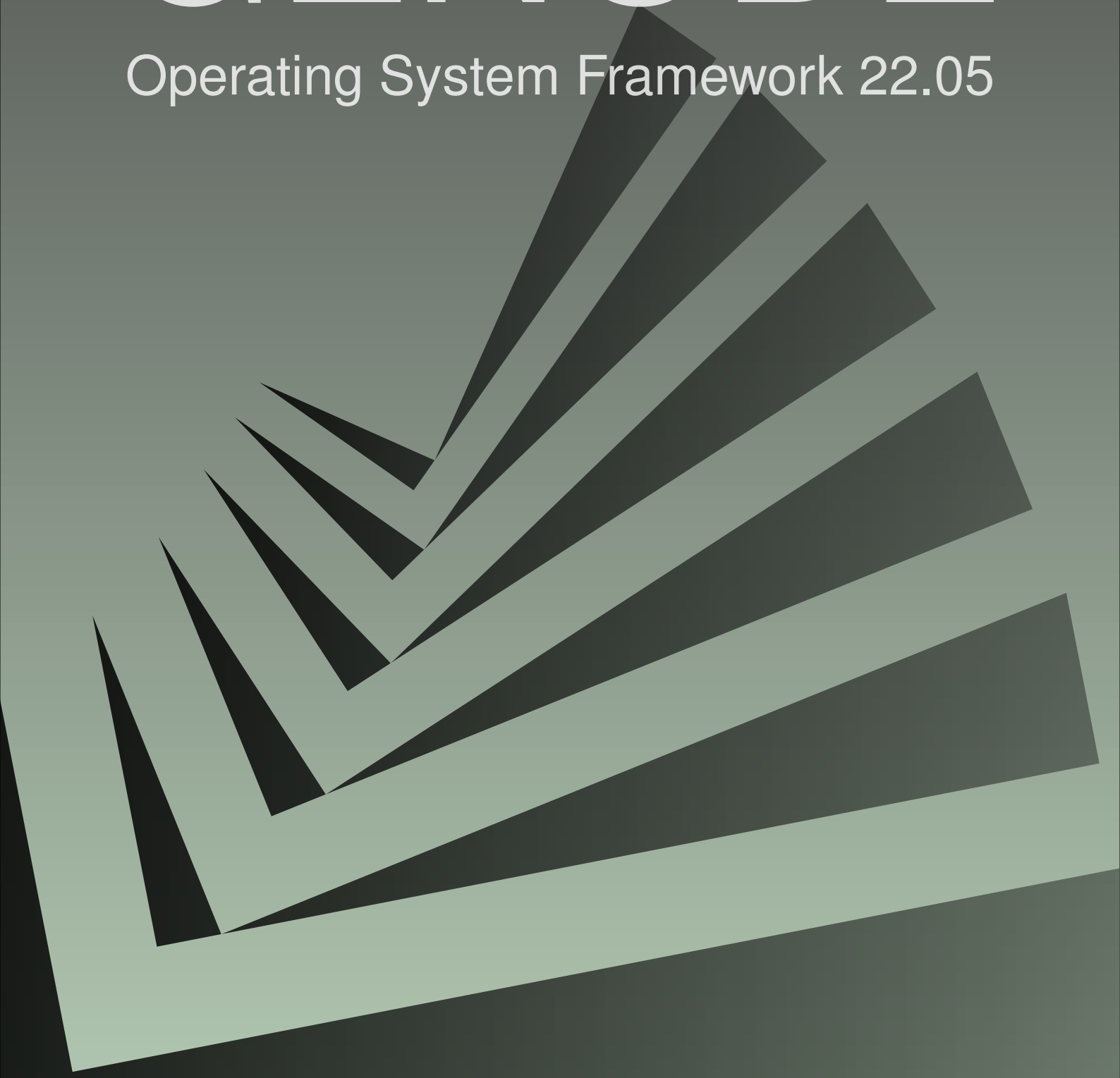
GENODE

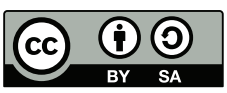
Operating System Framework 22.05



Platforms

Norman Feske

Contents



1 Introduction

This document complements the Genode Foundations book with low-level hardware-related topics. It is primarily intended for integrators and developers of device drivers. Before studying the Genode Platforms material, it is highly recommended to give the Genode Foundations book a read. The book can be downloaded at https://genode.org.

In this first edition, the document features a practical guide for the steps needed to bring Genode to a new ARM SoC. The content is based on the ongoing Pine Fun article series at Genodians.org[1]. Note that the document is not set in stone. We plan to continuously extend it with further practical topics as we go.

[1] https://genodians.org

2 Porting Genode to a new SoC

We get repeatedly asked about the principle steps and costs needed to enable Genode - and in particular Sculpt OS [1] - for various ARM-based hardware platforms. The variety of SoCs is too great to give a general answer. However, drawing from our experience with the porting Genode to several ARM-based platforms such as NXP's i.MX8, this chapter provides a practical guide for the steps of such a porting endeavour.

The guide is based on an article series at https://genodians.org. It is written in an informal style from the perspective of a developer carrying out the work, taking a specific board - namely the Pine-A64-LTS single board computer - as a playground. The code discussed throughout this chapter is available at the following public Git repository.

Git repository of the Allwinner board support

https://github.com/nfeske/genode-allwinner

The guide is not carved in stone. It will be progressively enhanced with further information - e. g., details about various classes of drivers - over time. Should you happen find important topics missing or spot mistakes or have suggestions for improving the material, please don't hesitate to send your feedback to *norman.feske@genode-labs.com*.

Goals Our goal would be to get the bare-bones Sculpt system up and running on an ARM SoC not yet supported by Genode. This bare-bones Sculpt system entails

- The principal ability for the user to interact with the system via a graphical user interface,
- Support for installing and deploying the existing arsenal of Genode components from regular packages,
- The ability to store information persistently on the device, and
- Network connectivity.

Thanks to Sculpt's built-in ability to integrate 3rd-party components - including functionality that is traditionally attributed to the core of the operating system - into the system in the form of packages, this bare-bones system enables a great variety of usage scenarios.

[1] https://genode.org/download/sculpt

Non-goals That said, the following features remain beyond the scope of this document because they are either too vendor-specific to be described in a general fashion or can be realized in the form of supplemental components.

- Hardware-accelerated graphics,
- Audio,
- Power management,
- Mobile data communication,
- Secure boot.

Working steps The work of enabling Genode for a new SoC requires the following steps in the described order. To give an indicator of the effort to be expected, each step is accompanied with a rough estimation.

1. Preparing the development testbed

   Before the actual development work can start, a few preparations are needed or at least recommended.

   One of our team members typically spends up to **one month** for this step.

   - Building and running a working Linux-based OS on the target board as reference, following the instructions of the vendor
   - Exploration and configuration of the target's boot mechanism
   - Creation of a test-control loop for triggering the booting the target board via the run tool, serving the boot image over the local network, and obtaining the log output.
   - Familiarization with the available board and SoC-vendor documentation and the Vendor-specific subsystems in the vendor's Linux kernel
   - Studying the device tree, correlating it with information gathered from the documentation.

2. Code skeleton for a new SoC

   Given the impressions gathered during the preparatory step, we take one of the SoCs that are already supported by Genode as reference. One should select the SoC with the most similarities such as the same ARM core revision or the same interrupt controller. The goal of this step is an almost empty skeleton code of Genode that gives us a little life sign when booted on the real hardware.

   It does not take a seasoned Genode developer longer than **two weeks** to complete this step. However, for a developer with no prior experience with Genode's code

base, an **additional** effort of **two weeks** for the required familiarization should be planned for.

- Mirroring the files of another SoC but with empty bodies, (describing roles of the individual files)
- Creating a bare-bone base-hw kernel ELF image
- Booting the custom image on the target hardware
- Serial output driver

3. Basic kernel functionality

   The goal of this step is getting the most basic Genode system scenario to run on the new SoC. This scenario comprises three components, namely the Genode core component (including the kernel), the init component, and a test program that produces some log output.

   On this way, one has to overcome the challenges of initializing the kernel, enabling the MMU, and exercising the kernel's IPC and context-switching mechanism. Assuming that the new SoC has the same architecture revision as the ones already supported by Genode, this step should take no longer than **two weeks**.

   - Enabling the MMU
   - Enabling caches
   - Memory layout parameters
   - Entering and returning from the user land (IPC, context switches)
   - Running Genode's log scenario

4. Support for user-level device drivers

   With the principal ability of running multiple user-level components, it is time to enable preemptive scheduling and the kernel mechanisms needed by user-level device drivers. Assuming the new SoC uses standard ARM building blocks like the core-local timer and the GIC interrupt controller as readily supported by Genode, this step does not entail much risks and should be completed within **a week**.

   However, should the SoC deviate from the beaten track of standard ARM building blocks, e. g., using a custom interrupt controller, the step may additionally require the development of an in-kernel driver for such a device. Genode provides several existing drivers that can be taken as a blue print. Depending of the quirkiness of the device, the development can take one or two weeks. Fortunately, vendor-specific timers and interrupt controllers are largely a problem of the past.

   - Enabling the in-kernel interrupt controller driver
   - Enabling in-kernel timer driver

- Definition of I/O resources
- IOMUX configuration (board-specific)

Once the principal support for user-level device drivers is in place, the development work can be tackled by multiple developers in parallel.

5. Network driver

   We usually plan to spend about **one month** for enabling a network driver for Genode. Depending on the complexity of the network controller, the driver may be ported from the Linux kernel, from the U-Boot boot loader, or written from scratch.

6. SD-card driver

   For driving SD-cards, we usually extend Genode's custom SD-card driver with SoC-specific support, which takes usually **two weeks**. One should be prepared for device-specific peculiarities though. In some cases, in the presence of flaky hardware, it took us up to 3 weeks more to reach a stable and performant state.

7. Framebuffer driver

   In the past, we used to develop framebuffer drivers from scratch. But nowadays, we prefer to reuse the vendor-provided driver code from the Linux kernel to attain feature parity with Linux. That said, depending on the driver, such porting work still requires substantial manual labour because the driver often does not only drive one device but multiple (such as power-gating via additional I2C-connected controllers, or a dedicated HDMI chip). As an indicator for the expected effort, the i.MX framebuffer driver took us **two months** to bring to life.

8. USB host-controller driver

   Genode's USB host-controller driver is based on the Linux USB driver. Adding supplemental support for a new SoC should generally possible within **one month**. With the USB host-controller driver in place, the actual USB device drivers (e. g., for HID and storage) should work out of the box.

   As a note of caution, in rare cases, in particular for the Raspberry Pi, the USB host controller driver can become an almost infinite time sink though.

9. Multi-processor support

   Real-world workloads demand multi-processor support. In theory, this should generally be covered well by Genode's ARM support as long as the SoC stays close to ARM's reference design. However, the bring-up of secondary CPUs, inter-processor interrupts, and the maintenance of TLB/cache coherence still poses risks because those topics may involve upcalls to vendor-specific firmware or may depend on the unexpected vendor-specific boot-time configuration (like

the surprise of one CPU core left configured with a different byte order). To stay on the safe side, one should plan **one month** for the potential troubleshooting around these areas.

10. Sculpt OS integration

    With the four peripheral drivers in place, Sculpt's demands on the platform's feature set is satisfied. The remaining task is the integration of those drivers into Sculpt, which should be doable in no more than **two weeks**.

    - Drivers subsystem definition
    - Sculpt-manager tweaks
    - Configuration

Summary Based on the steps outlined above, the effort seems to be modest but - given a healthy dose of enthusiasm - quite doable for an individual or a small team. The biggest risk is the incomplete or lacking documentation for most ARM SoCs.

Granted, such a bare-bone system is still a far cry from a sophisticated product like a smart phone, which features plenty of additional peripheral devices, an aggressive power-management regime, GPU-accelerated rendering, or Bluetooth. But once a bare-bones Sculpt system is ready to run, further device drivers can be developed as regular components independent from each other, which is the beauty of a component-based operating system like Sculpt OS.

2.1 Preparatory steps

After getting a rough overview of undertaking the port of Sculpt OS to another SoC in the previous section, let us take a closer look at the first step - taking technical and non-technical preparations.

For the preparatory work, I recommend taking one month of time. This may sound excessive but there are good reasons. First, Genode's tooling deviates from the beaten tracks known from commodity operating systems. In particular Genode's run tool is quite unique and powerful. But it comes at the price of a learning curve. The learning should not be done as a side activity but requires the focus of the developer. Second, the initial steps of enabling a new hardware tend to be fiddly. Especially when it comes to compiling and testing out a vendor-customized boot loader and Linux kernel from source, this can become a walk on muddy ground. Without patience or with time pressure, it can get messy and exhausting. Third, contemplating about non-technical preparatory aspects like licensing deserves some nights to sleep over it.

2.1.1 Licensing considerations

I see your raised eyebrows. Why bother with software licensing at this point? To pursue the upcoming steps with as little friction as possible, make up your mind about **your objectives** behind pursuing the porting work. The licensing of your code should follow from that. From the chosen license, in turn, follows the way of how to interact with the community. Let me illustrate this point with three example scenarios:

No strings attached

Open-source driver code authored by hardware vendors is often published under a permissive license to make the code broadly usable across projects with different open-source and proprietary licenses. Even for code contributed to the GPL-licensed Linux kernel, some vendors like Intel provide their contributions under the terms of the permissive MIT or BSD licenses, and thereby allow anyone to incorporate such code into other operating systems without licensing constraints. Usually such code is a clean-room implementation developed in-house at the vendor without incorporating 3rd-party code. This approach is preferable whenever the objective is the **highest possible adoption** of the code.

