able to access USB sticks or other media using FAT systems [Gro17] [Gro16].

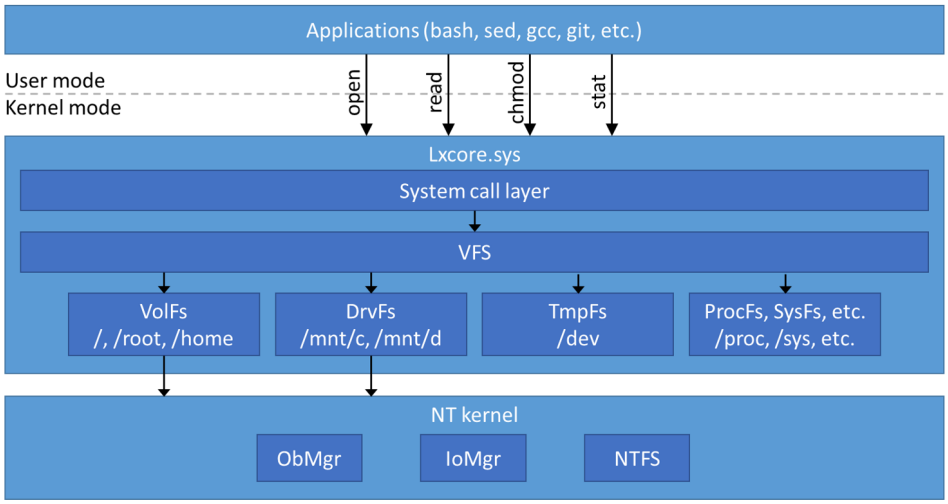


Figure 2: Demonstration of the WSL file system translation [Gro16].

```
//create a new directory
//where the windows drive should be mounted
//withing the WSL subsystem
sudo mkdir /mnt/e
//mount the drive e:/ to /mnt/e
sudo mount -t drvfs e: /mnt/e
```

Mount a drive e: residing in the Windos File System with WSL.

Figure 2 shows the translation of Linux file system operations to NT kernel operations which Windows 10 uses. It supports functionality for linux permissions, for example access, read/write permissions can be adjusted per user with *chmod*. WSL uses the VFS component in the kernel mode to provide this functionality. The VFS component is modeled after the Linux VFS system [Gro16].
When a system call is called, the *system call layer* provides kernel entry points like *chmod, open, read, ...* These calls are file related, so they are forwared to the *VFS layer*. There the *VFS layer* can resolve the calls, invoking other kernel operations where neccessary (e.g. *lookup, chmod, ...*). Calls to a file descriptor are handled by the operations defined by the specific file system. Basically the VFS system behaves similar to the Linux VFS system [Gro16].

But the representation of files is done in the Windows DrvFs or VolFs, which is limited in its file path naming. WSL interacts only with the DrvFS or VolFs system when accessing files on the disk. DrvFS provides interoperability with Windows while VolFs supports Linux file system features [Gro16].

VolFs is used to store the Linux system files, for example the content of the WSL Linux home directory. The VolFs mounts to VFS root directory, which is located within the Windows DrvFS. The home directory can also be accessed via network device connect from the Windows file system. There the user can copy, write, access files. But this depends on the given access permissions, which can be changed in WSL with *chmod*. For example copying SSH keys created by WSL cannot be accessed from the Windows user from the Windows filesystem. The permissions have to be adjusted within WSL to give access to the Windows file system [Gro16].

The main problem with this mounting mechanism is the compatibility between file systems. In the end WSL translates the file systems into each other - from the simulated Linux VFS to a file system windows 10 uses. Linux does support more symbols for file paths, that Windows does not allow. With our project, some file paths in the provided repository did not work with WSL, since the underlying Windows 10 system blocked the requested file path naming. With mounting another location the problem did not vanish, because the missing virtualization of WSL.
In our application we use FAT file system for the creation of a bootable SD-card. Later, before using visual studio code to access the remote Ubuntu system, we used the network mounting ability to access the remote system via openSSH. This did not work as intended and came with its own problems of failing connections and file system differences.

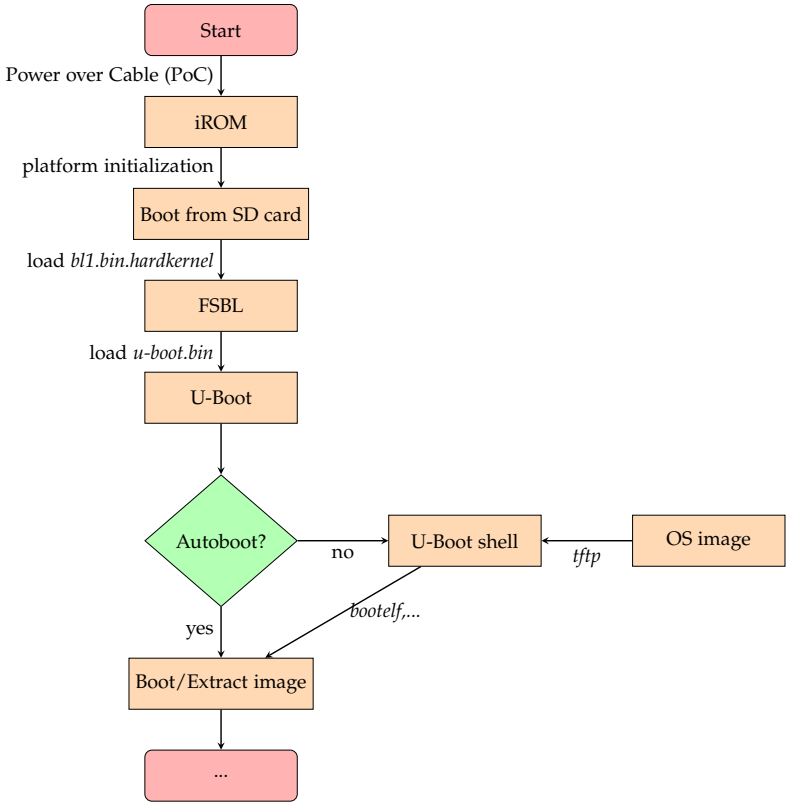
While working on our project, Microsoft released WSL 2 which supports full system call compatibility and runs on a full linux kernel. Microsoft changed the system fundamentally for better file system performance and usability. As a new change, WSL 2 uses virtualization technology which WSL 1 did not require on hardware level.

6 ARMv8, Bootstrapping, U-Boot

This section provides additional information on the hardware platform, as well as U-Boot and the ARMv8 architecture in order to better understand the challenges we faced with U-Boot on the Odroid-C2 during the practical course.

6.1 Odroid-C2 Boot Process

To fully understand U-Boot, it is necessary to comprehend the boot process of the Odroid-C2. The following description of the boot process is mainly based on the implementation in the U-boot directory from Hardkernel on the Odroid-C2 branch.



System booting is initialized with providing Power over Cable (PoC), as can be seen in [S905.20] under *System booting section*. After powering on the Odroid-C2, reset brings the platform into the startup state and forces the device to boot from the inserted SD card. After platform initialization through iROM, the first stage bootloader (FSBL) *bl1.bin.hardkernel* is loaded into RAM and continues the booting process. The CPU jumps to the reset vector and executes the code from *u-boot/arch/arm/cpu/armv8/start.S*. By default the device operates in Little Endian and MMU as well as i/dCache are disabled. From within *start.S*, *u-boot/arch/arm/lib/crt0.S* is called. This file handles target-independent stages of the U-Boot start-up where a C runtime environment is needed.

The U-Boot binary *u-boot.bin* serves as a second stage boot loader that loads the kernel and userland components into memory. U-Boot configuration is represented within a U-Boot environment memory area on top of the basic U-Boot binary. If boot and system partition are provided in addition to the FSBL, U-Boot and U-Boot environment, U-Boot can load a kernel/operating system into memory. A kernel needs some user land components in order to build a full functioning operating system.

The entire kernel and userland are usually bundled together inside an *elf* or binary file that can be directly executed on U-Boot. The startup process of applications on *Fiasco.OC* together with *L4Re* can be reconstructed in *contrib/.../foc/l4/pkg/bootstrap/server/src/startup.cc* from the exercise repository. The startup routine inside this file is started from *crt0.S*. Initially, platform configuration is checked before the L4 bootstrapper can be started. The binary will get extracted and positioned accordingly in RAM. Afterwards, the kernel (Fiasco.OC), the root pager (Sigma0) and the root task (Moe) are started, before the L4Re user land will be addressed. The bootup process for Genode is similar as it also provides userland components for the Fiasco.OC kernel.

More sophisticated operating systems involve several files for bootstrapping. The Ubuntu image from Hardkernel for the Odroid-C2 requires a *.dtb* file, an *initrd* file and an actual *Image* file for booting[ODR20.2][SO20.2]. The *.dts* file holds the information for the Flattened Device Tree (FDT) as a binary blob of the actual device tree information from *.dts* and *.dtsi* include files. A Device Tree is used to provide a way to describe non-discoverable hardware[DTS20]. *Initrd* stands for initial ramdisk and is responsible for loading a temporary root file system into memory[RAMDISK20]. The actual operating system is located inside the *Image* binary file. These image files come in various variants like *Image, uImage* and *zImage*[SO20.1]. *Image* is the default binary file, *zImage* is a compressed version of the *Image* file and *uImage* is the *Image* file together with a U-Boot wrapper around.

If loading the respective files fails for some reason, U-Boot falls back to the basic U-Boot command line. The U-Boot environment provides user configuration through the shell and basic commands for booting like *bootelf, bootp, bootm, go* or the *tftp* command for file transfer. Through these commands and settings, additional files can be loaded over TFTP and booted directly from the U-Boot command line.

6.2 U-Boot

As was seen in the previous section, U-Boot is utilized as the second stage bootloader for the Odroid-C2 that loads binary files into respective locations in RAM and hands over control to the extracted image environment. Hardkernel provides a separate U-Boot repository for different Odroid boards, apart from the upstream U-Boot repository from DENX.

6.2.1 Toolchain

In order to built U-Boot from a specific laptop as host for the Odroid-C2 as target platform, a Linaro Toolchain needs to be setup on the host. It is worth mentioning that the U-Boot build process from Hardkernel is only applicable on x86 Linux PCs. The system must be instructed to use the cross-compiler through environment variables *ARCH=arm* and *CROSS_COMPILE=path/to/toolchain*. *ARCH* specifies that the architecture of the build should be ARM and the *CROSS_COMPILE* variable is set to update the build environment towards the Linaro Toolchain. For the practical course, we used the Linaro Toolchain *x86_64_aarch64-linux-gnu* version *7.5.0* from 2019. The cross-compiler is designed for *x86_64* host machines that compile *linux-gnu* operating system for *aarch64/ARM64* target machines like the Odroid-C2. The specific cross-compiler can be identified by the target triplet of the toolchain ending. For this particular toolchain, the vendor field of the triplet was not further specified[OSDEV20].

6.2.2 U-Boot build

After setting up the toolchain, it is possible to build a U-Boot image for the target platform, in our case the Odroid-C2. This build process is regulated via Makefiles in the U-Boot repository. The command *make O=build_odroidc2 odroidc2_defconfig* is used to create a build directory for the Odroid-C2 by using the default configuration *odroidc2_defconfig* for the platform provided by Hardkernel. The U-Boot image *u-boot.bin* can then be built via the *make* command. When *make menuconfig* is applied before building the image, a custom configuration for U-Boot can be set. For example, it might be beneficial to disable the default autobooting setting for U-Boot from Hardkernel. Autobooting in U-Boot is determined by the *bootdelay* and *bootcmd* variable. The *bootcmd* variable defines a command executed by U-Boot automatically after the *bootdelay* countdown is over[EMCR20]. For development purposes, it makes more sense to disable autobooting and jump directly into the default U-Boot shell in order to load the images for booting manually over TFTP. The *bootcmd* value can be set inside the *menuconfig*. In the recommended U-Boot version from Hardkernel, the binary *u-boot.bin* will be placed inside a directory together with the pre-compiled FSBL *bl1.bin.hardkernel* and a script *sd_fusing.sh*.

6.2.3 U-Boot installation

To install the bootloaders on the Odroid-C2, the SD card must be flashed with the images. This process requires the SD card of the SBC to be inserted in a SD card reader. The script *sd_fusing.sh* is responsible for installing the bootloader onto the SD card in the recommended Hardkernel U-boot version. Independent of the U-boot version applied, the information of the *sd_fusing.sh* script specifies the locations the FSBL and U-Boot needs to be loaded to for the Odroid-C2. The script must be further provided with the device name of the SD card, e.g. *mmcblk0*, that can be detected via the *lsblk* command. Internally, the script makes use of the *dd* command in order to overwrite the output file

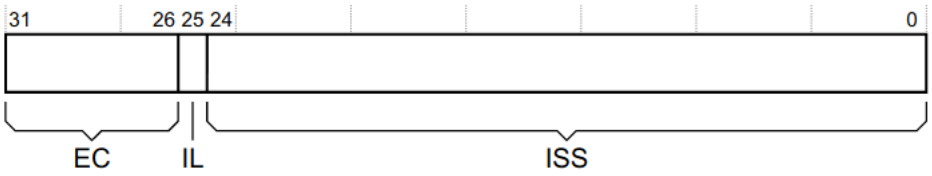
with the content of the input file. The *sync* command is applied after writing the images to disk as it makes sure that pending modifications to the file system that are kept in volatile memory are written to the underlying file system on the SD card. Thus, the command synchronizes RAM and SD card memory[SYNC]. The newly flashed SD card can now be removed from the SD card reader and reinserted into the Odroid-C2 SD card slot. When powering on the Odroid-C2, the platform should boot into the U-Boot command line if autoboot is disabled.

6.2.4 Setting up TFTP

During the process of embedded development, it is necessary to frequently load new images onto the platform. One basic approach would be to also flash the SD card with the new image, whenever it should be applied to the Odroid-C2. This requires the developer to remove the SD card from the SBC, flash the image on the laptop and reinsert it again back into the Odroid-C2. This approach does not scale for frequent builds. The preferred approach is to use TFTP to load the image from a remote TFTP server into the RAM of the Odroid-C2 via the U-Boot command line. As the image is placed in volatile memory, a reboot would delete temporary settings and files loaded into memory. This is not a major issue during development, as the demonstrated process is designed for speed instead of stability. If required in non-volatile memory, the image can also be transferred from RAM into the SD card in U-Boot or the SD card can be flashed. In order to use TFTP, we first had to setup a TFTP and DHCP server on the laptop the Odroid-C2 is connected to. The TFTP server is provided by the *tftpd-hpa* service that provides TFTP over port 69. The TFTP server needs to be configured according to the documentation in the Wiki and the *tftpd-hpa* service restarted. The *dnsmasq* service provides DHCP functionality for auto IP assignment for the Odroid-C2. The Odroid-C2 must be connected to the laptop via ethernet to use the TFTP service. Furthermore, the U-Boot environment of the Odroid-C2 must be correctly configured to use TFTP. The TFTP server ip can be set with the command *setenv serverip <ip>*. The Odroid-C2 will be assigned an IP address over DHCP with the *dhcp* command. After configuring the platform, it is possible to load images *<image>* to a specific memory location <mem-loc> with the U-Boot command *tftp <mem-loc> <image>* or the image is already pulled through the *dnsmasq* command. When applying a boot command like *bootelf, bootm* or *go* on a specific memory location, the image at that location will be extracted and placed at locations specified in the binary. It is important to include the image size into the memory location used in the *tftp* command to load the image onto the board. If an image would be simply placed at the location in memory where the actual extracted binary will be executed, the extraction operation would overwrite parts of the image while extracting the image, thus rendering specified location area useless. The boot process for that memory location would most definitely fail. Furthermore, this overwriting operation of the provided image would result in modifying executable code which is marked as read-only.

6.3 Synchronous Abort Handler

If the booting process of an image within U-Boot itself fails, the exception handler prints the exception cause over UART for debugging purposes. The developer can simply read the output over the serial line, as UART is already ported for the Odroid-C2 in the U-Boot version from Hardkernel. A common exception we encountered during the porting task was the *Synchronous Abort exception* that was handled by the *Synchronous Abort Handler*. An exception can be roughly differentiated into an interrupt and an abort[CYPRESS20]. While an interrupt is under the control of the developer, an abort is described as unintended exceptions resulting due to invalid or unsuccessful access of memory. A synchronous abort is generated as a result of an attempted execution of an instruction stream where the return address provides detail of the instruction that caused the exception[ARM20.2]. According to *arch/arm/lib/interrupts_64.c* in the DENX U-Boot repository, the exception handler prints the exception class (= *Synchronous Abort*), as well as the *Exception Syndrome Register* (ESR) content and system registers before resetting the CPU. This information can then be used by the developer for troubleshooting. Especially the value of the *ESR* allows for further debugging. To understand the *ESR* register, further information on the ARM architecture needs to be provided. The ARMv8-A architecture provides different privilege/exception levels for Kernel code and applications, as those modules require different levels of hardware access. The current level of privilege can only change when the CPU takes or returns from an exception. These privilege levels are referred to as Exception Levels in the ARMv8-A architecture. There exist four different exception levels *EL0-EL3*, where *EL0* has the lowest privilege and *EL3* the highest. Application code runs in EL0, operating system runs in EL1, the hypervisor runs in EL2 and low-level firmware and security code runs in EL3[ARM20.1]. As the porting task for this practical course involved kernel and userland porting, there was no hypervisor involved. Furthermore, U-Boot was already successfully ported by Hardkernel. As a result, error messages during the boot process from U-Boot come from kernel or application code. This means that exceptions are raised in EL0 or EL1. The exceptions taken from EL0 result in exceptions taken from EL1, thus no separate ESR is provided for EL0 by the ARM architecture. With this information, we can conclude that debugging information provided by the *Synchronous Abort Handler* is from the *ESR* in EL1. More information on this register is documented in the *ARM Architecture Reference Manual* for the *ARMv8* architecture under section *D8.2.24 ESR_EL1, Exception Syndrome Register (EL1)*[ARMV8].