Submitting code upstream to the Genode project

A second possible objective may be the integration of your work upstream into the official Genode project to make the new SoC platform straight-forward to use for the Genode community and to benefit from the **ongoing maintenance** of the code **by Genode Labs**. However, with this ambition in mind, you need to ensure that you and your employer agree with the process of contributing[1] and in

[1] https://genode.org/community/contributions

particular with the terms of the Genode contributor's agreement (PDF[1]), which grants Genode Labs the right to offer Genode - including your code - under both open-source and commercial licensing terms.

Pursuing a dual-licensing business

At the other extreme, your objective may be offering the results of your work as a **commercial product**, following a dual-licensing business model. In this case, you may consider publishing the code under the most restrictive copyleft license possible, along with the option for a commercial license. Or you may even go as far as considering the Genode Component Public License[2] (CPL). This route should be considered only when planning a **long-term commitment** in actively productising and supporting your code. Note that the GCPL is no win for the open-source community beyond Genode.

The path taken has far-reaching ramifications. The ability to incorporate 3rd-party code into your work. The visibility of your work within the Genode community. The selection of a suitable place for hosting your code. Community spirit. Or the viability of contributions by others to your code.

The decision may be taken for different components individually. For example, when taking the Linux USB stack as the basis for a USB host-controller driver component, this component naturally inherits Linux' GPLv2 license. At the same time, your custom in-kernel timer driver might fit best into the upstream Genode project.

In our experience, taking and openly communicating licensing decisions up front before starting actual development work reduces possible friction - especially if a legal department is involved - and avoids wrong expectations.

2.1.2 Selecting a suitable SoC

The question of which particular SoC to select as the basis for your work is of course closely related with the same objectives as discussed above. You may consider the following points:

- Costs of the chip and the devices featuring the chip. E.g., if you primarily intend to accommodate hobbyists, a low-end device might be preferable. But there are other arguments:
- Availability of accessible hardware featuring the SoC. Many SoCs are available only in large volumes and thereby end up in consumer devices only. More often than not, such consumer devices are completely locked down, rendering the attempt to install a custom operating system moot.

[1] https://genode.org/community/gca.pdf
[2] https://genode.org/documentation/articles/component_public_license

With *accessible* hardware, I'm also referring to the availability of development boards that mirror the architecture of a consumer device but with additional connectors for obtaining serial output, network connectivity, and possibly JTAG.

- Availability and quality of technical documentation. Even for many SoCs popular in the Linux community - think of the Raspberry Pi devices - public documentation is sparse or of questionable quality. If you find a "reference manual" of only a few hundred pages online, possibly imprinted with the term "CONFIDENTIAL", it's probably better to stay away from this chip. A modern SoC has usually more than 4000 pages of documentation. When browsing through it, look out for prose and architectural diagrams. Some "reference manuals" are merely disguised register listings, which are not very insightful.

- Support by the official Linux kernel. Even though most ARM devices run Linux, many vendors do not even attempt to contribute vendor-specific code upstream to the Linux project. Should the official Linux kernel features support for a particular SoC, this is a good sign for the maturity of the open-source drivers. In contrary, if only a certain whacky vendor kernel is known to work well with the SoC, it's probably best to shy away.

- Presence of hardware-based I/O protection (System-MMU). To fully leverage the advantages of Genode's architecture, the sandboxing of device drivers is important. Otherwise, all device drivers must be considered trusted.

  When we originally embraced the i.MX8M SoC, we silently assumed that every modern 64-bit SoC should feature a System-MMU in our modern times. We eventually learned that this is actually not the case for the i.MX8M.

  If different variants of one SoC with and without System-MMU are available, make sure to pick the variant that includes this feature.

2.1.3 Start by taking the known-good path

Even though you may be eager with bringing Genode to the new device, let us first exercise the device with its known-to work software stack.

1. Usually, development boards come with a Linux-based system pre-installed. Try it out. Test the functioning of all hardware connectors that are important to you.

2. Chase down the source code of the exact Linux kernel that is pre-installed on your board. In most cases, this so-called *vendor kernel* is a customized version of Linux, with the source code provided at a vendor-specific place. Download it. Follow the vendor-provided instructions to build it from source. Boot your custom built Linux kernel on your device.

This kernel will serve us as a working reference later. It allows us to cross-correlate problems between Genode and Linux, obtain traces of Linux device drivers, or to get hold of system-register states initialized by the Linux kernel to a working state.

3. Study the device tree of the working Linux kernel and correlate this information with the documentation. This helps to form a mental picture of the hardware and to identify possible risks (indicated by your level of confusion) early on.

   ...slowly leaving the known-good path...

4. Now that you are familiar with the vendor kernel, let's cross fingers and hope that the vanilla Linux kernel works just as well. Download the vanilla Linux kernel and look out for the support for your SoC. In the worst case, you won't find any. In the best case, the vanilla kernel works out of the box. In case the vanilla kernel works well, better use this one a reference for your further work.

2.1.4 Setting up an efficient development workflow

For the few test drives taken until this point, juggling SD-cards is probably fine. But down the road, you will need to boot your device with custom system image hundreds of times. Take the time for setting up a convenient test-control loop for your device to make this work enjoyable.

Explore Genode's run tool Read Section 5.4 "System integration and automated testing" of the Genode Foundations book as found at https://genode.org.

Try out the various options with an already supported platform. Browse the files at tool/run/[1] to learn about the various backend modules and options. E.g., look at tool/run/image/uboot[2] to demystify the creation of uImage files by Genode.

Run and test the U-Boot loader on your device U-Boot is the de-facto standard of booting embedded ARM boards today. We primarily use U-Boot for its ability to fetch a system image over the network. There is a good chance that your board comes equipped with U-Boot already. If not, investigate the option to chain-load U-Boot from your board's boot loader.

Once you got U-Boot to work, continue with reproducing the U-Boot binary from source. This may become handy for investigating device-driver issues later on (e. g., taking U-Boot's IOMUX or power or clock configuration as reference, peeking device states at boot time). Consider extending Genode's tool/create_uboot[3] utility, thereby

[1] https://github.com/genodelabs/genode/tree/master/tool/run
[2] https://github.com/genodelabs/genode/tree/master/tool/run/image/uboot
[3] https://github.com/genodelabs/genode/blob/master/tool/create_uboot

documenting the steps for reproducing the U-Boot version for your particular board from source.

Create a working test-control loop The goal of this step is to reach a state where you can type only one command like following from the Genode build directory to trigger a complete build-test cycle.

```
make run/log KERNEL=hw BOARD=<your-board>
```

The build-test cycle entails:

1. Compiling the source code of Genode components,
2. Applying a system configuration,
3. Assembling a system image,
4. Making the system image available over TFTP,
5. Power-cycling the board,
6. Letting the board fetch the system image and start it, and
7. Getting the serial output of the board right in your terminal.

To reach this level of convenience, the following topics must be addressed:

Network boot

- Set up TFTP server on you development machine
- Test your TFTP server locally from your development machine
- Configure DHCP server in your network to direct the boot loader of your development board to the TFTP server on your development machine

Let the run tool obtain the serial output from your board

Take a look at the various options of run tool at tool/run/log[1].

Network-controlled reset / power switch

As the icing on the cake, consider powering your board via a network-controlled power socket as described in [2].

[1] https://github.com/genodelabs/genode/blob/master/tool/run/log
[2] https://genodians.org/chelmuth/2019-03-13-powerplug

More options can be found at at *tool/run/power_off* [1] and *tool/run/power_on* [2].

For further inspiration, you may also enjoy the article [3].

[1] https://github.com/genodelabs/genode/blob/master/tool/run/power_off
[2] https://github.com/genodelabs/genode/blob/master/tool/run/power_on
[3] https://genodians.org/tomga/2019-08-13-rpi-automation

2.2 Getting acquainted with the target platform

The undertaking of bringing Genode - and Sculpt OS in particular - to a new ARM SoC comes with a great deal of uncertainties, namely the inner functioning of overly complex hardware, picking appropriate tools and methodologies, taking informed decisions about porting versus developing drivers, and relating all this to Genode.

Combined, these uncertainties pose a huge barrier. At Genode Labs, we have conquered this barrier a few times in the past, e. g., for supporting the NXP i.MX8 SoC. However, the porting of Genode to new hardware should not be left as an activity exclusive to Genode Labs. In order to assist developers outside of Genode's inner circle with joining the fun, we'd like to share what we know. This sharing should have the form of profound documentation that serves as a guide and removes points of friction as much as possible.

To deliver substance, I figured that I should not merely talk the talk by speaking from past experience, but also walk the walk again while writing down my practical steps as I go. So I went forward looking around for tasty hardware, when Pine64[1] caught my eyes.

Why Pine64? I got excited about Pine64 for several reasons.

First, devices in the form factors of the PinePhone and the A64 development boards are readily available at affordable prices. The Pine64 website carries a very positive message, highlighting community, openness, sustainability, transparency, no marketing nonsense.

Second, the products are designed for hackability. This is evidenced by the vibrant developer community, mainline Linux kernel support, and the availability of literally more than a dozen Linux distributions. One can boot the PinePhone directly from SD-card. How cool is that!

Third, the used Allwinner SoC - introduced as early as 2015 - is rather aged. In contrast to bleeding-edge hardware, I would not need to explore unconquered territory. Others have hopefully discovered most pitfalls before me. The SoC seems to strike a nice balance of modern features (64 bit, multi core, virtualization) with modest complexity. The performance of the SoC is notably at the lower end of the smartphone product category. From the perspective of an operating-systems developer, I don't see this as a con but more as a welcome challenge. Will Genode be able to shine on such a constrained device? Let's find out!

The only downside of the SoC worth mentioning is the lack of an IO-MMU as protection mechanism against rampant I/O devices or drivers. So the sandboxing of device drivers can never be water-tight.

[1] https://www.pine64.org/

2.2.1 Getting a first impression

We ordered a Pine64-LTS[1] board, a PinePhone[2], and a serial cable[3] for the PinePhone directly from the online store. For some kind of safety reason, the phone had to be ordered separately. In hindsight, we better had ordered a power supply for the Pine64-LTS board as well. We skipped it as we already have kilograms of AC power supplies of other boards at hand. However, it turned out that kilograms of power supplies with 5mm connectors are of little use when the board features a less mainstream 3.5mm connector. Such details matter sometimes.

For getting our hands dirty with technical work, we will have to leave the PinePhone alone for a while and turn our attention to the **Pine-A64-LTS** board. The Pine64 wiki[4] provides the perfect starting point.

Booting an officially supported GNU/Linux image The wiki lists numerous ready-to-use Linux distributions[5]. I went for Armbian[6]. Just a few minutes later, after downloading the disk image from https://dl.armbian.com/pine64so/Buster_current, writing the image to an SD card, connecting an HDMI display and a USB keyboard, and booting the board with the SD card inserted, I was greeted with Armbian login, allowing me to login as root user.

At this point, I'm most interested in getting a first overview of the hardware. The following information are insightful:

```
root@pine64so:/# cat /proc/cpuinfo
...
root@pine64so:/# cat /proc/meminfo
```

Well, that is not too surprising. It's more like a ritual.

```
root@pine64so:/# dmesg | less
```

The kernel boot log is quite chatty. The following lines caught my eyes.

[1] https://pine64.com/product-category/pinephone/
[2] https://pine64.com/product-category/pinephone/
[3] https://pine64.com/product/pinebook-pinephone-pinetab-serial-console/
[4] https://wiki.pine64.org/index.php/PINE_A64-LTS/SOPine_Main_Page
[5] https://wiki.pine64.org/wiki/SOPINE_Software_Release
[6] https://www.armbian.com

```
[    2.228675] sun4i-drm display-engine: bound 1100000.mixer...
[    2.230477] sun4i-drm display-engine: bound 1200000.mixer...
[    2.231001] sun4i-drm display-engine: No panel or bridge found...
[    2.231018] sun4i-drm display-engine: bound 1c0c000.lcd-controller...
[    2.231227] sun4i-drm display-engine: bound 1c0d000.lcd-controller...
[    2.231293] sun8i-dw-hdmi 1ee0000.hdmi: Couldn't get regulator
[    2.231734] sun4i-drm display-engine: Couldn't bind all pipelines...
```

...once we get to graphics, we have to grep the Linux kernel for "sun4i-drm" and "sun8i-dw-hdmi". Whatever sun4i and sun8i means. Does "dw" stands for Designware? I shudder for a moment...

```
[    2.250163] 1c28000.serial: ttyS0 at MMIO 0x1c28000 (irq = 31,...
[    2.250239] printk: console [ttyS0] enabled
[    2.250893] sun50i-a64-pinctrl 1c20800.pinctrl: supply vcc-pg...
[    2.251327] 1c28400.serial: ttyS1 at MMIO 0x1c28400 (irq = 32,...
[    2.251471] serial serial0: tty port ttyS1 registered
```

...the Linux kernel uses the serial controller at 0x1c28000 by default. That will be the first device we need a driver for. Never heard of a "16550A" device though...

```
[    2.277178] ehci-platform 1c1b000.usb: EHCI Host Controller
[    2.277210] ehci-platform 1c1b000.usb: new USB bus registered,...
[    2.277359] ehci-platform 1c1b000.usb: irq 22, io mem 0x01c1b000
[    2.289613] ehci-platform 1c1b000.usb: USB 2.0 started, EHCI 1.00
...
[    2.291208] ohci-platform 1c1b400.usb: Generic Platform OHCI controller
[    2.291228] ohci-platform 1c1b400.usb: new USB bus registered,...
[    2.291342] ohci-platform 1c1b400.usb: irq 23, io mem 0x01c1b400
```

...an OHCI USB controller, I get a little blast from the past...

```
[    2.384988] sunxi-mmc 1c0f000.mmc: initialized,...
[    2.410167] sunxi-mmc 1c10000.mmc: initialized,...
[    2.422925] mmc0: Problem switching card into high-speed mode!
[    2.423025] mmc0: new SDHC card at address 0001
```

...two multi-media card (MMC) devices, apparently driven by an Allwinner-specific controller. "Problem switching card into high-speed mode!". MMC and problem are almost synonymous. Allwinner will not positively surprise us...

```
[    3.412571] dwmac-sun8i 1c30000.ethernet: IRQ eth_wake_irq not found
```

...the good news is that there is a dedicated Ethernet controller, not merely a USB-network device. The bad news is that the controller is an IP core purchased from Designware. After the deep scars I got from USB on the Raspberry Pi, I was hoping not to touch anything with "dw" in its name again...

```
[    9.189128] Call trace:
[    9.191219]  ktime_get_update_offsets_now+0x5c/0x100
[    9.193340]  hrtimer_interrupt+0xa0/0x2f0
[    9.195466]  sun50i_a64_read_cntpct_el0+0x30/0x38
[    9.197542]  arch_counter_read+0x18/0x28
[    9.199712]  arch_timer_handler_phys+0x34/0x48
[    9.201813]  handle_percpu_devid_irq+0x84/0x148
[    9.203971]  ktime_get_update_offsets_now+0x5c/0x100
[    9.206022]  hrtimer_interrupt+0xa0/0x2f0
[    9.208071]  generic_handle_irq+0x30/0x48
[    9.210150]  __handle_domain_irq+0x64/0xc0
... many more lines ...
```

...a Linux kernel thread died during boot. The "sun50i" symbol hints at an Allwinner-related driver issue. The kernel marches on nevertheless...

```
[    9.703995] lima 1c40000.gpu: gp - mali400 version major 1 minor 1
...
```

...it's really nice to have a GPU without the need for any proprietary blobs, thanks to the reverse-engineering efforts by the Lima project.