The *ESR_EL1* register is 32-bit wide and holds syndrome information for an exception taken to EL1. The Exception class (EC) bits [31:26] indicate the reason for the exception

that this register holds information about. The Instruction Length (IL) bit [25] differentiates between 16-bit and 32-bit instruction length for synchronous exception. The remaining bits of *ESR_EL1* account for the Instruction specific syndrome (ISS) [24:0]. The content of the ESR value in the error message can then be compared with the information provided in the ARM document. With this approach, we were able to debug the logging information and reason about the cause for the exception.
Further debugging information is provided by the Synchronous Abort Handler by printing the content of system registers. The ARM architecture has general purpose registers R0-R14, where R13 is stack pointer (SP) and R14 is link register (LR). The program counter is described by R15. Especially the contents of the *Link Register* (LR) and *Exception Link Register* (ELR) are of interest to us. The LR is a special register that holds return link information and thus holds the address to return to when a function call completes. The ELR on the other hand holds the address to return to if an exception is taken. In our case, the content of *ELR_EL1* is provided by the Synchronous Abort Handler.

6.4 Challenges & Solutions

6.4.1 Enabling *bootelf* in U-Boot

As the images created by the FOC/L4Re and FOC/Genode build system are *elf* files, U-Boot should be equipped with the *bootelf* command. When applying the recommended U-Boot version from Hardkernel however, *bootelf* is not enabled by default. In fact, *bootelf* can not even be made available through *make menuconfig*. We have found multiple workarounds for the missing *bootelf* command. With some research inside the U-Boot repository from Hardkernel, it turns out that the *bootelf* option can be directly compiled into the U-Boot image. The *README* file in the U-Boot repository specified that *bootelf* can be activated through the *CONFIG_CMD_ELF* variable. In order to change the default configuration, two files need to be modified: *include/configs/odroidc2.h* and *configs/odroidc2_defconfig*. When recompiling the image, the *bootelf* command should be available in the U-Boot command line. This re-compilation does not necessarily update the U-Boot binary in the *sd_fuse* directory.
As an alternative, the *elf* file can be translated into another file format, as those files can be executed with the commands *bootm* or *go* that come with U-Boot by default. The *elf* file can be translated into a binary executable with the *objcopy* command. More precisely, the *aarch64-linux-gnu-objcopy* cross-compiler version from the Linaro Toolchain should be used. With the tool *mkimage* images specifically for U-Boot can be generated.

6.4.2 Different U-Boot versions

We started the porting task with porting the Fiasco.OC kernel together with the L4Re to our platform, before porting towards the more complex Genode OS Framework. To test our current porting implementation, we compiled the *bootstrap_hello* application for FOC/L4Re, pulled the image over TFTP on the Odroid-C2 and started bootstrapping

the application image. We hoped that the output of the Odroid-C2 would give us more information for further porting steps. Unfortunately, the only output we received was the Synchronous Abort handler information from U-Boot before platform reset is executed. Furthermore, with the basic recommended U-Boot version from Hardkernel, bootstrapping fails within the first instruction of the entire image. We concluded that the U-Boot build for the Odroid-C2 was defective. As there was no leverage point for actual porting in the console output at that point, we agreed on putting focus on identifying and fixing the U-Boot build before pursuing further porting attempts. We assumed that fixing U-Boot would be more effective because with actual valuable console output, porting attempts would be more efficient. Our approach was to test different formats of the FOC/L4Re binary on multiple different U-Boot builds.

Odroid-C2 branch
We started our investigation in the U-Boot repository from Hardkernel. In this repository multiple branches for different platforms exist. As was recommended by Hardkernel, the U-Boot build for the Odroid-C2 should be based on the *odroidc2-v2015.01* branch, build with the *aarch64-none-elf* cross-compiler that is based on the Linaro toolchain version *4.9* from 2014 instead of the most recent version *7.5* from 2019. After building the image, the *sd_fusing.sh* script must be applied in order to flash the SD card with the newly build image. As all changes to this branch resulted in faulting behaviour of U-Boot during bootstrapping of images, we started experimentation with other branches, builds and configurations.

Odroid-N2 branch
We also tried an alternate branch on the U-Boot repository from Hardkernel. As the head of the repository was pointing to the *odroidn2-v2015.01* branch that was also based on the 2015 version of U-Boot, we figured that it was at least worth testing builds for this branch. In order to compile the *N2* branch, two different cross-compilers must be applied: *aarch64-none-elf* and *arm-none-eabi* that are both based on the Linaro toolchain version *4.8*. Compilation was directed with the Odroid-C2 configurations. Unfortunately, multiple architecture-dependent variables had to be set during compilation. Missing values resulted in compilation errors. These values had to be specifically set within multiple files distributed across the entire N2 branch. For time reasons, we did not further investigate into that direction as the build errors did not stop. Furthermore, we believed that there must exist an easier solution for our problem.

U-Boot from Ubuntu image
After disappointing results from builds based on the Hardkernel U-Boot repository, we looked at other repositories and builds. We were hopeful that we could use the U-Boot that comes with the Ubuntu minimal image that we had to flash onto the SD card in exercise 1 of the practical course. As we were able to successfully boot Ubuntu with this image, we concluded that the respective U-Boot version of that image must be working

correctly. To implement our idea, we first flashed the SD card with the minimal Ubuntu image from Hardkernel. Thereafter, we removed the *Image* file from the SD card in order to force U-Boot to fall back to the basic command line. Unfortunately though, the bootstrapping process of the FOC/L4Re image still caused the same synchronous exception.

U-Boot from another repository
During our experiments, we were also looking for help on the internet. We came across a promising thread in one of the Hardkernel message boards[2]. At that time, we were trying to get another U-Boot version working for the Odroid-C2 that would resolve our bootstrapping problems. By following the instructions from *board/hardkernel/odroid-c2/README* of the recommended repository from the message board, we were able to build a U-Boot version for the Odroid-C2. Unfortunately, basic commands like *tftp* or *dhcp* were not pre-configured in the resulting U-Boot image. As a result, we did not further follow that approach.

Image files
The U-Boot build process creates elf files by default. After testing multiple U-Boot builds with negligible results, we concluded that the root of the issue might be in the elf file itself. During the U-Boot experiments, we also tried different image formats. Besides the elf format, binary executables were build with *aarch64-linux-gnu-objcopy* from the original elf file and tested on the Odroid-C2. With the *mkimage* command even *uImages* were translated and executed on the platform. Unfortunately, it did not seem to make a big difference, as U-Boot was still faulting with Synchronous Abort Handler, yet the error codes were different.

Debugging/Troubleshooting
Different image formats and U-Boot builds resulted in different error messages from the Synchronous Abort Handler. The important information for debugging purposes can be seen through the LR, ELR and ESR values. For some builds the bootstrapping sequence failed at the first instruction, others were able to execute a few instructions before faulting. These results led us to believe that we were in fact on the right track, as our changes resulted in different output from the Synchronous Abort Handler. Different U-Boot builds also resulted in distinct ESR values that could be debugged with the ARM TRM. Output from the Synchronous Abort Handler does not always result in valuable debug information. A common ESR value we encountered is 0x02000000 = 0b000000 1 00000000000000000000000000. According to the TRM, the respective value defines exceptions with unknown reason. Debugging what went wrong is difficult if not impossible without hardware debug tools. Whenever we encountered aforementioned message, we usually did not debug any further and proceeded with other builds and configurations to test. More valuable information could be taken from

[2] `https://forum.odroid.com/viewtopic.php?t=20869`

ESR value of 0x86000010 = 0b100001 1 00000000000000000000010000. The EC bit values 0b100001 hint to an instruction abort without a change in exception levels. No change in exception level means that the exception is taken from an exception level that is using AArch64[ARMV8]. ISS bits [5:0] 0b010000 further specify a *Synchronous External Abort*, meaning that while the CPU was trying to make a memory access, the origin of the abort is external to the processor, i.e. the access did not fault in the MMU. The synchronous nature of the abort allows us to trust the ELR and LR values for debugging purposes. Another ESR value that was encountered during the practical course was the 0x96000010 = 0b100101 1 00000000000000000000010000. The EC bits 0b100101 of the ESR value represent a data abort that was taken without a change in exception level. A Data Abort Exception is a response by a memory system to an invalid data access[CYPRESS20]. The ISS bits [5:0] 0b010000 further specify the exception being a result of a *Synchronous External Abort*. Although not realized during that time, in hindsight it would have been possible to conclude from these messages that U-Boot was working as expected. The *Synchronous External Abort* means that the reason for faulting comes from outside the CPU, hinting towards porting issues instead of faulty U-Boot builds. An odd Synchronous Abort message was printed when using the U-Boot build from the Ubuntu image provided for Odroid. The procedure resulted in an ESR value of 0xf20003e8 = 0b111100 1 00000000000000001111101000. The TRM describes that message as a software breakpoint instruction exception.

6.4.3 Final U-Boot version

For our final U-Boot version, we integrated the 2020 mainline U-Boot repository from DENX. Our approach was to use the U-Boot image built with the DENX repository together with the FSBL *bl1.bin.hardkernel* from the C2 branch of the Hardkernel repository. We started in the mainline repository with applying the default U-Boot configuration *make odroid-c2_defconfig* before disabling the autobooting feature with *make menuconfig*. Then we built the U-Boot binary *u-boot.bin* with the common *make* command. After building the image, the U-Boot binary needs to be signed with the x86-64 executable, *aml_encrypt_gxb* that comes shipped together with the u-boot source code. Furthermore, other files from *fip/gxb* are also included during the signing process. Afterwards, the signed image can be written to the SD card. With the right porting steps applied to the FOC/L4Re image, we were now able to finally start booting the image on that particular U-Boot version and receive readable output from UART. At that point, only one of our Odroids was able to print text in a readable format over the serial line during startup of the FOC/L4Re image. The other Odroid was still faulting with the usual Synchronous Abort Handler messages. This was surprising to us at first, as we applied the same cross-compiler settings with the same changes to the porting repository. We even compared the board revision of the two Odroids, with both being *REV 0.2 20171114*. To be entire sure that both attempts were using the exact same conditions, we even used the FOC/L4Re image that was build by the laptop the successfully booting Odroid was connected to. In the end, we came to the conclusion that the only difference between

the two approaches was the U-Boot version. The working Odroid was using the 2020 masterline U-Boot version, while the faulting Odroid was using the 2015 U-Boot version from the C2 branch recommended by Hardkernel. After updating the Odroid with the faulty behaviour to the 2020 mainline U-Boot version, the porting results could be finally verified by both Odroids.

Several weeks of the porting task were spent on the notion of a flawed U-Boot build and multiple different U-Boot branches and configurations were considered during that time. Those weeks were still valuable after all as we deepened our knowledge about the hardware platform as well as the boot process of the Odroid-C2. The different U-Boot configurations and builds further resulted in different error codes before resetting. After applying every possible U-Boot configuration known to us, we realized that the overall problem was mainly a porting issue after all. This conclusion came after building the hello world demo example of U-Boot and executing it on the platform. U-Boot was able to execute the binary, thus proving that the booting issues stem from our porting attempts in the FOC/L4Re image.

7 Porting to Fiasco.OC/L4Re

The porting task for Fiasco.OC and L4Re comprises the peripherals UART, Interrupt Controller, Timer, Watchdog Timer and NIC. Due to time constraints, there were no attempts to port the NIC functionality for the Odroid-C2. Our general approach to porting was based on the information provided by the reference platforms *zynqmp* and *rcar3* that also implement the ARMv8 architecture. Our task was to find architecture-specific information for the Odroid-C2 and adapt the provided files from the reference boards. Architecture-specific information was either taken from the Amlogic S905 TRM[3], the U-Boot repository from DENX[4] or the Linux Kernel[5] itself. If none of these resources provided the required information, we also used online resources.

7.1 Memory Mapping

The first and most straight forward porting information is setting up the memory map. Specifically the RAM start address and address range needs to be identified and several files in the project repository must be adapted to these values. According to the *Memory Map* section in the TRM, the DDR has a range of 0x00000000-0xBFFFFFFF, thus 3GiB. But the TRM also states that the Odroid-C2 has 2GiB of DDR memory, which would result in a DDR range of 0x000000000-0x07fffffff. This assumption is further supported by the memory entry in */arch/arm64/boot/dts/amlogic/meson-gxbb-odroidc2.dts* of the Linux Kernel:

```
memory@0 { ... reg = <0x0 0x0 0x0 0x80000000>; };
```

This verifies that the actual DDR range must be in fact 2GiB starting from address 0x0. We assume that the 3GiB is for future board revisions that might offer 3GiB of RAM in order to provide backward compatibility. The acquired memory information was then used to adapt several files towards the Odroid-C2 architecture. In the *Modules* file of the Fiasco.OC kernel, we specified the following RAM base address:

```
RAM_PHYS_BASE          := 0x0
```

More architecture-specific information must be provided in the L4Re directory of our repository. The following line was added to the *setup_memory_map()* function in the *odroidc2.cc* file:

```
mem_manager->ram->add(Region(0x000000000, 0x07fffffff, ".ram", Region::Ram));
```

Architecture specific information must be also provided to the configuration file *odroidc2.conf* in the L4Re directory:

```
PLATFORM_RAM_BASE    = 0
PLATFORM_RAM_SIZE_MB = 2048
```

[3] https://dn.odroid.com/S905/DataSheet/S905_Public_Datasheet_V1.1.4.pdf
[4] https://gitlab.denx.de/u-boot/u-boot
[5] https://github.com/torvalds/linux

7.2 UART

Universal Asynchronous Transmitter Receiver (UART) is a protocol used for full-duplex serial communication. UART is implemented as a hardware peripheral on many microcontrollers. It transforms parallel data on the system bus into a serial data stream that is transmitted over a serial line to the recipient that converts the received serial stream back into parallel data. A baud rate generator defines the speed at which the transmitter/receiver has to send/receive data. Data to be sent is placed inside the transmit holding register (THR) and will be shifted bit-wise by the transmit shift register (TSR) onto the serial line. On the receiving end, the receive shift register shifts the incoming data bitwise into the receive holding register. From there, the data can be consumed for further computation.
The kernel-specific information concerning UART is placed in *uart-arm-odroidc2.cpp* and *kernel_uart-arm-odroidc2.cpp*. In the L4Re directory, the file *odroidc2.cc* must be adapted to architecture-specific UART information. First, we looked into *.../fiasco/src/kern/koptions-def.h* in order to find out what information needs to be set in *odroidc2.cc*:

```
struct Uart
{
  Unsigned32  base_baud;    ///< Base baud rate of the UART (if applicable)
  Unsigned32  baud;         ///< Baud rate (this is the baud rate to use)
  Unsigned16  irqno;        ///< (Receive) IRQ
  Unsigned8   reg_shift;    ///< Shift value for register addressing
  Unsigned8   access_type;  ///< Accesstype of UART: unset, MMIO or ports
  Unsigned64  base_address; ///< Start address of UART
} __attribute__((packed));
```

The different fields of the UART struct must be specifically set in the *init()* function of the *odroidc2.cc* file with architecture specific information. According to the TRM, the Odroid-C2 contains five different UART peripherals:

UART	**Power Domain**
UART0	Located in the EE domain
UART1	Located in the EE domain
UART2	Located in the EE domain
UART0-AO	Located in the AO domain
UART2-AO	Located in the AO domain

Three of these UARTs are located in the EE domain, the others in the AO domain. According to the TRM, the Cortex-A53 can't be powered without the EE domain. The EE domain can't be powered up without the AO domain. To understand which UART to use, we conducted *arch/arm64/boot/dts/amlogic/meson-gxbb-odroidc2.dts* from the Linux Kernel:

```
aliases { serial0 = &uart_AO; ... };
```

```
...
&uart_AO { ... };
```

As this is the only UART information specified in *meson-gxbb-odroidc2.dts*, we already knew that the UART for the Odroid-C2 must be located in the AO domain. Therefore, the potential UARTs could only be UART0-AO or UART2-AO. When looking into *arch/arm64/boot/dts/amlogic/meson-gxbb.dtsi*, there is more information provided for all five UARTs:

```
&uart_A { clocks = ... < ... CLKID_UART0> ...; ... };       -> UART0
&uart_B { clocks = ... < ... CLKID_UART1> ...; ... };       -> UART1
&uart_C { clocks = ... < ... CLKID_UART2> ...; ... };       -> UART2
&uart_AO { clocks = ... < ... CLKID_AO_UART1> ...; ... };   -> UART0-AO
&uart_AO_B { clocks = ... < ... CLKID_AO_UART2> ...; ... }; -> UART2-AO
```

Therefore, we need to find architecture specific information for UART0-AO, which is referred to as *uart_AO* in the Linux Kernel.