The kernel log is not the only place revealing information about the hardware.

```
root@pine64so:/# cat /proc/iomem

01000000-0100ffff : 1000000.clock clock@0
01100000-011fffff : 1100000.mixer mixer@100000
01200000-012fffff : 1200000.mixer mixer@200000
...
...
40000000-bdffffff : System RAM
```

Here, we get a complete view of the physical-memory layout, including the locations of all memory-mapped devices as well as the actual RAM. The (almost) 2 GiB of physical memory does not start at 0 but rather at 0x40000000.

```
root@pine64so:/# cat /proc/interrupts
```

Here, we see how the relationship between devices, interrupt numbers, and CPUs (interrupt routing) as configured by the Linux kernel.

Another point of interest is the device tree that can be found at */proc/device-tree*, which is actually a symbolic link to */sys/firmware/devicetree/base*.

At this point, it is too early to digest all this information. Let's save it for later. The easiest way is storing data on a USB stick.

1. When plugging in a USB stick to the second USB port, the kernel's `dmesg` output tells us that it is detected as */dev/sdb* as well as the partitions, e. g., */dev/sdb1* for the first partition.

2. Knowing the device name of the partition, we can mount its file system at */mnt* via `mount /dev/sdb1 /mnt`.

3. Now we can copy any files interest to `/mnt/`.

As an additional function test, one can quickly give the **network interface** a try. Once when plugging in a network cable to our local network, the LED on the network PHY starts blinking happily, and `ifconfig` reveals that the board got an IP address from our local DHCP server. A quick `wget https://genode.org` works just as expected.

Serial line Knowing that the board is fully functional when running a Linux-based OS, we have to work towards using the board as an embedded development target. Textual output over **serial** is the most important prerequisite for that. The times when development boards featured 9-pin D-SUB connectors is long past. Nowadays, we need to look out for the right pins on one of the board's expansion sockets. The board has several of them. So now is a good time to get acquainted with the board's schematics[1].

The schematics hint at several serial devices (UART). E.g., UART1 at the SDIO WIFI + BT pin header. The go-to solution is not obvious. Fortunately, a little web search later, we land on a nice wiki page describing the UART on Pine64[2]. In particular, we learn "Better always use UART0 on the EXP connector nearby, accessible on pins 7 (TXD), 8 (RXD), 9 (GND)."

Everyone should have a few TTL-232R-RPi cables at hand. If you don't, hurry up and order some. Pay attention to signal level. In our case, the board needs a 3.3V cable. All we need is cross-connecting TX to RX, RX to TX, and ground to ground.

On Linux-based development machines, we usually use picocom[3] as serial terminal program. When connecting the USB cable, the Linux kernel's `dmesg` output tells us about the new device */dev/ttyUSB0*, which we can readily access with picocom.

[1] https://files.pine64.org/doc/SOPINE-A64/PINE%20A64-TLS-20180130.pdf
[2] https://linux-sunxi.org/Pine64#Serial_port_.2F_UART
[3] https://www.mankier.com/1/picocom

```
picocom --baud 115200 /dev/ttyUSB0
```

When pressing enter, we are greeted with the login of Armbian.

For the next steps, display and keyboard are no longer needed. All we need is the serial line.

JTAG I'm hopeful that serial output will suffice for most debugging work. However, in desperate situations like when facing cache-coherency issues, a JTAG debugger like Lauterbach or Flyswatter can really save the day (or the week). So when encountering a new board, we always look out for JTAG debugging pins. If present, we get the cozy feeling of having this option available as a last resort.

In the case of the Pine64, we must live without this cozy feeling. While searching the forum https://forum.pine64.org, I learned that the SoC is indeed equipped with JTAG pins but the wiring of the Pine board does not make them accessible. Apparently, there is too little interest in JTAG by the community at large, which is perfectly understandable. Most users don't mess around at the low level where JTAG becomes the tool of choice.

2.2.2 The U-Boot boot loader

U-Boot[1] is widely regarded as *the* canonical boot loader for ARM platforms, and we Genode developers agree. The primary reason for our high opinion is U-Boot's ability to fetch boot images over the network from a TFTP server, which is fundamental to our work flows.

The secondary reason is that U-Boot brings the hardware into a state that is convenient for the booted operating system. For example, since U-Boot prints messages over serial, it needs to initialize the serial controller correctly, fiddly stuff like setting up the baud rate or powering the USB FUE. With those preparations done by the boot loader, Genode's drivers can conveniently skip those steps and still work nicely.

The third great benefit of U-Boot to us is the arsenal of drivers supported by the project. Granted, we don't actually use most of those drivers in practice. But others are using them. So the drivers work reliably, are well maintained, and are usually much less complex compared to drivers found in the Linux kernel. This makes the drivers a very useful reference while developing drivers for Genode.

Since Armbian uses U-Boot, we can in principle keep using it. During the boot, one can press <space> at the serial terminal to intercept the automated boot. This brings us to the interactive U-Boot prompt.

[1] https://www.denx.de/wiki/U-Boot

Building U-Boot from source Building the boot loader from source is not just an affair of honor, it also fosters our understanding and our full control over the boot process. The ability to control the boot loader is empowering and can serve as an experimentation ground. The steps for building U-Boot manually for Allwinner-based devices are described in the excellent documentation[1].

For reference, here are the steps I took.

1. Cloning the git repository and checking a recent release branch:

```
$ git clone git://git.denx.de/u-boot.git
$ cd u-boot
u-boot$ git checkout -b v2020.10 v2020.10
```

2. Looking out for a suitable default configuration for the Pine64-LTS board, guessing it would have something like "pine" in the name:

```
u-boot$ find configs/ | grep -i pine
configs/pinebook-pro-rk3399_defconfig
configs/sopine_baseboard_defconfig
configs/pine64_plus_defconfig
configs/pine64-lts_defconfig
configs/pinebook_defconfig
configs/pine_h64_defconfig
```

   Well, *pine64-lts_defconfig* sounds like I'm lucky for the Pine64 board. But the PinePhone is notably absent. A look at https://linux-sunxi.org/PinePhone clarifies the situation: "As we currently do not have any specific U-Boot config for this device, Use the pine64-lts_defconfig build target temporarily as a hack." That's fine by me.

3. Building the ARM Trusted Firmware

   The ARM Trusted Firmware is the effort to unify low-level firmware interfaces - think of the bring-up secondary CPU cores - across SoC vendors. A recent article[2] by Stefan Kalkowski goes into more detail.

   The building steps described at linux-sunxi.org are easy to follow. For us, the build output is quite instructive for guiding our attention.

[1] https://linux-sunxi.org/Mainline_U-Boot
[2] http://genodians.org/skalk/2020-02-18-armv8-smp

```
$ make CROSS_COMPILE=aarch64-linux-gnu- PLAT=sun50i_a64 DEBUG=1 bl31
...
CC      drivers/allwinner/axp/axp803.c
CC      drivers/allwinner/axp/common.c
CC      drivers/allwinner/sunxi_msgbox.c
CC      drivers/allwinner/sunxi_rsb.c
...
CC      plat/allwinner/sun50i_a64/sunxi_power.c
CC      plat/common/plat_gicv2.c
...
Built /home/no/pine64/arm-trusted-firmware/build/sun50i_a64/debug/bl31.bin success
```

There are many more lines. They point us to interesting details. For example, *drivers/allwinner/axp/axp803.c* contains the default settings of the AXP power-management chip, *plat/allwinner/sun50i_a64/sunxi_power.c* tells us how the AXP chip is accessed via memory-mapped I/O.

4. Installing the boot loader on the SD-card

   The steps are described in detail at https://linux-sunxi.org/Bootable_SD_card. For me, it is great to see the option of using a GPT partitioning scheme, which we already use for Sculpt OS on PC hardware. This will hopefully become handy at a later stage.

A few useful U-Boot commands When booting U-Boot from our freshly prepared SD card, we can see U-Boot initializing and probing a bunch of devices. In our current situation, **booting over the network** is the most important functionality. So we turn our attention to the `bootp` command.

```
=> help bootp
bootp - boot image via network using BOOTP/TFTP protocol

Usage:
bootp [loadAddress] [[hostIPaddr:]bootfilename]
```

Let's give it a quick try. My development machine has the IP address 10.0.0.32 within the local network and happens to have a TFTP server running. Just for the test, I put a little file called *something* into the TFTP directory and issue the following command to U-Boot:

```
=>  bootp 10.0.0.32:/var/lib/tftpboot/something

TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/something'.
Load address: 0x42000000
```

Of course, I don't want to manually type this command on every boot. It is much better to tell U-Boot to execute the command automatically for us. This is possible by customizing U-Boot's `bootcmd` environment variable.

```
=> help editenv
editenv - edit environment variable

Usage:
editenv name
    - edit environment variable 'name'

=> editenv bootcmd
edit: bootp 10.0.0.32:/var/lib/tftpboot/something
```

With the `bootcmd` customized to our liking, lets save the new setting. U-Boot provides the command `saveenv` for that, which stores the settings at a predefined location on the MMC / SD card.

```
=> saveenv
Saving Environment to FAT... Card did not respond to voltage select!
Failed (1)
```

Well, this did not work as anticipated. The reason is that there are two MMC devices present. The SD-card is connected to the first MMC controller whereas U-Boot is apparently configured to store its environment via the second MMC controller. Fortunately, the latter setting can be configured in U-Boot's build configuration.

Inside the *u-boot/.config*, we find a configuration variable called CONFIG_ENV_FAT_DEVICE_AND_PA
In the interactive `menuconfig`, the corresponding setting is located at the *Environment* sub menu:

```
(1:auto) Device and partition for where to store the environemt in FAT
```

Changing the setting to 0:auto should do the trick. Of course, we have to go again through the steps of building U-Boot and writing it to the SD-card. But that is a small price to pay for the convenience that awaits us.

Next time in U-Boot, editing the `bootcmd` again to our liking and invoking the `saveenv` command makes us smile:

```
=> saveenv
Saving Environment to FAT... OK
```

From now on, we can save a number of key strokes on each boot. One final tweak would increase our comfort even more. By default, U-Boot initializes the USB controller at boot time. This takes a few seconds, delaying our boot time. Since we don't plan to boot from any USB device during our development workflow, it is better to **skip the USB initialization**. This can be done by changing the `preboot` environment variable from "usb start" to nothing, and of course make the change persistent via the `saveenv` command.

2.3 Bare-metal serial output

In the previous section, we started getting acquainted with the Pine64 hardware, established a serial connection using Linux, and explored the use of the U-Boot boot loader. Now we can move towards running Genode's kernel on the device. Before touching Genode, however, we need to take two precautions.

1. We need to understand the hand-over of execution from the boot loader to the loaded kernel code.
2. In order to know that the right things are happening within our custom code, we need a way to get information out.

To address both questions, we are going to build a custom code blob that can be copied to a predefined physical-memory address and, when executed, prints characters over the serial line. For the latter, we need a primitive way to print debug messages over a serial connection. This section goes through the steps of executing custom code on bare-metal hardware with no kernel underneath, and attaining serial output by poking UART device registers directly.

Information gathering During our initial exploration in Section 2.2, we stumbled over a serial device of type "16550A" at address 0x1c28000 that is apparently used by the Linux kernel by default. We have already seen it in action when we interacted with U-Boot and the Armbian system over the serial connection. Just for reference, here is the corresponding `dmesg` output again:

```
[    2.250163] 1c28000.serial: ttyS0 at MMIO 0x1c28000
                (irq = 31, base_baud = 1500000) is a 16550A
```

There are several ways to find out more about this particular device. For example, one might be inclined to consult chip-vendor documentation. This, however, can be a muddy approach. More often than not, ARM-based SoCs are poorly covered by public documentation, or the available documentation contains uncertainties or even errors. Whenever feasible, I like to follow the path of ground truth, looking at known-to-work code as reference. Let's examine the build configuration of our build of U-Boot, which can be readily found in the *u-boot/.config* file. When searching it for the string "Serial", we quickly end up at the following line:

```
CONFIG_SYS_NS16550=y
```

The driver has to have something like "NS16550" in its name. So let's grep the source tree for files named after this string:

```
$ cd u-boot
$ find | grep -i NS16550
./drivers/serial/ns16550.c
./drivers/serial/ns16550.su
./drivers/serial/.ns16550.o.cmd
./drivers/serial/ns16550.o
./drivers/serial/serial_ns16550.c
./include/ns16550.h
./include/config/sys/ns16550.h
./spl/drivers/serial/ns16550.su
./spl/drivers/serial/serial_ns16550.o
./spl/drivers/serial/.ns16550.o.cmd
./spl/drivers/serial/ns16550.o
./spl/drivers/serial/serial_ns16550.su
./spl/drivers/serial/.serial_ns16550.o.cmd
```

That looks promising. At this point, we are especially interested in drawing the connection to the UART device address 0x1c28000. Remember how we specified `PLAT=sun50i_a64` to the build system of U-Boot? The "sun50i_a64" has to refer to our SoC. So let's grep the source tree for any connection between "sun" and "NS16550".

```
grep -r NS16550 | grep -i sun
...
include/configs/sunxi-common.h:# define CONFIG_SYS_NS16550_COM1  SUNXI_UART0_BASE
include/configs/sunxi-common.h:# define CONFIG_SYS_NS16550_COM2  SUNXI_UART1_BASE
...
```

Next stop, SUNXI_UART0_BASE:

```
grep -r SUNXI_UART0_BASE
...
arch/arm/include/asm/arch-sunxi/cpu_sun9i.h:#define SUNXI_UART0_BASE (REGS_APB1_BASE + 0x000
arch/arm/include/asm/arch-sunxi/cpu_sun4i.h:#define SUNXI_UART0_BASE 0x01c28000
...
```

Now seeing the address 0x01c28000, we know for certain that we are looking at the right device and the corresponding driver code.

The next step consists of cross-checking several pieces of information. Searching the web for "NS16550 pdf" brings up the data sheet for the device (http://caro.su/msx/ocm_de1/16550.pdf). In contrast to SoC chip vendor documentation, data sheets of individual IP cores like this are - if publicly available - usually of good quality. So we are lucky. Glimpsing over the data sheet, we learn that the register at offset 0 is the

so-called transmitter holding register (THR). We must write to this register to print a character. It is interesting to see that all device registers are 8 bits wide. This raises the question how those registers are mapped to system-bus addresses of the ARM SoC. The answer can be found at the Allwinner A64 manual[1] as linked by the Pine64 wiki. Here, we learn that the individual registers are mapped to 32-bit aligned memory-mapped I/O registers. Thus, the register offsets found in the NS16550 data sheet have to be multiplied with 4. The THR register is of course mapped to offset 0. For cross-checking this information, the U-Boot driver code at *drivers/serial/ns16550.c* becomes handy.

What has the NS16550 data sheet has to say about the THR register?

```
Before writing this register the user must ensure that the UART is
ready to accept data for transmission, for example checking that THR
Empty flag is set in the LSR
```

LSR stands for line status register. According to the data sheet, it is the 5th register. Hence, it should be accessible at the ARM system bus at offset 5*4 = 0x14. We also learn that the mentioned "Empty" flag hides behind bit 5 of the LSR.

20 bytes yelling "U" As a preliminary test, let's try to unconditionally write the character `U` (ASCII value 0x55) to the THR register in an infinite loop. The corresponding C program (saving the file as *main.c*) looks as follows:

```
int _start()
{
  for (;;)
    *(unsigned long *)0x1c28000 = 'U';
}
```

Since we will ultimately have to use Genode's tool chain very soon, now would be a good time to install it[2]. The tool chain comes with AARCH64 support. All the utilities can be found at */usr/local/genode/tool/current/bin/*. One may consider adding this directory to the shell's PATH variable to avoid the need for typing out this rather long path. But that is just a matter of convenience.

The following invocation of GCC compiles our little C program into an ELF binary:

```
$ genode-aarch64-gcc -nostdlib main.c -o serial_test
```

[1] https://files.pine64.org/doc/datasheet/pine64/Allwinner_A64_User_Manual_V1.0.pdf
[2] https://genode.org/download/tool-chain

The `-nostdlib` flag tells the compiler that we don't want to link any C runtime or default startup code. Let's inspect the result by disassembling the binary using `objdump`.

```
$ genode-aarch64-objdump -ld serial_test

serial_test:     file format elf64-littleaarch64


Disassembly of section .text:

0000000000400000 <_start>:
_start():
  400000:  d2900000   mov  x0, #0x8000                  // #32768
  400004:  f2a03840   movk  x0, #0x1c2, lsl #16
  400008:  d2800aa1   mov  x1, #0x55                    // #85
  40000c:  f9000001   str  x1, [x0]
  400010:  17fffffc   b  400000 <_start>
```

Even though the instructions look quite alien to me (not being too familiar with the AARCH64 ISA at this point), this looks very reasonable. It's good that the generated code does not rely on a stack pointer because we cannot assume to have a valid stack. However, the link address 0x400000 is concerning because the RAM base address of the A64 SoC is not lower than 0x40000000. Remember, when we looked at Linux' */proc/iomem*, we spotted the following line:

```
40000000-bdffffff : System RAM
```

So we will have to tweak the linker arguments a bit. From our experiments with U-Boot, we learned that U-Boot's default load address 0x42000000 lies within this range. We can use the linker argument `-Ttext` to explicitly specify our desired link address for the text (code) segment:

```
genode-aarch64-gcc -Wl,-Ttext=0x42000000 -nostdlib main.c -o serial_test
```

The `-Wl,` prefix is merely needed to tell the GCC frontend to pass the following argument to the linker. With this tweak, the disassembled binary looks even better:

```
Disassembly of section .text:

0000000042000000 <_start>:
_start():
    42000000:  d2900000   mov  x0, #0x8000                  // #32768
    42000004:  f2a03840   movk  x0, #0x1c2, lsl #16
    42000008:  d2800aa1   mov  x1, #0x55                    // #85
    4200000c:  f9000001   str  x1, [x0]
    42000010:  17fffffc   b  42000000 <_start>
```

The `serial_test` is a complete ELF binary with all kinds of meta data. For running the instructions on the target, we either need an ELF loader (U-Boot can of course do that for us) or we climb the hill barefoot. The latter gives us more control. So let's convert the ELF binary into a raw binary using `objcopy`.

```
genode-aarch64-objcopy -Obinary serial_test serial_test.img
```

We named the raw binary *serial_test.img*. Checking its size, it is quite thrilling to see that it is just 20 bytes of pure usefulness! No overhead.

The next step would be fetching the image via U-Boot's TFTP support. The TFTP server running my development machine serves the directory */var/lib/tftpboot/*. So we have to copy our *serial_test.img* to this directory before turning to U-Boot's console:

```
=> bootp 10.0.0.32:/var/lib/tftpboot/serial_test.img
BOOTP broadcast 1
BOOTP broadcast 2
BOOTP broadcast 3
DHCP client bound to address 10.0.0.178 (1100 ms)
Using ethernet@1c30000 device
TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/serial_test.img'.
Load address: 0x42000000
Loading: #
   4.9 KiB/s
done
Bytes transferred = 20 (14 hex)
```

It seems our program in its entirety reached its designated place. Now it's time to take a jump!

```
=> go 0x42000000
## Starting application at 0x42000000 ...
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU...
```

The serial console gets flooded with U characters. What a joyful moment!
Let's reiterate what we gained by this experiment:

- We know how to compile a custom C program into binary code that works on the target.
- We successfully loaded our binary onto the target and passed control from the boot loader to our code.
- We got a positive lifesign back from our code.

Evolving from primordial vocals to words Until now, we just violently poked the THR register without listening for the status of the UART device. To make the program utter words instead of merely vocals, this ignorance has to stop.

While modifying our program, we have to be careful to not using the stack. While doing these iterative experiments, a little Makefile becomes handy, which prints the disassembled program after each compilation:

```
CROSS_DEV_PREFIX := /usr/local/genode/tool/current/bin/genode-aarch64-

serial_test: main.c
  $(CROSS_DEV_PREFIX)gcc -Wl,-Ttext=0x42000000 -nostdlib $< -o $@
  $(CROSS_DEV_PREFIX)objdump -ld $@

serial_test.img: serial_test
  $(CROSS_DEV_PREFIX)objcopy -Obinary $< $@

test: serial_test.img
  cp $< /var/lib/tftpboot/
```

This little workflow tool not only makes life so much more convenient but it also documents the use of the various commands for the future me. Since I regard it as a mere personal tool of mine, I even don't hesitate place commands like the copying of the image to my TFTP directory in there. Now, by issuing `make test`, the command takes all the steps of compiling, showing the assembly code, creating the raw binary, and copying to the TFTP directory all at once.

Turning back to our actual program, the next baby step would be the output of a string of characters instead of just one character, like so:

```
static char const *text = "Aye aye.\n\r";
static char const *s;

for (;;)
  for (s = text; *s; s++)
    *(unsigned int volatile *)0x1c28000 = *s;
```

You may wonder why the variables `text` and `s` are marked as static? If I made them local variables, which would normally be the better practice, the compiler would generate a stack frame. For example, by merely changing the `for` loop to the innocent looking line

```
for (char const *s = text; *s; s++)
```

the corresponding assembly program will generate instructions changing and dereferencing the stack-pointer register:

```
42000000:  d10043ff   sub  sp, sp, #0x10
42000004:  90000080   adrp  x0, 42010000 <_start+0x10000>
42000008:  91018000   add  x0, x0, #0x60
4200000c:  f9400000   ldr  x0, [x0]
42000010:  f90007e0   str  x0, [sp, #8]
...
```

Since we don't have a stack, this is a big no-no! The `static` keyword tells the compiler to statically allocate the variable at the data (or bss) segment of the binary. Speaking of binary segments, for a bit of a shock, have a look at the binary size now:

```
$ ls -la serial_test.img
-rwxrwxr-x 1 no no 65640 Dez 17 15:30 serial_test.img
```

Isn't that embarrassing? With our change, we inflated the binary size from 20 bytes to more than 64 KiB. This effect is caused by our use of variables, which were completely absent in the initial version. The use of at least one variable prompts the compiler/linker to generate a data segment in addition to the text (code) segment. By default, the linker places each segment at an aligned address using a default alignment. On AARCH64, this default alignment is 64 KiB so that the segment always starts at the beginning of a MMU page when using virtual memory. Because of this default behavior, our few instructions are followed by almost 64 KiB of zeros before the variables start at the next 64 KiB boundary. As of now, we don't use any MMU. So we could in principle weaken the default alignment. Just for reference, the GCC argument for defining

a segment alignment of 16 bytes would be `-Wl,-z -Wl,max-page-size=0x10`. Voila! The image shrunk from 64 KiB to less that 200 bytes. Well, I'll stop the bean counting for now and run this version of the program:

```
## Starting application at 0x42000000 ...
Aye aye.
Aye aye.
Aye aye.
Aye aye.
Aye aye.
Aye aye.
Ayeaaaaaaaaaaaaaaaaaaa...
```

Figure 1

Even though we can see strings of characters, at one point, the output regresses to primordial vocals again. This had to be anticipated since we don't yet check the TX status bit before writing a new character to the THR register. Interestingly, it worked for a while, presumably as long as the capacity of the UART's TX FIFO buffer could swallow the characters.

By the way, while tinkering with devices at such a bare-bones level with almost no infrastructure, an artificial delay can be accomplished as follows:

```
for (i = 0; i < 1000000; i++)
  asm volatile("nop");
```

By adding these lines to the body of the outer for loop, we can indeed observe stable output. But that is of course just a hack. Let's us better change the code to actually evaluate the status bit.

```
int _start()
{
  enum {
    UART_BASE = 0x1c28000,

    THR = UART_BASE,
    LSR = UART_BASE + 0x14,

    LSR_THRE = (1 << 5)
  };

  /* static is needed to prevent the compiler from creating a stack frame */
  static char const *text = "Aye aye.";

  for (;;) {

    static char const *s;

    for (s = text; *s; s++) {

      /* poll 'TX Holding Register Empty' bit */
      while (((*(unsigned int volatile *)LSR) & LSR_THRE) == 0);

      *(unsigned int volatile *)THR = *s;
    }
  }
}
```

Note the amount of lipstick I applied to the code.

- Adding a comment here and there.
- Grouping things with vertical whitespace.
- Using enum values to give magic values tangible names.

I agree that this may be a little excessive for such a temporary test program. But keep in mind that I wrote it not for my present me, but for you, and my future me. Also note that I removed the line break from the `text`, which has no reason other than making the following picture more pretty.

Figure 2: Infinite obedience

Thanks to listening to the UART's TX status bit, the output has become reliable. So now, we have a minimal and known-to-work blueprint for our upcoming kernel's UART driver. With this primitive way to get information out of the board, we can turn our attention to the kernel-porting work, which is the topic of the next section.

2.4 Kernel skeleton

Of the several kernels supported by the Genode OS framework, the so-called base-hw kernel is our go-to microkernel for ARM-based devices. Section 7.7. "Execution on bare hardware" of the Genode Foundations book goes into detail about its underlying software design. This section describes the process of porting this kernel to a new board, specifically the Pine-A64-LTS single-board computer.

Equipped with the bare-metal serial-output facility developed in the previous section, we are eager to turn our attention to the kernel. Before attempting the porting of the kernel to the new board, however, it is recommended to run it first on one of the already supported boards to have a working reference. In the case of the Pine-A64 board, which is based on an Allwinner multi-core 64-bit ARM SoC, the closest approximation would be the NXP i.MX8Q EVK board, which ticks the boxes ARM, multi-core, and 64-bit. At the very least, one should give the kernel a try using Qemu's virtual pbxa9 board, which is a 32-bit platform. Even though this board has not much in common with ours, it is still useful for seeing how the various bits and pieces described below are supposed to work together.