7.2.1 Base Address

The UART base address for the different UARTs of the Odroid-C2 is given in the following table taken from the TRM:

UART	**Address Range**
UART0	0xc11084c0 - 0xc11084d4
UART1	0xc11084dc - 0xc11084f0
UART2	0xc1108700 - 0xc1108714
UART0-AO	0xc81004c0 - 0xc81004d4
UART2-AO	0xc81004e0 - 0xc81004f4

With this information, we concluded that the address range for UART0-AO is 0xc81004c0 - 0xc81004d4. This can be verified by *arch/arm64/boot/dts/amlogic/meson-gx.dtsi* from the Linux kernel:

```
aobus: bus@c8100000 { ... uart_AO: serial@4c0 { ... }; ... };
```

The Linux kernel confirms the data from the TRM, as 0xc8100000 + 0x004c0 = 0xc81004c0 is the base address of UART0-AO. Further confirmation comes from *configs/odroid-c2_defconfig* of the U-Boot repository from DENX:

```
CONFIG_DEBUG_UART_BASE=0xc81004c0
```

This base address is used in the *odroidc2.cc* file:

```
kuart.base_address = 0xc81004c0
```

7.2.2 Register Shift

When comparing the implementations of the shift register from different platforms, mostly the *reg_shift* variable was either set to 0 or to 2 in the *init()* function of the *odroidc2.cc* file. More information on the shift register is given in *drivers/serial/Kconfig* from the DENX U-Boot repository:

```
Some UARTs (notably ns16550) support different register layouts where the
registers are spaced either as bytes, words or some other value. Use this
value to specify the shift to use, where 0=byte registers, 2=32-bit word
registers, etc.
```

Based on this information, all we need to do is to find out if the SoC uses 8-bit registers (*reg_shift = 0*), 16-bit registers (*reg_shift = 1*), 32-bit registers (*reg_shift = 2*) or 64-bit registers (*reg_shift = 4*) for UART. According to the TRM, UART registers are 32-bit wide on the Odroid-C2. Therefore, we can set the following value in the *init()* function of the *odroidc2.cc* file:

```
kuart.reg_shift = 2
```

7.2.3 Baud rate

In the *odroidc2.cc* the baud rate, as well as the base baud rate must be specified. The baud rate *kuart.baud* was set to the standard value of 115200. The decision was based on the fact that every platform in the L4Re directory set the baud rate to 115200.
The base baud rate *kuart.base_baud* on the other hand was different for every platform. The S905 SoC requires the 24MHz oscillator to generate the system clock. The following information is provided by *arch/arm64/boot/dts/amlogic/meson-gxbb.dtsi* from the Linux kernel for the UART0-AO:

```
&uart_AO { clocks = <&xtal> ...; ... };
```

When searching for the clock name *xtal* in the Linux kernel, *arch/arm64/boot/dts/amlogic/meson-gx.dtsi* provided the following:

```
xtal: xtal-clk { ... clock-frequency = <24000000>; ...};
```

This information states that the uart base clock has a clock frequency of 24000000 Hz = 24 MHz. This number is further verified by *configs/odroid-c2_defconfig* in the U-Boot repository form DENX:

```
CONFIG_DEBUG_UART_CLOCK=24000000
```

According to *setserial* man pages, the UART baud base is the clock frequency divided by 16. This would result in $24000000/16 = 1500000$. To verify this claim, we loaded the minimal Ubuntu image for the Odroid-C2 and executed the *setserial* command to print the following information for the UART (=*/dev/ttyS0*, ttyS0 is the device for the first UART serial port[TTYLX20]):

```
root@odroid:~# setserial -a /dev/ttyS0
        Baud_base: 1500000, ...
```

With this information, the following two lines can be added to *odroidc2.cc*:

```
kuart.base_baud    = 1500000;
kuart.baud         = 115200;
```

Another platform that uses the same base baud is the *sunxi* platform. It turns out that the sunxi clock system also implements a 24 MHz oscillator[SUNXI], which therefore results in the same base baud rate *kuart.base_baud = 1500000*. This further verifies our choice.

7.2.4 IRQ number

In order to find the correct IRQ number, we first looked into the TRM of the Odroid-C2. The interrupt source table of the TRM lists the following IRQ numbers for the different UARTs:

```
uart0_irq = 54
uart1_irq = 105
uart2_irq = 123
ao_uart2_irq = 229
```

As we already know, we need to find the IRQ number for *ao_uart0_irq*, which is not listed in the IRQ table. The required information is provided by *arch/arm64/boot/dts/amlogic/meson-gx.dtsi* from the Linux Kernel:

```
aobus: bus@c8100000 { ...
        uart_AO: serial@4c0 { ...
                interrupts = <GIC_SPI 193 IRQ_TYPE_EDGE_RISING>;
        ... };
... };
```

Therefore, the IRQ number for *ao_uart0_irq* is 193. With this information, we can add the following line to *odroidc2.cc*:

```
kuart.irqno = 193;
```

According to the TRM, this IRQ number belongs to the *mali_irq_gpmmu* as interrupt source.

7.3 Interrupt Controller

An interrupt controller helps the CPU to handle interrupt requests (IRQs). These IRQs may originate from different sources, e.g. timer interrupts or external I/O devices. Furthermore, they may happen simultaneously. To properly handle the incoming IRQs,

the interrupt controller prioritizes specific types of IRQs in order for the CPU to jump to the most appropriate interrupt service routine (ISR). The interrupt controller on the SoC supports 256 interrupts in the chip. Interrupt numbers 0-15 are reserved for Software Generated Interrupts (SGI). These interrupts are specifically generated by software and are most commonly used for inter-core communication. Interrupt numbers 16-31 are reserved for Private Peripheral Interrupts (PPI). These interrupts are generated by a peripheral that is private to a core. Interrupt numbers 32-255 are reserved for Shared Peripheral Interrupts (SPI)[INTRPT20]. These interrupts are generated by peripherals that are shared among all cores of the system, thus representing system-wide IRQs. The TRM only specifies the 224 SPI IRQ numbers (32 - 255).
The porting information for the programmable interrupt controller (PIC) is located in *pic-arm-odroidc2.cpp* that uses architecture specific information from *mem_layout-arm-odroidc2.cpp*. The Linux Kernel file */arch/arm64/boot/dts/amlogic/meson-gx.dtsi* provides the following base addresses:

```
gic: interrupt-controller@c4301000 {
        compatible = "arm,gic-400";
        reg = <0x0 0xc4301000 0 0x1000>,
              <0x0 0xc4302000 0 0x2000>,
              <0x0 0xc4304000 0 0x2000>,
              <0x0 0xc4306000 0 0x2000>;
};
```

With the *ARM Generic Interrupt Controller Architecture Specification*[6], we get more information on the different GIC control register. According to the document, *Gic_cpu_phys_base* refers to a CPU interface register address (GICC), *Gic_dist_phys_base* indicates a distributor register address (GICD), *Gic_v_phys_base* stands for the base address of a virtual CPU interface register (GICV) and *Gic_h_phys_base* indicates a virtual interface control register that is typically accessed by a hypervisor (GICH). Furthermore, the Linux Kernel entry also states that the GIC of the Odroid-C2 is compatible with the *CoreLink GIC-400 Generic Interrupt Controller*[7]. The *GIC-400 register map* section of specified document provides the following information:

Address Range	**GIC-400 functional block**
0x0000-0x0FFF	Reserved
0x1000-0x1FFF	Distributor (GICD)
0x2000-0x3FFF	CPU interfaces (GICC)
0x4000-0x5FFF	Virtual interface control block (GICH)
0x6000-0x7FFF	Virtual CPU interface (GICV)

The *Memory Map* section of the Amlogic TRM states that the GIC address range is 0xC4300000-0xC4307FFF. With a base address of 0xc4300000, we can calculate the base

[6] https://static.docs.arm.com/ihi0048/b/IHI0048B_b_gic_architecture_specification.pdf
[7] https://static.docs.arm.com/ddi0471/a/DDI0471A_gic400_r0p0_trm.pdf

addresses of the different GIC registers based on the register offsets provided in the table above:

```
GICD: 0xc4300000 + 0x1000 = 0xc4301000
GICC: 0xc4300000 + 0x2000 = 0xc4302000
GICH: 0xc4300000 + 0x4000 = 0xc4304000
GICV: 0xc4300000 + 0x6000 = 0xc4306000
```

With this information, we can provide the following information in *mem_layout-arm-odroidc2.cpp* of the Fiasco.OC kernel:

```
Gic_dist_phys_base   = 0xc4301000,
Gic_cpu_phys_base    = 0xc4302000,
Gic_h_phys_base      = 0xc4304000,
Gic_v_phys_base      = 0xc4306000,
```

7.4 Timer

Timers can be freely programmed and can be used as general counters or interrupt generators. A timer decrements a counter in regular intervals based on a specific clock source, e.g. system clock, until it reaches 0, e.g. simple count-down and stop counter or it loops back up, e.g. periodic timer. The main program and a timer are asynchronous. The program can either poll the timer or program execution is interrupted by a timer interrupt. Polling means checking constantly a status register for timer events, thus being a time consuming task. Instead of checking for a timer event, it is also possible to use a periodic timer. When using a periodic timer, a timer event occurs under a certain condition, which triggers an interrupt. The timer sends a hardware signal to the interrupt controller which suspends the execution of the main program and makes the processor jump to a specific interrupt service routine (ISR), also termed interrupt handler. After handling the interrupt, program flow returns to the main program.[PERTIM20] According to the TRM, the SoC contains 11 general purpose timers and 2 watchdog timers.
The porting-specific information for the timer is located in *timer-arm-generic-odroidc2.cpp*. The file *arch/arm64/boot/dts/amlogic/meson-gx.dtsi* in the Linux kernel provides the following timer specific information for the Odroid-C2:

```
timer {
        compatible = "arm,armv8-timer";
        interrupts = <GIC_PPI 13 ...>, <GIC_PPI 14 ...>,
                     <GIC_PPI 11 ...>, <GIC_PPI 10 ...>;
};
```