2.4.1 A tour through the code base

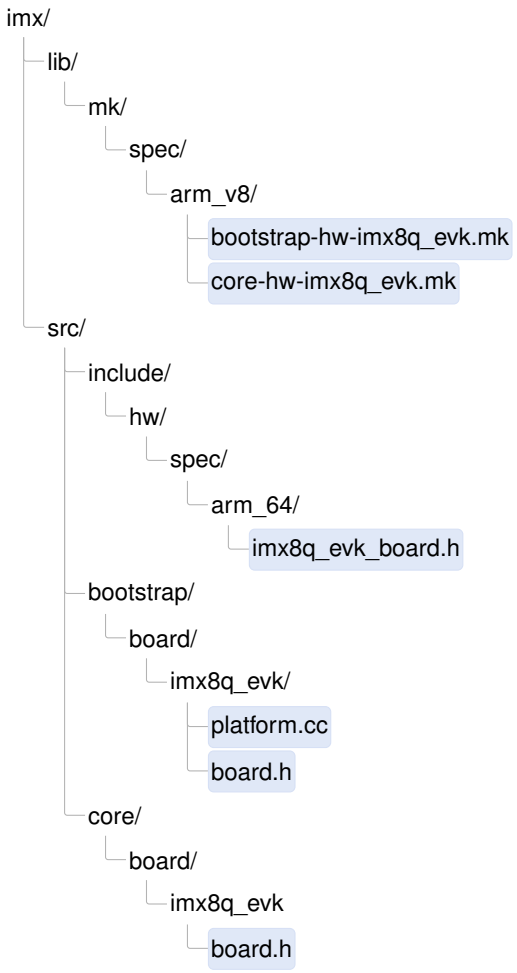
The starting point of our line of work will be the existing board support for the i.MX8Q EVK. To get an idea of the amount of work ahead of us, let's examine the base-hw source tree within Genode for occurrences of the board's name. The search pattern "imx" is a good start.

```
$ find repos/base-hw -type f | grep imx8
repos/base-hw/lib/mk/spec/arm_v8/core-hw-imx8q_evk.mk
repos/base-hw/lib/mk/spec/arm_v8/bootstrap-hw-imx8q_evk.mk
repos/base-hw/recipes/src/base-hw-imx8q_evk/hash
repos/base-hw/recipes/src/base-hw-imx8q_evk/content.mk
repos/base-hw/recipes/src/base-hw-imx8q_evk/used_apis
repos/base-hw/src/bootstrap/board/imx8q_evk/platform.cc
repos/base-hw/src/bootstrap/board/imx8q_evk/board.h
repos/base-hw/src/include/hw/spec/arm_64/imx8q_evk_board.h
repos/base-hw/src/core/board/imx8q_evk/board.h
```

We can ignore everything inside the *recipes/* directory for now. This directory contains package descriptions. We will come back to the packaging topic later. A `grep -v` hides these files from our view.

```
$ find repos/base-hw -type f | grep imx8 | grep -v recipes
repos/base-hw/lib/mk/spec/arm_v8/core-hw-imx8q_evk.mk
repos/base-hw/lib/mk/spec/arm_v8/bootstrap-hw-imx8q_evk.mk
repos/base-hw/src/bootstrap/board/imx8q_evk/platform.cc
repos/base-hw/src/bootstrap/board/imx8q_evk/board.h
repos/base-hw/src/include/hw/spec/arm_64/imx8q_evk_board.h
repos/base-hw/src/core/board/imx8q_evk/board.h
```

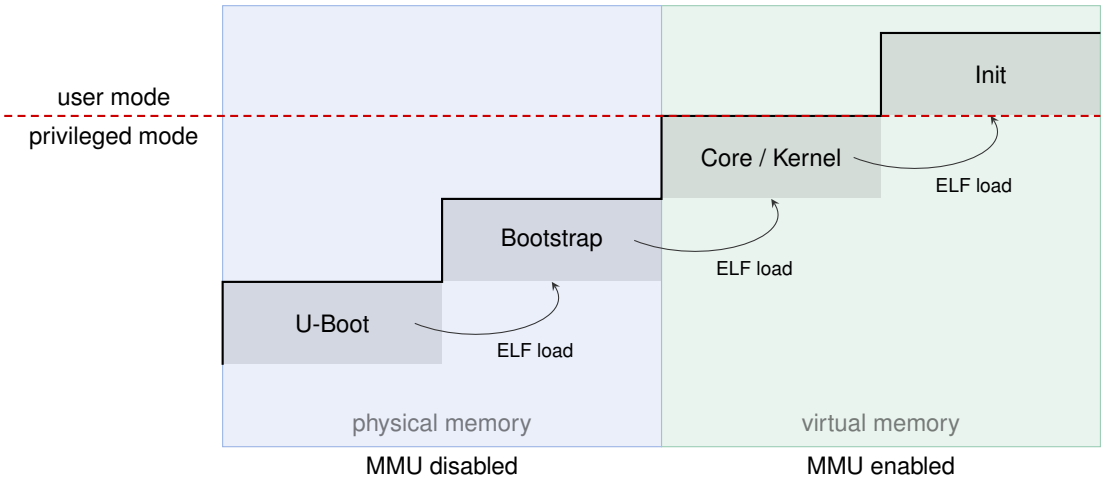
On the one hand, it is nice to see such a small number of files to be concerned about. On the other hand, those files appear quite scattered throughout the source tree with a deep hierarchy, which is a bit confusing. To lift the clouds, let's have a look at the source-tree structure.



The files appearing under *lib/mk/* are build-description files for libraries. There are two such files, having the file extension `.mk`. They are located in a sub directory called

spec/arm_v8/, which means that the build system[1] considers them only when building for an instruction set architecture that matches ARMv8.

Distinction between bootstrap and core Given the set of files depicted above, we can immediately spot two construction sites, namely "bootstrap" and "core". The distinction between those two parts is illustrated in the following picture.



The **bootstrap** program is started by the boot loader while the CPU is running in physical mode. The MMU is disabled at this point. Only one CPU - usually referred to as the boot CPU - is active. Bootstrap is tasked with all the dirty and quirky work needed in preparation to bring up the so-called core component. This involves board-specific trickery like tweaking clocks and voltages, setting up the page tables for executing the core program in virtual memory, enabling the MMU, the initialization of additional CPU cores, and the ELF-loading of the core ELF executable. Once these steps are taken, bootstrap passes the control to the core component and ceases to exist.

The **core** component contains the microkernel executed in privileged mode. When using Genode on a traditional microkernel like NOVA or seL4, core is the first user-level program started by the kernel. It is usually called roottask. In contrast, when using base-hw as we are going to do now, core and the kernel are one single program. Core *is* the microkernel at the root of Genode's component tree. Hence, in the following, the terms core and kernel are used synonymously.

Core is executed with the MMU enabled. It is globally mapped at the upper part of the virtual address space. To operate as the kernel, it contains basic drivers for the interrupt controller, kernel timer (for preemptive scheduling), cache maintenance, and cross-CPU synchronization. For the interplay with the user level components running on top

[1] https://genode.org/documentation/genode-foundations/20.05/development/Build_system.html

of core, it features code paths for exiting the kernel into the user land and, vice versa, for entering the kernel from the user land (syscalls, exceptions, interrupts). Functionality-wise, it implements mechanisms for inter-component communication, asynchronous notifications, physical-memory allocation, the management of virtual address spaces, and the world-switching between virtual machines (if used as a hypervisor). In short, everything a microkernel needs to do and - more importantly - nothing a microkernel shouldn't do.

Review of the board-specific code Before starting the work on the new board support, let us briefly look into each of the files for the existing i.MX8q EVK board. Genode's support for the NXP i.MX family is hosted in the dedicated genode-imx repository[1]. Let's draw our attention to the files named after board.

```
repos/imx$ find | grep imx8q_evk
```

In the list of files, we spot three header files, one *board.h* header under *src/bootstrap/*, one *board.h* header under *src/core/*, and one *imx8q_evk_board.h* header under *src/include/*. The former two files are specific for bootstrap and core, whereas the latter contains definitions useful for both programs. The *board.h* files are located in directories named after the board. With this structure, generic (board-agnostic) code can `#include <board.h>`. The build system picks the right *board.h* file by adding the board-specific directory to the include-search path.

Let us start with with definitions used across bootstrap and core.

repos/imx/src/include/hw/spec/arm_64/imx8q_evk_board.h

[1] https://github.com/genodelabs/genode-imx

```
#include <drivers/uart/imx.h>
#include <hw/spec/arm/boot_info.h>

namespace Hw::Imx8q_evk_board {
    using Serial = Genode::Imx_uart;

    enum {
        RAM_BASE   = 0x40000000,
        RAM_SIZE   = 0xc0000000,

        UART_BASE  = 0x30860000,
        UART_SIZE  = 0x1000,
        UART_CLOCK = 250000000,
    };

    namespace Cpu_mmio {
        enum {
            IRQ_CONTROLLER_DISTR_BASE  = 0x38800000,
            IRQ_CONTROLLER_DISTR_SIZE  = 0x10000,
            IRQ_CONTROLLER_VT_CPU_BASE = 0x31020000,
            IRQ_CONTROLLER_VT_CPU_SIZE = 0x2000,
            IRQ_CONTROLLER_REDIST_BASE = 0x38880000,
            IRQ_CONTROLLER_REDIST_SIZE = 0xc0000,
        };
    };
}
```

Both bootstrap and core need to know the memory-mapped device registers for the UART device to print diagnostic messages. The UART driver (*drivers/uart.imx.h*) is included. The `Serial` type refers to the concrete UART driver implementation as present on the board. Thanks to this definition, generic code is able to rely on the UART functionality via the type name `Serial`.

The start and size of physical memory must be known by both bootstrap and core. So it is defined here.

Both bootstrap and core access the interrupt controller. Whereas bootstrap performs the one-time initializations needed in order to start secondary CPU cores, core drives the interrupt controller at runtime.

The bootstrap-specific files concern build descriptions and actual code. The build description looks as follows.

repos/imx/lib/mk/spec/arm_v8/bootstrap-hw-imx8q_evk.mk

```
REP_INC_DIR += src/bootstrap/board/imx8q_evk

SRC_CC  += bootstrap/board/imx8q_evk/platform.cc
SRC_CC  += bootstrap/spec/arm/gicv3.cc
SRC_CC  += bootstrap/spec/arm_64/cortex_a53_mmu.cc
SRC_CC  += lib/base/arm_64/kernel/interface.cc
SRC_CC  += spec/64bit/memory_map.cc
SRC_S   += bootstrap/spec/arm_64/crt0.s

NR_OF_CPUS = 4

vpath spec/64bit/memory_map.cc $(call select_from_repositories,src/lib/hw)

vpath bootstrap/% $(REP_DIR)/src

include $(call select_from_repositories,lib/mk/bootstrap-hw.inc)
```

The i.MX8 SoC uses the GICv3 as interrupt-controller. Hence, the driver gicv3.cc is included. In contrast, as we learned from the Linux boot log, the Allwinner A64 SoC uses the GICv2 interrupt controller.

The MMU driver differs between the various ARM versions. The i.MX8 is based on A53 CPU cores. The Allwinner A64 uses the same.

The assembly file *arm_64/crt0.s* contains the entry point into the program as jumped to by the boot loader.

The NR_OF_CPUS definition is used for the static allocation of data structures that must be present for each CPU. Hence, this value is globally defined.

The strange looking `$(call select_from_repositories...)` is a mechanism for accessing files across different source repositories. You can find the mechanism described in Section 5.3. “Build system” in the Genode Foundations book.

repos/imx/src/bootstrap/board/imx8q_evk/board.h

```
#include <hw/spec/arm_64/imx8q_evk_board.h>
#include <hw/spec/arm_64/cpu.h>
#include <hw/spec/arm/gicv3.h>
#include <hw/spec/arm/lpae.h>

namespace Board {
    using namespace Hw::Imx8q_evk_board;

    struct Cpu : Hw::Arm_64_cpu
    {
        static void wake_up_all_cpus(void*);
    };

    using Hw::Pic;
}
```

The Board namespace aggregates the knowledge of the board details that matter to the bootstrap code, namely the specific interrupt controller (gicv3.h) and the declaration of the `wake_up_all_cpus` function. The `Board` namespace hosts the `Pic` (programmable interrupt controller) type, which allows the generic code of bootstrap to interact with the interrupt controller without knowing the exact type of device.

repos/imx/src/bootstrap/board/imx8q_evk/platform.cc

```
Bootstrap::Platform::Board::Board()
:
  early_ram_regions(Memory_region { ::Board::RAM_BASE, ::Board::RAM_SIZE }),
  late_ram_regions(Memory_region { }),
  core_mmio(Memory_region { ::Board::UART_BASE, ::Board::UART_SIZE },
            Memory_region { ::Board::Cpu_mmio::IRQ_CONTROLLER_DISTR_BASE,
                            ::Board::Cpu_mmio::IRQ_CONTROLLER_DISTR_SIZE },
            Memory_region { ::Board::Cpu_mmio::IRQ_CONTROLLER_REDIST_BASE,
                            ::Board::Cpu_mmio::IRQ_CONTROLLER_REDIST_SIZE })
{
    ::Board::Pic pic {};

    ... incomprehensible magic spells, some gibberish about GPIO, CCM, PLL ...
}

void Board::Cpu::wake_up_all_cpus(void * ip)
{
    ... more magic spells, digressing into assembly code ...
}
```

The `early_ram_regions`, `late_ram_regions`, and `core_mmio` data structures are initialized with the known ranges of physical memory and memory-mapped I/O registers. This information is designated to be passed further to core.

The call of `::Board::Pic pic ;` performs basic interrupt-controller initialization that is needed only once. It is followed by a sequence of board-specific tweaks to bring the board into a defined state for the kernel to rely on. For instance, setting the I/O MUX configuration, default voltages, and frequencies. The U-boot boot loader already does a fine job for establishing a base line but it is rather conservative. The code for the i.MX8 EVK boosts the voltages and frequencies for improving the performance.