IRQ numbers 16-31 are reserved for PPI. Thus, the final IRQ numbers for the different timers are given by adding the respective PPI number to the PPI base number 16:

```
16 + 13 = 29
16 + 14 = 30
16 + 11 = 27
16 + 10 = 26
```

According to [ARM20.3], the arm generic timers have the following IRQ numbers:

Timer	**IRQ**
EL1 Physical Timer	30
EL1 Virtual Timer	27
Non-secure EL2 Physical Timer	26
Non-secure EL2 Virtual Timer	28
EL3 Physical Timer	29
Secure EL2 Physical Timer	20
Secure EL2 Virtual Timer	19

The highlighted rows refer to the IRQ numbers that are used by the reference platforms as well as our implementation for the Odroid-C2. IRQ 29 stands for the physical timer at the firmware exception level (EL3). IRQ 27 is the virtual timer at the OS exception level (EL1) and IRQ 26 is the timer at the hypervisor exception level (EL2). With this reasoning, we set the following information in *timer-arm-generic-odroidc2.cpp*:

```
Generic_timer::Physical: return 29;
Generic_timer::Virtual:  return 27;
Generic_timer::Hyp:      return 26;
```

7.5 Watchdog

A watchdog timer is a feature in microcontrollers that ensures that the system is not stuck in an endless loop for any blocking situation within the code. This is crucial for many systems as calls should eventually return to the main event loop of a program. In case a blocking situation surpasses a specified time interval, the system is classified as unresponsive. In these situations the watchdog recovers the unresponsive system by triggering a forced reboot regardless of the current state of the CPU. According to the TRM, the Odroid-C2 contains 2 watchdog timers.
The file *arch/arm64/boot/dts/amlogic/meson-gx.dtsi* in the Linux Kernel states the following information for the watchdog timer:

```
cbus: bus@c1100000 { ...
        watchdog@98d0 { ... reg = <0x0 0x098d0 0x0 0x10>; ... };
... };
```

With this information, we can calculate the watchdog base address $0xc1100000 + 0x098d0 = 0xc11098d0$. According to the TRM, specific timers are listed with a certain

offset and the register final address can then be calculated with the following formula: 0xC1100000 + offset * 4. For the *WATCHDOG_CNTL* register, the TRM states an offset of 0x2634, thus the watchdog base address is 0xC1100000 + 0x2634 * 4 = 0xc11098d0. This verifies the information from the Linux Kernel. The following line can be added to *mem_layout-arm-odroidc2.cpp*:

```
Watchdog_phys_base   = 0xc11098d0
```

7.6 Challenges along the way

Over the course of the porting task, we ran into many dead ends and faced challenges along the way. This section documents these challenges and outlines our approaches in solving them.

7.6.1 Trying to get any UART output

As our initial porting attempts did not result in any output from UART during the booting process, we concluded that either something is wrong with our U-Boot version or our specified porting information is incomplete. In the end, we realized that both were the case. To get any UART output, we have to use a special U-Boot version that is based on the 2020 master branch from DENX instead of the recommended U-Boot branch from Hardkernel that is based on a 2015 revision of U-Boot. More information on the U-Boot fix was elaborated in earlier sections. To get any UART output, we also had to make further porting adjustments. The first breakthrough was setting the correct base address for UART. We were using 0xc11084c0 of UART0 that is located in the EE domain. As described in earlier sections, we finally came to the conclusion that we need to address UART0-AO that has base address 0xc81004c0. We also noticed that we loaded the *bootstrap_hello.elf* image to the default address 0x01000000 over TFTP. We realized that when using the command bootelf to start the *bootstrap_hello.elf* file, the extraction of the image would overwrite the image, rendering it useless. By taking the file size into account, we loaded the elf image to another location (e.g.0x4000000), such that unpacking wouldn't overwrite the elf file, bootelf is extracting from. This finally gave us some output. Unfortunately, the output was garbled.

7.6.2 Getting readable UART output

UART was printing unreadable characters at first. We realized that the initial UART implementation was using the wrong UART driver. The L4 and kernel directories offer different UART drivers defined in *l4/pkg/drivers-frst/uart/src/* that can be included with header files from *l4/pkg/drivers-frst/uart/include*. As the default UART driver *uart_sh.h* was only printing garbled output, we had to look for a different driver than the one from rcar3 or zynqmp. We chose *uart_16550.h* and included the respective header files in *odroidc2.cc* from the L4Re directory and *uart-arm-odroidc2.cpp* from the Fiasco.OC kernel directory. Furthermore, we had to link the object file with *OBJECTS_LIBUART* +=

uart_16550.o in the *Modules* file. With these changes applied and recompiled, we finally got some readable output:

```
## Starting application at 0x01000000 ...
L4 Bootstrapper
  Build: #152 Thu Aug  6 09:23:00 CEST 2020, 7.5.0
  RAM: 0000000000000000 - 000000007fffffff: 2097152kB
  Total RAM: 2048MB
  Scanning fiasco
  Scanning sigma0
  Scanning moe
"Synchronous Abort" handler, esr 0x96000010
...
```

At that point, UART was printing very slowly, around one character per second. Furthermore, as can be seen in the output above, the boot process still fails during booting with a synchronous abort.

7.6.3 Making UART print faster

The peculiar thing was that when interrupting the boot process with Ctrl+C, UART prints most of the information processed up to this point instantly before falling back to the synchronous abort handler. This means the information must be buffered somewhere and is already available. This led us to the conclusion that the problem must be with the UART implementation of the driver. The UART driver implementation for *uart_16550.h* is located at *4/pkg/drivers-frst/uart/src/uart_16550.cc*. Specifically the *write()* function was of interest to us:

```
int Uart_16550::write(char const *s, unsigned long count) const {
    /* disable uart irqs */
    ...
    /* transmission */
    Poll_timeout_counter cnt(5000000);
    for (i = 0; i < count; i++) {
      cnt.set(5000000);
      while (cnt.test(!(_regs->read<unsigned char>(LSR) & 0x20)));
      _regs->write<unsigned char>(TRB, s[i]);
    }
    /* wait till everything is transmitted */
    cnt.set(5000000);
    while (cnt.test(!(_regs->read<unsigned char>(LSR) & 0x40)));
    _regs->write<unsigned char>(IER, old_ier);
    ...
}
```

According to the comments, first the UART IRQs are disabled, then the characters are transmitted and afterwards, the function is pending until everything was transmitted and the IRQs are reactivated.
To fully understand the function, we needed to take a closer look at UART. UART specific registers are specified in *l4/pkg/drivers-frst/uart/include/uart_16550.h*. The relevant registers are:

```
IER = Interrupt Enable Register
TRB = Transmit/Receive Buffer
THR = Transmitter Holding Register
TSR = Transmit Shift Register
LSR = Line Status Register
```

The bit values of the IER represent interrupts that may take place. In the function, IRQs are disabled by a bitwise AND operation. The TRB is a buffer that contains the data to be transmitted. The data will be pushed into the THR and the TSR shifts the data bit-by-bit onto the serial line (parallel to serial conversion). The data on the serial line is transformed into parallel data again on the receiving end (serial to parallel conversion). The LSR shows the current state of communication[UART]. In the *write()* function, especially bits 5 (= THR empty) and 6 (= THR and TSR empty = no data to be sent in buffer any more = everything was sent) are of interest. The *cnt* variable represents the poll timeout counter and is defined in *l4/pkg/drivers-frst/include/poll_timeout_counter.h*. The *test()* function checks the input expression and decrements a counter variable in case the condition is not fulfilled.
With this information, we can now precisely understand what happens during the UART write operation. With *_regs->read<unsigned char>(LSR) & 0x20*, the function checks whether bit 5 of the LSR is set. If this is the case, this means the THR is empty, i.e. nothing is currently being sent, thus the THR can be filled with new data. The *cnt.test()* evaluates to false instantly and the while loop terminates. If the THR is not empty, this means that there is already data currently located in the THR that first needs to be pushed sequentially onto the serial line. As a result, the counter variable will be decremented each iteration of the while loop, until either the THR is empty or the counter variable reaches the value 0. After stalling, the data is written into the TRB to be transmitted over the serial line. After *count* bytes of *s* was written into the TRB, the for loop terminates. Afterwards, there is another stall that will make sure that all the data is pushed onto the serial line before returning form the *write()* function. The second while loop stalls as long as there is still data in the THR or the TSR, i.e. bit 6 of LSR is not set to 1, or the counter variable is not 0. When all the data is transmitted, the THR as well as the TSR are empty, thus bit 6 of the LSR is 1. This leads to *_regs->read<unsigned char>(LSR) & 0x40* evaluating to true and thus terminating the while loop. Afterwards, the old irq state is restored and the program returns to the caller.
The reason for UART printing slow is the last occurrence of *cnt.set(5000000)* after the for loop. This command is responsible for the pending UART behaviour. When using a value smaller than 5000000, UART prints much faster.

7.6.4 Where the boot process still fails

Currently, the boot process for the FOC/L4Re build fails somewhere in early boot. More information on the booting process is provided by the *startup()* function in *startup.cc* of the L4Re directory:

```
void startup(char const *cmdline) {
        ...
        Boot_modules *mods = plat->modules();
        add_elf_regions(mods->module(kernel_module), Region::Kernel);
        add_elf_regions(mods->module(sigma0_module), Region::Sigma0);
        add_elf_regions(mods->module(roottask_module), Region::Root);
        l4util_mb_info_t *mbi = plat->modules()->construct_mbi(_mod_addr);
        assert(mbi->flags & L4UTIL_MB_MODS);
        ...
}
```

The startup function definitely proceeds until line 7 of the code listing, because UART is already printing *Scanning moe* before faulting with a synchronous abort. In order to see where the booting process fails, we put printing statements behind every expression in this function. According to this strategy, we even had booting processes where line 7 is completed. We never reached printing statements after line 8. The boot process seems to fail at the first assert (line 8) when trying to access *mbi->flags*. Due to the fact that UART prints data that is located in the TRB, some data in the buffer might not be printed if there is an abort. Thus, we can not precisely pinpoint at which line the boot process fails.

7.6.5 Open Questions

Due to time constraints, we were not able to fully realize the optimal outcome for the porting task. Our booting process still faults with a synchronous abort and we can not precisely pinpoint where the booting process exactly fails, which complicates debugging attempts. Furthermore, NIC functionality was not ported during the practical course. There is also still booting behaviour, we do not fully understand. For example, the first characters to be printed are still transmitted in a non-readable format before the message *L4 Bootstrapper* is printed. Furthermore, sometimes we need multiple booting attempts until the expected booting behaviour can be observed. In the other cases, the booting process is interrupted prematuraly by the synchronous abort handler. In these premature booting faults, although the transmitted characters are readable, UART does not seem to transmit the correct characters, e.g. *L4 nf 6* instead of *L4 Bootstrapper*.

8 Porting to Genode

The Genode OS framework is a general purpose abstraction layer on top of different microkernels. It provides a powerful infrastructure for creating many different types of applications, as well as `libc` support. Within Genode several major components, such as the linux network stack, have been ported to L4 based microkernels. This makes Genode an attractive target for implementing higher level functionality.

During this practical course our main focus was porting the Fiasco.OC microkernel into the Genode.OS framework. We focused on getting the previously configured and tested kernel, but also tried around the seL4 microkernel, because this supports the Odroid-C2 natively.

8.1 Fiasco.OC and L4re

After Porting Fiasco.OC and L4re more or less successfully, the configuration has to be included into genode for the genode build system to pick up our carefully crafted configuration, and with that our modifications. We included the `contrib/foc-*` folder into our repo in order to better track our local changes, so it is not necessary to use the `prepare_port` utility to get the code.

In order to prepare genode for an arm_v8a port a build directory has to be created that includes `arm_v8a` in its name. Now we can configure the kernel to use (foc) as well as the target board to use (odroidc2)

The main task of the genode port now was to follow the breadcrumps of the *raspberry pi 3*, which is a similar board but running on an arm_v7 architecture[RASPI20], so the exact code is not comparable.

But following which configuration is done for the *raspberry pi 3* gives a good hint how to configure for the *Odroid-C2*.

In the following description `FOC-BASE` is the directory
`/contrib/foc-91ca3363690c5b9c992a110375242f5d426a6848/src/kernel/foc`

./repos/base-foc/lib/mk/spec/odroidc2 Root of the build system for the odroidc2 platform. These are referencing all following files

./repos/base-foc/config/odroid.kernel Kernel Configuration originally developed in `${FOC-BASE}/kernel/fiasco/build_odroidc2/globalconfig.out`

./repos/base-foc/config/odroidc2.user L4re userland configuration originally developed in `${FOC-BASE}/l4/build_odroidc2/.config`

./repos/base-foc/recipes/src/base-foc-odroidc2 This Directory contains the Fiasco.OC-specific part of Genode. for the odroid

Afterwards Unfortunately the Fiasco.OC kernel never fully booted in our configuration, so the full genode test could never be made. We also had several linking issues due to complications with the genode build system, which we had no time do dive deeper.

In its current state it is complaining about the nonexistence of the following packages.

```
error: package 'libkproxy', not found
error: package 'libloader', not found
error: package 'l4re-util', not found
error: package 'cxx_libc_io', not found
error: package 'libc_be_minimal_log_io', not found
error: package 'libsupc++_minimal', not found
```

8.2 seL4

Besides Fiasco.OC we learned about another microkernel, the seL4. This kernel is also supported by genode, and it does support the Odroid-C2 board natively. The initial approach contained some bugfixes in the seL4 build system, and especially creating a virtual environment for python where the dependencies for the buildsystem are stored.

This is achieved by running

```
sudo apt install libarchive-dev python3-tempita python3-virtualenv
python3 -m virtualenv venv
. ./venv/bin/activate
pip3 install sel4-deps

./tool/ports/prepare_port sel4
```

Afterwards the build can be configured with `BOARD=odroidc2` and `KERNEL=sel4`.

To help genode configure the seL4 correctly this should be done in `repos/base-sel4/lib/mk/spec/ar` containing the following information:

```
PLAT  := odroidc2
CPU   := cortex-a53
BOARD := odroidc2

include $(REP_DIR)/lib/mk/spec/arm/kernel-sel4.inc
```

Unfortunately while digging deeper into the build system of Genode, we found out, that it only supports ARM-v7a 32 Bit architecture for the seL4. Because did not want to invest the time into this sidetrack, we did not further investigate this route.

But we learned a lot about the genode build system and powers and drawbacks of the concepts behind Genode.

9 State of the Port

Our efforts of porting the Fiasco.OC to the Hardkernel *Odroid-C2* platform where not fully successful. The first struggles where getting the correct U-Boot repository as well

as the correct configuration. The solution was to rely on the upstream version, and not a five year old fork.

After having a uboot and tftp setup, we could start working on the Fiasco.OC kernel, and our development gained a lot of traction, unfortunately we did not have enough time to work on all the issues. The biggest overall problem we encountered was, that Genode as well as Fiasco.OC do not seem to be well-tested on the arm-v8a platform, requiring several workarounds for the buildprocess, when compared to the simlpicity of the armv7 platform of the raspberry pi Most of the other platform dependent hardware such as interrupt controller, timers and watchdogs were set up as successful as we could test it.

10 Genode Application Development

An application is a component type for Genode. That is Genode contains different kinds of components, i.e. device drivers, resource multiplexers, protocol stacks, applications and runtime environments [Fes20].
The Genode Architecture is designed to give the ability to add new components independently from other factors. Therefor it is possible to create sophisticated systems out of many different components. The Genode repository contains some components for connectivity or GUI-applications which can be used in a system [Gen20].

10.1 Parallel Genode Application Setup

The goal here is to develop an application for Genode which can then be executed on the Odroid C2. Our development approach is to develop the Genode application in parallel while we are struggling with porting Genode to the OdroidC2 SOC. That is, the used version of Genode was build for the ARM v7a platform instead of the ARM v8a platform the OdroidC2 uses. The main difference for our experienced problems was the architecture difference - ARM v7a is a 32-bit system, ARM v8a a 64-bit system. While working on porting Genode OS and other platforms to the OdroidC2, we learned that many parts of the used systems were implemented in an experimental way for 64-bit architecture and therefor did not work as intended.

10.2 Genode Components

The Genode architecture is based on a component structure, also called building blocks. Individual components are contained within a protection domain and can interact with other components in controlled ways. Components are generally not seen as applications but more as operating-system functionalities. The operating system can be adjusted to the specific need - components can be added / removed as required by the specific setup. This is an advantage for general security and space consumption. As experienced during development, Genode is designed to advocate the development of small components - it minimizes the risk of unwanted leaks and other vulnerabilities [Fes20].
The inter-component communication is necessary for providing functionality of the operating system. With splitting the system into smaller blocks, single blocks might be reused for other functionalities, which also minimizes the code redundancy, but might disregard performance considerations. This implies that it is required to use a dynamic linker for being able to link at run time. This provides a certain flexibility for reusing components [Fes20], [Gen20].
Genode differentiates component types into *Device Drivers*, *Protocol stacks*, *Resource multiplexers*, *Runtime environments* and *applications*. For us, the category *Runtime environments* and *applications* was important [Fes20].