The `wake_up_all_cpus` call invokes a hook to enable secondary CPU cores. The used mechanism varies from board to board, specifically depending on the operation of the ARM Trusted Firmware. We have to brace ourself for some investigation once we look into multi-processor support. At the beginning, however, we will use only the boot CPU. So we can ignore this function for now.

Finally, let's turn our attention to the core-specific files.

repos/imx/lib/mk/spec/arm_v8/core-hw-imx8q_evk.mk

```
REP_INC_DIR += src/core/board/imx8q_evk
REP_INC_DIR += src/core/spec/arm/virtualization

# add C++ sources
SRC_CC += kernel/vm_thread_on.cc
SRC_CC += spec/arm/gicv3.cc
SRC_CC += spec/arm_v8/virtualization/kernel/vm.cc
SRC_CC += spec/arm/virtualization/platform_services.cc
SRC_CC += spec/arm/virtualization/vm_session_component.cc
SRC_CC += vm_session_common.cc
SRC_CC += vm_session_component.cc

#add assembly sources
SRC_S += spec/arm_v8/virtualization/exception_vector.s

NR_OF_CPUS = 4

# include less specific configuration
include $(call select_from_repositories,lib/mk/spec/arm_v8/core-hw.inc)
```

Core needs to know the type of the interrupt controller because it processes interrupts at runtime. Here, the GICv3 driver is incorporated.

Similar to bootstrap, a few data structures within core are statically allocated for each CPU, hence the NR_OF_CPUS must be specified here as well.

We can ignore the files with `vm_*` and `virtualization` in their names for now. They are important for hosting virtual machines. Since the virtualization support is a generic feature of the ARM CPU, we don't have to take board-specific precautions.

repos/imx/src/core/board/imx8q_evk/board.h

```
#include <hw/spec/arm_64/imx8q_evk_board.h>
#include <spec/arm/generic_timer.h>
#include <spec/arm/virtualization/gicv3.h>
#include <spec/arm_v8/cpu.h>
#include <spec/arm_64/cpu/vm_state_virtualization.h>
#include <spec/arm/virtualization/board.h>

namespace Board {
    using namespace Hw::Imx8q_evk_board;

    enum {
        TIMER_IRQ           = 14 + 16,
        VT_TIMER_IRQ        = 11 + 16,
        VT_MAINTAINANCE_IRQ = 9  + 16,
        VCPU_MAX            = 16
    };
}
```

In addition to the aggregation of headers matching the board and SoC - like the generic timer driver - we see the definitions of just the few interrupt numbers that are important to core. The kernel is completely oblivious about all other peripheral devices.

The VCPU_MAX definition is solely used for the dimensioning of an array that keeps the state of virtual CPUs for virtual machine. It is not important for now.

2.4.2 A new home for the board support

The easiest way to add support for a new board is the mirroring of the files introduced above. We could march forward with adding new files and directories to a new branch of the Genode repository. Alternatively, the Genode build system allows us to host our custom board-specific files in a dedicated source repository that we can maintain independently from the Genode main repository. The latter approach has the following advantages.

First, it reinforces a clean separation between board-specific code from generic Genode code. In particular, the *segregation of code* constricts the working set of files relevant for a given board, keeping only important code in view.

Operationally, it allows the decoupling of *code ownership* in terms of responsibility, quality assurance, licensing hygiene, development process, and the choice of source hosting.

Finally, it alleviates the pressure to agree on one big joint code base, *removing* potential points of *friction* between developers.

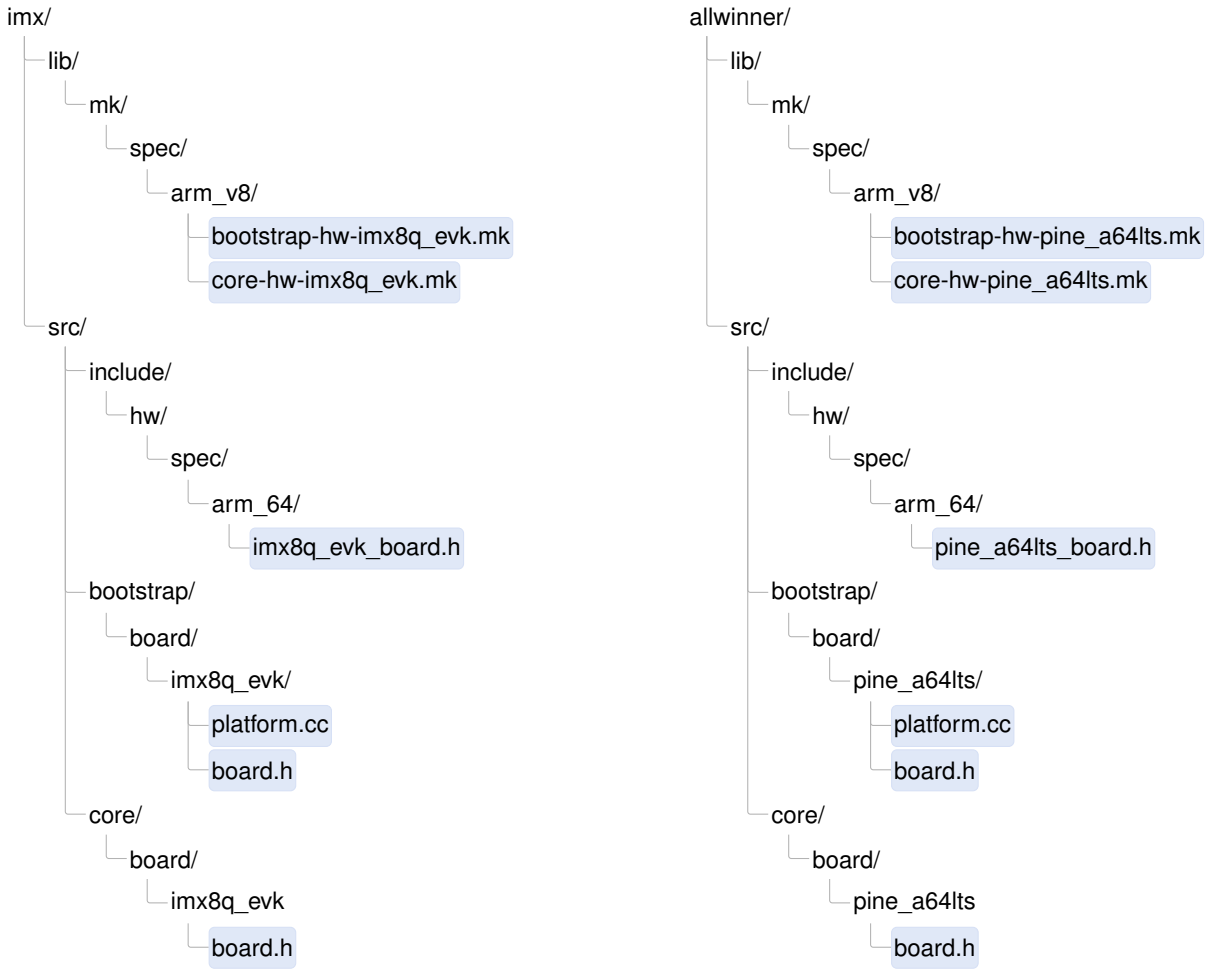
In the following, we will put our code into a new repository named *allwinner*.

```
mkdir repos/allwinner
```

In principle, the directory can be anywhere but I find it practical to host it under the *repos* directory of the Genode source tree. One may also opt to use a symlink, e. g., *repos/allwinner* pointing to *~/src/genode-allwinner.git*.

We need to come up with with a concise name for our board support. Throughout Genode, we follow certain **naming conventions**. In particular, we use underscore _ for tightly coupled words, and minus - for loosely coupled terms. For example, in the file name *core-hw-imx8q_evk.mk*, "imx8q_evk" belong closely together whereas the words "core" and "hw" are used as some kind of category (read: the "core" component for the "hw" kernel for the "imx8q_evk" board). With the background of these conventions, the board name **pine_a64lts** seems sensible. Specific enough while still concise.

For the initial content from our new *allwinner* repository be blatantly mirror the files of the base-hw repository.



At the current stage, we are concerned about getting the build process right. To concentrate at this one thing at a time, let us pretend that the Pine-A64-LTS board works equal to the i.MX8 EVK. We don't mind that the technicalities copied from the existing board don't match our new board until we run the code on the board. That said, as the build-description files (those with the `mk` suffix) steer the build process, they must be made consistent with our directory structure. So we have to revisit those files while looking out for the pattern `imx8q_evk`.

A look into *lib/mk/spec/arm_v8/bootstrap-hw-pine_a64lts.mk* reveals the following line:

```
REP_INC_DIR += src/bootstrap/board/imx8q_evk
```

We have to replace it with

```
REP_INC_DIR += src/bootstrap/board/pine_a64lts
```

Similarly, *allwinner/lib/mk/spec/arm_v8/core-hw-pine_a64lts.mk* contains the line:

```
REP_INC_DIR += src/core/board/imx8q_evk
```

This must be changed to

```
REP_INC_DIR += src/core/board/pine_a64lts
```

System-integration dry-run Let us see how the Genode build system swallows - or chokes on - our new board support. First, we need a build directory for the ARMv8 architecture.

```
$ ./tool/create_builddir arm_v8a
Successfully created build directory at /.../genode/build/arm_v8a.
Please adjust /.../genode/build/arm_v8a/etc/build.conf according to your needs.
```

As suggested, we open *build/etc/build.conf* in our favorite text editor. Normally, I enable parallel builds by uncommenting the corresponding line right at the beginning of the file. But for now, let us keep it disabled until the skeleton builds successfully. The steps of the build system are easier to follow if it operates deterministically.

We need to extend the `REPOSITORIES` variable with the path to our custom repository. For the *allwinner* repository, that would be following line:

```
REPOSITORIES += $(GENODE_DIR)/repos/allwinner
```

Note that the order of REPOSITORIES defines the search order of the build system for files. If the *allwinner* repository should be able to override content of the other repositories, specifically base-hw, the above line should appear before the others.

With these changes in place, we can issue the build of bootstrap for new board.

```
$ cd build/arm_v8a
$ make bootstrap/hw KERNEL=hw BOARD=pine_a64lts
 ...
 Library bootstrap-hw-pine_a64lts
   ...
   MERGE    bootstrap-hw-pine_a64lts.lib.a
 Program bootstrap/hw/bootstrap_hw_pine_a64lts
```

The result can be found in the sub directory *bootstrap/hw/*. We find a single object file named *bootstrap-hw-pine_a64lts.o* along with a stripped version of this file.

Likewise, core for the base-hw kernel and the new board can be built as follows.

```
$ make core KERNEL=hw BOARD=pine_a64lts
   ...
   MERGE    core-hw-pine_a64lts.lib.a
 Program core/hw/core_hw_pine_a64lts
```

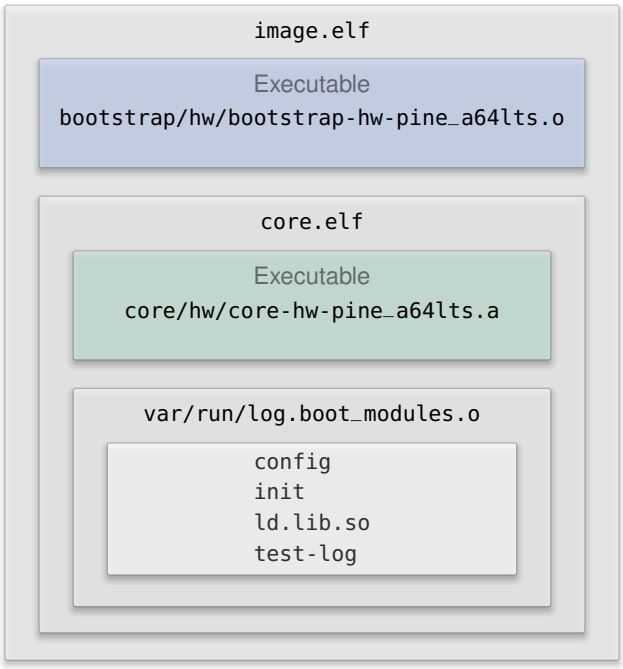
Similar to the build of bootstrap, we can find the result at the corresponding subdirectory, here *core/hw/*. We find a single archive file named *core-hw-pine_a64lts.a* along with a stripped version of this file.

Next up, we are going to build a **system image** that contains both core and bootstrap. Now would be a good time to enable parallel builds. Edit the *etc/build.conf* file by uncommenting the following line (removing the hash # character).

```
#MAKE += -j4
```

One may also opt to write the BOARD and KERNEL arguments directly into the *build.conf* file as illustrated by the commented-out examples. This spares the need to specify the arguments each time when issuing a build command.

A system image contains bootstrap, core, and additional boot modules. The first two puzzle pieces are already in place. But what about the boot modules? In contrast to bootstrap and core, which are always the same for each system scenario, the boot modules vary between system scenarios. Genode system scenarios are defined in the form of run scripts. The run script at *repos/base/run/log.run* is a good starting point. As defined by this particular run script, the system image for the "log" system scenario is comprised of core, init, ld.lib.so, init, and test-log in addition to a configuration. A system image (image.elf) for this scenario would look like this:



Genode's run tool automates the process of assembling such Matryoshkas from the various pieces. Let's give it a try:

```
$ make run/log KERNEL=hw BOARD=pine_a64lts
...
... long sequence of compile steps
...
genode build completed
using 'ld-hw.lib.so' as 'ld.lib.so'
core link address is 0xffffffc000000000

Error: unknown image link address

 File board/pine_a64lts/image_link_address not present in any repository.

Makefile:329: recipe for target 'run/log' failed
```

This message should prompt us to have closer look at the run tool.

```
$ cd genode
$ grep -r "unknown image link address" tool
tool/run/boot_dir/hw: puts stderr "\nError: unknown image link address\n"
```

The file *tool/run/boot_dir/hw* is the part of the run tool that defines the integration of a system image from its parts for the base-hw kernel. It is worth skimming over the file to

get a rough understanding of how the system image is assembled from its ingredients. The error message above comes from the function `bootstrap_link_address` called during the system-image integration step.