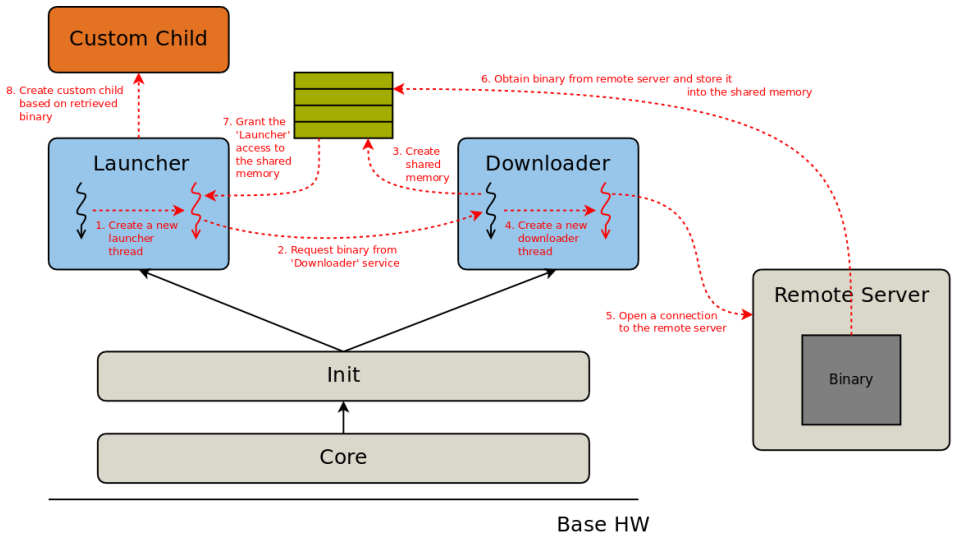


Figure 3: Demonstration of the application system. The components are: *Downloader*, *Launcher* and *Custom Child*. The Launcher starts the Downloader session, waits for its completion, then uses the acquired binary file from the shared memory to create a child session.

10.3 Genode Services and Sessions

For developing the application we had to understand how Genode utilizes components. As the code itself is not well documented, this provided a good source for different challenges we had to face over the time.

Components can have Parent-Child relationships or Server-Client relationships. Components provide a service, such as downloading a file to local storage, which has to be announced to its parent. Without this announcement, the parent cannot use the childs service. To do that, every component is required to create a RPC object, which implements the *root interface*. The announcement is done when initializing the child component. The parent component can then use the announced service or ignore it [Fes20].

Sessions are the counterpart to a service announcements. A session can be created by a client with issuing a session request to its parent. In this scenario the child requests a specific service which is not its child to its parent (server). The parent can then decide if it wants to give the specific capability to the client, i.e. the RPC object. This is called a *session capability* [Fes20].

10.4 Creation of Client and Server Components

In our project, the server is the *Downloader component* which provides a shared-memory and downloader capability. To provide that capability at first the memory has to be allocated and then we attach the allocated memory to the *environment memory region*. The *environment* is an object containing Donwloader specific functionalities. It is attached to the parents *root entry point* [Gen20][Fes20].

```
Genode::Dataspace_capability request_shared_mem()
{
_ds_cap = _env.ram().alloc(SD_SIZE);
_addr = _env.rm().attach(_ds_cap);
return _ds_cap;
}
```

Shared memory capability in the Downloader component.

The *Launcher* acts as client component. It uses the *Downloader Component* for requesting a downloader capability. The Downloader provides that capability and offers shared memory to share with the *Launcher client*.
The Launcher requests a Downloader session component for downloading and acquiring the binary file. It starts the Downloader session, waits for its completion and then acquires the downloaded file in the provided shared memory of the server component. Additionally the Launcher can create a child session component based on the downloaded binary file.
In contrast to the server component, the Launcher component does not provide any service announcements to its parent, a Genode system component. Instead just the Launcher threat is started. Additionally to that, the Launcher provides the ability to create a custom child session, therefor it has to create parent services for the child session.

10.5 Signal Notifications

The *Downloader component* provides three accessible functions for starting the download and requesting / releasing shared memory. The program flow is following to acquire the binary file:

```
Downloader::Connection &_dl;
Genode::Signal_context _sigc { };
Genode::Signal_receiver _sigr { };

_dl.request_shared_memory();
Genode::Signal_context_capability sccap = _sigr.manage(&_sigc);
```

```
_dl.request_binary(sccap);
_sigr.wait_for_signal();
_sigr.dissolve(&_sigc);

/* create custom child
... */

_dl.release_shared_mem();
```

The *Signal* is used to communicate accross different RPC objects. With that it is possible to communicate with the *Downloader session* to get feedback if the *Downloader* has finished loading the binary file to shared memory. After we know the file is stored in the shared memory, we access the shared memory to create the child session.
The *Download Threat* uses a *Signal Transmitter* to indicate its state of completion. The *Signal Transmitter* is created in the
request_binary(Genode::Signal_context_capability sigcc) method and then used to create the *Downloader Threat*. After downloading the file, the transmitter signals the signal handler in the *Downloader Session*. When the *signal.submit()* is received, the Launcher knows that the binary file is fully loaded to the shared memory of the *Downloader Session* and *Launcher Session*.

10.6 Development on remote Ubuntu PC with Visual Studio

With the WSL setup network modules like *tap device* could not be used for creating a DHCP session, since the underlying Windows system does not support that Linux functionality. Johannes created a remote setup which was already used to access the Odroid C2. With the remote setup, we could connect to the Ubuntu Laptop via SSH connection to work on the project directly on an Ubuntu system.
To achive that, we used Visual Studio Code with a module called *Remote SSH*. This module can connect to the remote maching and provides access to the project files with code analyzing features like Intellisense, syntax colorization and more. Visual Studio Code also provides a terminal connected to the remote device, which can be used for working on the remote machine. This setup provided a simple way for working through example code and Genode source code to figure out functionalities.

10.7 Current State of Development

In the current state the application was tested with the QEMU™emulator [20] on Ubuntu 18.04. We experienced difficulties while connecting to the server to acquire the binary file. We assumed that the problem might be somwhere in QEMU or Genode - since the network information gets somehow lost within the system. For example the netmask is overwritten at some point. We tried different solutions such as setting the network settings in the *ex05.run* specific for the downloader component.

With that problem we were not able to connect to the server and download the binary. Therefor no further development for the child session creation was done, since this step depends on the acquired binary file.
Further work needs to be done in the *Launcher component*, a ROM session needs to be created for the child, then the child itself has to be created and terminated after some seconds. For that also the *Custom Child session component* needs to be adjusted to provide the ROM session and transfer the session quotas to the parent service, the *Launcher component*.

List of Figures

References

[20] *QEMU*. 2020.

[ARM20.1] *Privilege and Exception Levels*. `https://developer.arm.com/architectures/learn-the-architecture/exception-model/privilege-and-exception-levels`.

[ARM20.2] *Precise abort vs synchronous abort in armv7*. `https://community.arm.com/developer/ip-products/misc/f/cortex-a-forum/9229/precise-abort-vs-synchronous-abort-in-armv7`.

[ARM20.3] *The processor timers*. `https://developer.arm.com/architectures/learn-the-architecture/generic-timer/the-processor-timers`.

[ARMV8] *ARM Technical Reference Manual ARMv8*. `https://developer.arm.com/documentation/ddi0487/fb`.

[CYPRESS20] *Troubleshooting Guide for Arm Abort Exceptions in Traveo I MCUs - KBA224420*. `https://community.cypress.com/docs/DOC-15354`.

[DTS20] *Device Tree*. `https://elinux.org/Device_Tree_What_It_Is`.

[EMCR20] *Autobooting Linux from U-Boot*. `https://www.emcraft.com/som/vf6/autobooting-linux-from-u-boot`.

[Fes20] N. Feske. *GENODE: Operating System Framework 20.05*. 2020.

[Gen20] Genode Labs. *Genode*. Github, 2020.

[Gro16] S. Groot. *WSL File System Support*. 2016.

[Gro17] S. Groot. *File System Improvements to the Windows Subsystem for Linux*. 2017.

[INTRPT20] *what is the difference between PPI, SPI and SGI interrupts?* `https://stackoverflow.com/questions/27709349/what-is-the-difference-between-ppi-spi-and-sgi-interrupts`.

[NSPAWN20] *systemd-nspawn(1) Linux User's Manual*. June 2020.

[ODR20.2] *Build U-Boot*. `https://wiki.odroid.com/odroid-c2/software/building_u-boot`.

[OSDEV20] *Target Triplet*. https://wiki.osdev.org/Target_Triplet.

[PERTIM20] *Introduction to Microcontroller Timers: Periodic Timers*. https://www.allaboutcircuits.com/technical-articles/introduction-to-microcontroller-timers-periodic-timers/.

[RAMDISK20] *Initial Ramdisk*. https://en.wikipedia.org/wiki/Initial_ramdisk.

[RASPI20] *Raspberry Pi Product Brief*. https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[S905.20] *Amlogic S905 datasheet*. https://dn.odroid.com/S905/DataSheet/S905_Public_Datasheet_V1.1.4.pdf.

[SO20.1] *Image vs zImage vs uImage*. https://stackoverflow.com/questions/22322304/image-vs-zimage-vs-uimage.

[SO20.2] *zImage, rootfs*. https://unix.stackexchange.com/questions/295131/what-is-zimage-rootfs.

[SUNXI] *Frequently asked questions about the sunxi clock system*. https://www.kernel.org/doc/html/latest/arm/sunxi/clocks.html.

[SYNC] *sync - Unix, Linux Command*. https://www.tutorialspoint.com/unix_commands/sync.htm.

[TTYLX20] *What is the difference between ttys0, ttyUSB0 and ttyAMA0 in Linux?* https://unix.stackexchange.com/questions/307390/what-is-the-difference-between-ttys0-ttyusb0-and-ttyama0-in-linux.

[UART] *Serial UART information*. https://www.lammertbies.nl/comm/info/serial-uart.