The link address is evaluated by the boot loader when loading the system image as ELF binary. It defines the start of the text segment of the system image in physical memory. As the physical memory layout differs between SoCs and boards, we must provide a value that is suitable for the memory layout of the Pine-A64-LTS board. From looking at Linux' */proc/iomem*, we remember that the system RAM of our board starts at 0x40000000.

As indicated by the error message above, the run tool expects to find the link address in a file called *board/pine_a64lts/image_link_address*. Let's create such a file with a sensible value. It is common practice to leave some room at the very beginning of the memory, which is often occupied by the boot loader. It is usually fine to link the system image to 64 KiB after the start of the physical memory.

```
$ cd allwinner
$ mkdir -p board/pine_a64lts
$ echo 0x40010000 > board/pine_a64lts/image_link_address
```

With the link address defined, another attempt to build the system image for the log scenario succeeds. The result can be found in the build directory's *var/run/* sub directory:

```
$ find var/run
var/run
var/run/log.boot_modules.o
var/run/log
var/run/log/boot
var/run/log/boot/image.elf
var/run/log.core
var/run/log.bootstrap
var/run/log.config
```

The most interesting file is certainly *var/run/log/boot/image.elf*, which is the final system image. To quickly validate the link address, let's check the ELF entrypoint.

```
$ readelf -a var/run/log/boot/image.elf | grep Entry
  Entry point address:               0x40010000
```

The value looks familiar. While we are at it, the other files are also worth inspecting.

var/run/log.boot_modules.o is an aggregate of all boot modules of the system scenario.

var/run/log.core is an ELF binary of core without the boot modules. The binary contains all debug information. This is handy for debugging the core component. For example, using this binary, the instruction pointer of a page fault within core can be related to the matching source code using `objdump`.

var/run/log.bootstrap is an ELF binary of the bootstrap code without core and the boot modules. As for the core log.core binary, it is handy for debugging the bootstrap code.

var/run/log.config is the `config` boot module passed to the initial init component. It corresponds to the snippet passed the `install_config` function as found in the *log.run* script.

By the way, one may prefer booting a uImage instead of an ELF image because a uImage is compressed using gzip by default, which reduces the boot time. The run tool supports that via the argument `-include image/uboot`. One can either extend the RUN_OPT variable by adding a corresponding line to *etc/build.conf* or pass the option to the make command line:

```
$ RUN_OPT='--include image/uboot' make run/log BOARD=pine_a64lts KERNEL=hw
```

After completing the build, the uImage file can be found at *var/run/log/uImage*.

This is not magic. At this point, I recommend taking a look at the run tool's snippets located at *tool/run/*. In particular, *tool/run/image/uboot* contains the sequence of commands used for generating the uImage from the ELF image.

2.4.3 Getting to grips using meaningful numbers

The faux system image that we just created contains information cowardly copied from the imx8q_evk board, and which certainly mismatches the pine_a64lts board. So let's revisit the files in our repository one by one and look out for any numbers. Numbers are important. According to my experience, hexadecimal numbers are especially important. Don't forget to squinch your eyes when looking at them. Change them with caution.

```
$ cd repos/allwinner
$ find -type f
./lib/mk/spec/arm_v8/bootstrap-hw-pine_a64lts.mk
./lib/mk/spec/arm_v8/core-hw-pine_a64lts.mk
./board/pine_a64lts/image_link_address
./src/bootstrap/board/pine_a64lts/platform.cc
./src/bootstrap/board/pine_a64lts/board.h
./src/include/hw/spec/arm_64/pine_a64lts_board.h
./src/core/board/pine_a64lts/board.h
```

lib/mk/spec/arm_v8/bootstrap-hw-pine_a64lts.mk

The following line catches our attention:

```
SRC_CC  += bootstrap/spec/arm/gicv3.cc
```

The i.MX8 SoC uses ARM's Generic Interrupt Controller version 3 (GICv3). From booting Linux on the Pine-A64 board, we learned that the Allwinner SoC uses the GIC version 2. Fortunately, the base-hw kernel supports both versions. So we can change the line to:

```
SRC_CC  += bootstrap/spec/arm/gicv2.cc
```

The NR_OF_CPUS value can stay unmodified because the Allwinner SoC has 4 cores.

lib/mk/spec/arm_v8/core-hw-pine_a64lts.mk

We merely also have to adjust the GIC version from 3 to 2.

src/bootstrap/board/pine_a64lts/platform.cc

The file contains a lot of i.MX8Q-specific initialization steps like tweaking clocks and voltages. We can remove this code without looking back. The body of the `Bootstrap::Platform::Board` constructor can be reduced to the mere initialization of the interrupt controller:

```
{
  ::Board::Pic pic { };
}
```

The list of memory regions passed to the `core_mmio` member can be pruned to the single entry for the UART. The other entries that refer to the IRQ controller should be removed because they refer to the wrong version of the GIC anyway. We will supplement the proper regions for the GICv2 later, once we turn our attention to interrupts.

```
core_mmio(Memory_region { ::Board::UART_BASE, ::Board::UART_SIZE })
```

At this point, I am admittedly unsure about the `wake_up_all_cpus` implementation, in particular whether the opcode of the CPU_ON smc instruction would match. I guess not. We will come to multi-processor support at a later stage. So let's better remove the uncertainty by reducing the implementation to

```
void Board::Cpu::wake_up_all_cpus(void *) { }
```

src/bootstrap/board/pine_a64lts/board.h

We see several things that cry for adjustment.

- Updating the include guards
- Including the correct board definitions by replacing

  ```
  #include <hw/spec/arm_64/imx8q_evk_board.h>
  ```

  by

  ```
  #include <hw/spec/arm_64/pine_a64lts_board.h>
  ```

- Incorporating the GICv2 driver instead of the GICv3 driver by changing

  ```
  #include <hw/spec/arm/gicv3.h>
  ```

  to

  ```
  #include <hw/spec/arm/gicv2.h>
  ```

- Defining the C++ type `Pic` such that it refers to the `Hw::Gicv2` driver:

  ```
  using Pic = Hw::Gicv2;
  ```

src/include/hw/spec/arm_64/pine_a64lts_board.h

To our despair, the file is full of numbers.

- It includes the driver for the UART used for printing debug messages. Of course, the specified drivers/uart/imx.h driver won't work. While experimenting with the bare-metal serial output, we have learned that the Allwinner SoC uses a NS16550 UART controller. Let us pretend having a driver by changing the line to

```
#include <drivers/uart/ns16550.h>
```

- The board-specific name space should reflect the name of our board:

```
namespace Hw::Pine_a64lts_board {
```

- We want the C++ type `Hw::Serial` to refer to our hypothetical NS16550 driver.

```
using Serial = Genode::Ns16550_uart;
```

- The RAM_BASE and RAM_SIZE values must match those we found from the look at Linux */proc/iomem*.

```
RAM_BASE = 0x40000000,
RAM_SIZE = 0x7e000000,
```

- We already have found known-good values for UART_BASE and UART_SIZE during our bare-metal serial output experimentation. The UART_CLOCK value won't be needed in our case. So we define it as zero.

```
UART_BASE  = 0x1c28000,
UART_SIZE  = 0x1000,
UART_CLOCK = 0,
```

- The IRQ_CONTROLLER_REDIST_BASE and SIZE are not used for the GICv2. So the values can be removed.
- The values for IRQ_CONTROLLER_DISTR_BASE and SIZE as well as VT_CPU_BASE and SIZE will become important once we will turn our attention to the interrupt controller. But this is not today. So we keep the existing numbers, keeping in mind that they won't work.
- When using the GICv2, we need to add the definition of IRQ_CONTROLLER_CPU_BASE and VT_CTRL_BASE. Until we use interrupts, we can pick an arbitrary number. To display good manners, let's leave the lowest 12 bits to zero, pretending that each device resource starts at a page boundary.

```
IRQ_CONTROLLER_CPU_BASE     = 0xaaaaa000,
IRQ_CONTROLLER_VT_CTRL_BASE = 0xbbbbb000,
```

/src/core/board/pine_a64lts/board.h

The file contains mostly interrupt numbers. We will turn our attention to interrupts later. Let's not touch them for now because we cannot validate the values anyway at this point. Apart from these numbers, a few adjustments must be made.

- Updating the include guard
- Including the board definitions from *pine_a64lts_board.h*
- Adjusting the GIC version of the included header from *gicv3.h* to *gicv2.h*
- Importing the board-specific namespace `Hw::Pine_a64lts_board`

To wrap up this step, let's check if we missed any leftover by grepping for remaining occurrences of patterns like "imx" or "gicv3".

```
$ grep -ri imx repos/allwinner
```

Now would also be a good time to revisit the file headers, updating the information about the author, creation date, brief description, and copyright. Should the code be considered to eventually become part of the Genode upstream project at some point, it is sensible to leave the license disclaimer as is, clarifying that the code is designated be a part of the Genode OS framework.

UART driver for bootstrap and core The next attempt to build the system image for the log scenario fails predictably:

```
$ make run/log KERNEL=hw BOARD=pine_a64lts
...
    COMPILE  core_region_map.o
In file included from /.../repos/allwinner/src/core/board/pine_a64lts/board.h:17,
                 from /.../repos/base-hw/src/core/platform.h:37,
                 from /.../repos/base-hw/src/core/core_region_map.cc:18:
/.../repos/allwinner/src/include/hw/spec/arm_64/pine_a64lts_board.h:17:10:
           fatal error: drivers/uart/ns16550.h: No such file or directory
 #include <drivers/uart/ns16550.h>
          ^~~~~~~~~~~~~~~~~~~~~~~~
```

We can find a number of blueprints for our new UART driver at *repos/base/include/drivers/uart/*. By following the lines of the existing drivers and combining our knowledge from the bare-metal serial experiments, we can come up with the following little driver placed at *allwinner/include/drivers/uart/ns16550.h*.

```
#include <util/mmio.h>

namespace Genode { class Ns16550_uart; }


class Genode::Ns16550_uart : Mmio
{
  private:

    struct Thr : Register<0x00, 32>
    {
      struct Data : Bitfield<0,8> { };
    };

    struct Lsr : Register<0x14, 32>
    {
      struct Thr_empty : Bitfield<5,1> { };
    };

  public:

    Ns16550_uart(addr_t const base, uint32_t, uint32_t) : Mmio(base) { }

    void put_char(char const c)
    {
      while (read<Lsr::Thr_empty>() == 0);

      write<Thr::Data>(c);
    }
};
```

Like all drivers dedicatedly developed for Genode, it uses Genode's `Register` API to safely access bits of memory-mapped I/O registers. You can find the API described in Section 8.18 "Utilities for user-level device drivers" in the Genode-Foundations book.

Climbing the mountain step by step We are almost there. On our walk, we repeatedly try to build the system image, look at the compiler and linker errors, fix them, and repeat.

```
$ make run/log KERNEL=hw BOARD=pine_a64lts
    ...
    COMPILE  bootstrap/spec/arm/gicv2.o
/.../repos/base-hw/src/bootstrap/spec/arm/gicv2.cc:
                    In constructor 'Hw::Gicv2::Gicv2()':
/.../repos/base-hw/src/bootstrap/spec/arm/gicv2.cc:23:28:
                    error: 'NON_SECURE' is not a member of 'Board'
  bool use_group_1 = Board::NON_SECURE &&
                             ^~~~~~~~~~
```

The interrupt-controller driver apparently needs to distinguish the cases where the kernel is running in the so-called "secure world" or "normal world" of ARM TrustZone. If you want to learn more about schizophrenia as a feature of ARM processors, let me point you to our article on ARM TrustZone[1]. Admittedly, I'm not completely sure about which of both worlds are executing our kernel. But it is probably safe to assume that the boot process switches to the normal world before loading and starting our system image. So we add the definition of NON_SECURE to *allwinner/src/bootstrap/board/pine_a64lts/board.h*.

```
namespace Board {
  ...
  static constexpr bool NON_SECURE = true;
}
```

The next slope on our way up the hill:

```
$ make run/log KERNEL=hw BOARD=pine_a64lts
    ...
    MERGE    bootstrap-hw-pine_a64lts.lib.a
/.../genode-aarch64-ar: bootstrap/board/pine_a64lts/platform.o:
                        No such file or directory
/.../repos/base/mk/lib.mk:180: recipe for target
                               'bootstrap-hw-pine_a64lts.lib.a' failed
```

We have to guide the build system to consider source files in the *allwinner* repository, by adding the following line to *lib/mk/spec/arm_v8/bootstrap-hw-pine_a64lts.mk*.

```
vpath bootstrap/% $(REP_DIR)/src
```

[1] https://genode.org/documentation/articles/trustzone

Next try. This time, we get a link error:

```
/.../aarch64-none-elf/bin/ld: debug/core-hw-pine_a64lts.a(cpu.o):
                              in function 'Board::Pic::Pic()':
/.../repos/base-hw/src/core/spec/arm/virtualization/gicv2.h:22:
    undefined reference to 'Board::Pic::Gich::Gich()'
```

It turns out that the virtualization-related parts of the GICv2 driver reside in a distinct compilation unit located at *base-hw/src/core/spec/arm/virtualization/gicv2.cc*, which is not yet included in the build description for core. We have to add the following line to *allwinner/lib/mk/spec/arm_v8/core-hw-pine_a64lts.mk*.

```
SRC_CC += spec/arm/virtualization/gicv2.cc
```

With these minor obstacles addressed, we get a system image that should largely be compatible with our board. The urge to try out the freshly baked system image on the board is strong. Why not?

2.4.4 A first life sign of the kernel

Testing the system image on the board comes down to the following few steps.

1. Make sure to build the uImage using the `image/uboot` RUN_OPT.

   ```
   $ RUN_OPT='--include image/uboot' make run/log BOARD=pine_a64lts KERNEL=hw
   ```

2. Copy the uImage from *build/arm_v8a/var/run/log/uImage* to the TFTP directory. In my case, that is */var/lib/tftpboot/*.

3. Boot the board and use U-Boot's `bootp` and `bootm` commands to load the uImage via TFTP and start it.

```
=> bootp 10.0.0.32:/var/lib/tftpboot/uImage
BOOTP broadcast 1
BOOTP broadcast 2
BOOTP broadcast 3
DHCP client bound to address 10.0.0.178 (1121 ms)
Using ethernet@1c30000 device
TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/uImage'.
Load address: 0x42000000
Loading: #################################################################
         2.7 MiB/s
done
=> bootm
## Booting kernel from Legacy Image at 42000000 ...
   Image Name:
   Image Type:   AArch64 Linux Kernel Image (gzip compressed)
   Data Size:    887610 Bytes = 866.8 KiB
   Load Address: 40010000
   Entry Point:  40010000
   Verifying Checksum ... OK
   Uncompressing Kernel Image

Starting kernel ...

Error: Assertion failed: id < _count && _cpus[id].constructed()
Error:   File: /.../repos/base-hw/src/core/kernel/cpu.cc:205
Error:   Function: Kernel::Cpu& Kernel::Cpu_pool::cpu(unsigned int)
```

The excitement is real! That's the first life sign of Genode's kernel! We get three satisfactory results at once. First, our custom `Ns16550_uart` driver is working, as evidenced by the beautifully formatted error messages. So we did not mess up any of the important numbers there. Second, in contrast to the archaic experiments with the bare-metal serial output, which did not even use a stack, we can now enjoy the comfort of Genode's C++ runtime. We don't feel like living in a cave any longer.

2.5 Low-level debugging

Some kids from the city once told me about programs called “debuggers”. They also use a technology named “green light” to cross the streets. City kids. As we are still far away from urban territory, we are in need of the rural ways of debugging. What are our options?

Remember, at the end of the previous section, we were greeted with the first life sign of the kernel:

```
Error: Assertion failed: id < _count && _cpus[id].constructed()
Error:   File: /.../repos/base-hw/src/core/kernel/cpu.cc:205
Error:   Function: Kernel::Cpu& Kernel::Cpu_pool::cpu(unsigned int)
```

This raises the questions: What is the trouble about? How did we get there? What went wrong? Thankfully, the message gives us a concrete reference to the code *cpu.cc* at line 205.

```
Cpu & Cpu_pool::cpu(unsigned const id)
{
    assert(id < _count && _cpus[id].constructed());
    return *_cpus[id];
}
```

By looking at this code, I’m tempted to draw the connection to the corners we cut regarding the `Board::Cpu::wake_up_all_cpus` method, which we deliberately left empty. But let us leave this speculation for later.

To get hold of the situation, it is useful to know which part of the condition fails. This can be revealed by adding the following instrumentation at the beginning of the method.

```
Genode::log("cpu: id=", id, " _count=", _count, " "
            "constructed=", _cpus[id].constructed());
```

The `Genode::log` function is declared in the *base/log.h* header. Note that it relies on a fair bit of framework infrastructure such as synchronization primitives. In desperate situations during the debugging of lowest-level framework code, the `Genode::raw` function can become handy as a drop-in replacement. In contrast to `log`, the `raw` function relies on less infrastructure. In practice, I use `log` by default and `raw` as last resort. After rebuilding the system image and rebooting the board, it turns out that the `log` function works well at this point.

```
...
cpu: id=0 _count=4 constructed=0
Error: Assertion failed: id < _count && _cpus[id].constructed()
```

The instrumentation tells us that the first element of the `_cpus` array has not been properly constructed. But how to find out why? We ultimately need to know the call chain that led to execution of the `Cpu_pool::cpu` method.

2.5.1 Option 1: Walking the source code

The most obvious approach is studying the source code, and determining the immediate callers of the method using `grep`.

```
repos/base-hw$ grep -r "\<cpu("
```

Unfortunately, "cpu" is a pretty bad pattern to grep for. It is too generic. However, we know that the code in question must reside inside *repos/base-hw/* and can thereby restrict the search to only this part of the source tree. Furthermore, by using "
mbox<" (match only the start of a word) and appending "(" to the pattern (as expected at the caller site), we can narrow down the number of matches to a useful level.

```
src/test/cpu_quota/main.cc:                  env.cpu()),
src/test/cpu_quota/main.cc:       Cpu_session::Quota quota = env.cpu().quota();
src/core/spec/arm_v8/virtualization/kernel/vm.cc: _vcpu_context(cpu_pool().cpu(cpu))
src/core/spec/arm_v8/virtualization/kernel/vm.cc: affinity(cpu_pool().cpu(cpu));
src/core/spec/arm_v7/virtualization/kernel/vm.cc: _vcpu_context(cpu_pool().cpu(cpu))
src/core/spec/arm_v7/virtualization/kernel/vm.cc: affinity(cpu_pool().cpu(cpu));
src/core/kernel/kernel.cc:    Cpu &cpu = cpu_pool().cpu(Cpu::executing_id());
src/core/kernel/cpu_mp.cc:    Irq(Board::Pic::IPI, cpu), cpu(cpu)
src/core/kernel/cpu.h:        Cpu & cpu(unsigned const id);
src/core/kernel/cpu.h:        Cpu & primary_cpu() { return cpu(Cpu::primary_id()); }
src/core/kernel/cpu.h:        Cpu & executing_cpu() { return cpu(Cpu::executing_id()); }
src/core/kernel/cpu.h:            for (unsigned i = 0; i < _count; i++) func(cpu(i));
src/core/kernel/cpu.cc:Cpu & Cpu_pool::cpu(unsigned const id)
src/core/kernel/cpu_up.cc:Kernel::Cpu::Ipi::Ipi(Kernel::Cpu & cpu) : ...
src/core/kernel/cpu_context.h:        void cpu(Cpu &cpu) { _cpu = &cpu; }
src/core/kernel/thread.cc:    Cpu & cpu = cpu_pool().cpu(user_arg_2());
```

From these results, we can immediately disregard the lines referring to *src/test/*. Also the virtualization-related matches are most likely not of interest. When inspecting the remaining lines, the number of potential callers comes down to 5:

```
src/core/kernel/kernel.cc:    Cpu &cpu = cpu_pool().cpu(Cpu::executing_id());
src/core/kernel/cpu.h:        Cpu & primary_cpu() { return cpu(Cpu::primary_id()); }
src/core/kernel/cpu.h:        Cpu & executing_cpu() { return cpu(Cpu::executing_id()); }
src/core/kernel/cpu.h:            for (unsigned i = 0; i < _count; i++) func(cpu(i));
src/core/kernel/thread.cc:    Cpu & cpu = cpu_pool().cpu(user_arg_2());
```

With such a low amount of callers, we can apply brute force by adding the following line just before each call.

```
Genode::log(__FILE__, ":", __LINE__);
```

The compiler replaces `__FILE__` with a string of the file name of the source code and `__LINE__` with a string of the line number where `__LINE__` appears within the source file. Another useful magic macro is `__PRETTY_FUNCTION__`, which expands to the name of the surrounding function.

After rebooting the board with the instrumentations in place, we see the origin of the call:

```
/.../repos/base-hw/src/core/kernel/kernel.cc:25
```

A look at the surrounding code reveals that the function call originates from a function called `kernel`:

```
extern "C" void kernel()
{
  ...
  Cpu &cpu = cpu_pool().cpu(Cpu::executing_id());
  ...
}
```

You might guess what's next?

```
repos/base-hw$ grep -r "\<kernel(" *
```

There is only one caller, which is at *src/core/kernel/init.cc* and brings the `kernel_init` function to our attention.

Granted, this step-wise instrumentation may feel a bit like chopping wood with a nail clipper. But I sometimes enjoy the process anyway. By following call chains in reverse by browsing and instrumenting the code, one develops some kind of peripheral vision

for the code around the call path, which fosters the sense of familiarity with the code base.

Of course, using `grep` manually as described above may be too archaic for your taste. There exist plenty of dedicated tools (like ctags, cscope) and IDEs for aiding source-code discovery after all. Personally, I prefer simple tools. As a small life hack, I have put the following snippet in my Vim configuration:

```
nnoremap <leader>g :execute
         \ "grep! -R -I --exclude-dir=.git
                      \ --exclude=*.orig
                      \ --exclude=.*.swp
                      \ --exclude=*.rej
                      \ --exclude=*~ "
                      \ . shellescape("\\<" . expand("<cword>") . "\\>")
                      \ . " ."<cr>:copen<cr><cr>
```

Similar to how the * and # commands search for the word under the cursor in the current buffer, the `<leader>g` command above allows me to grep the word under the cursor in the source tree and presents the results in a quickfix window. So I can quickly jump to each occurrence and travel across source files like a poor man's hypertext system.

That all said, once when ending up in a situation with many callers, the approach of manually instrumenting all caller sites becomes a nuisance, which leads us to the second option.

2.5.2 Option 2: One step of ground truth at a time

Instead of instrumenting all potential caller sites, we can let the return addresses as found on the stack guide us by using the following line as instrumentation:

```
Genode::log("called from ", __builtin_return_address(0));
```

When executed, this line prints us the return address of the current function scope. This address corresponds to the caller. By placing this line into the `Cpu_pool::cpu` method, we get the following output.

```
Starting kernel ...

called from 0xffffffc000058720
Error: Assertion failed: id < _count && _cpus[id].constructed()
```

The high number immediately tells us that the executed code resides somewhere high up in virtual memory. That means, we have already passed the bootstrap stage, the MMU is enabled, and core/kernel code is executed. As explained in the Section 2.4, the corresponding ELF binary resides at *build/arm_v8a/var/run/log.core* and can be inspected via `readelf`.

```
build/arm_v8a$ readelf -l var/run/log.core | grep LOAD
  LOAD           0x0000000000001000 0xffffffc000000000 0xffffffc000000000
  LOAD           0x00000000000c1000 0xffffffc0000c0000 0xffffffc0000c0000
  LOAD           0x00000000000ef5c0 0x0000000000000000 0x0000000000000000
```

The addresses of the ELF segments correlate nicely with the value printed by our instrumentation. To determine the exact source-code location for the given return address, the **objdump** tool becomes handy. It allows one to disassemble an ELF binary while displaying the source-code intermixed. The tool is specific to the used CPU architecture. That is, for 64-bit ARM, it is called *genode-aarch64-objdump*. To use it interactively from the shell, the tool chain's *bin/* directory should be added to the shell's PATH variable:

```
$ export PATH=/usr/local/genode/tool/current/bin/:$PATH
```

With the PATH variable set, we can disassemble the log.core ELF binary and pipe the result to `less` for inspection:

```
build/arm_v8a$ genode-aarch64-objdump -lSd var/run/log.core | less
```

Note that the amount of output generated by objdump can be huge. By replacing `less` by `wc -l` one can see that the output comprises more than 300,000 lines! Still, this amount of data leaves `less` unimpressed, which leaves me impressed. We can simply search for our address `ffffffc000058720` (with the 0x prefix stripped away) via the slash (`/`) command and end up at the following section of output:

```
kernel():
/.../base-hw/src/core/kernel/kernel.cc:25
ffffffc000058718:       12001c21        and     w1, w1, #0xff
ffffffc00005871c:       97fff8e3        bl      ffffffc000056aa8 <_ZN6Kernel8Cpu
_pool3cpuEj>
ffffffc000058720:       aa0003f4        mov     x20, x0
/.../base-hw/src/core/kernel/kernel.cc:29
        Cpu_job * new_job;
```

The source location *kernel.cc* line 25 looks familiar.

Alternatively to going though the disassembled output of objdump, the **addr2line** utility can be used to streamline the lookup of a source-code location by a given instruction address.

```
$ genode-aarch64-addr2line -e var/run/log.core 0xffffffc000058720
/.../base-hw/src/core/kernel/kernel.cc:25
```

This is fast and convenient. But sometimes, in particular when code is excessively inlined, the contextual information given by the objdump output can be valuable. Most often, I scroll upwards until hitting the next occurrence of a `.cc` file and watch silently the lines - header-file names and fragments of source code - that scroll by. Again, peripheral vision at play.

2.5.3 Option 3: Backtraces

The `__builtin_return_address` feature of the compiler allows us to follow the call chain one step at a time. Each step involves a manual instrumentation, a compile-test cycle, and the invocation of the `addr2line` utility.

To avoid such repetitive work, Genode provides the utility function `Genode::backtrace()` to walk the stack and print the return addresses along the way. This function is declared in the *os/backtrace.h* header. An instrumentation of the `Cpu_pool::cpu` method would look as follows.

```
#include <os/backtrace.h>

Cpu & Cpu_pool::cpu(unsigned const id)
{
  Genode::backtrace();
  ...
}
```

To assist the `backtrace()` function to parse stack frames correctly, the Genode build system must be instructed to preserve frame-pointer information. This can be achieved by placing the following line to the build directory's *etc/tools.conf* file. Note that by default there is no such file. So you will have to create one containing this line.

```
CC_OPT += -fno-omit-frame-pointer
```

After rebuilding and running the system image the next time, we are greeted with quite a lot of output:

```
Starting kernel ...

0xffffffc000058738
0xffffffc00000085c
0xffffffc0000568e0
0xffffffc000056a60
0xffffffc000057e44
0x400273d8
0x40026754
0x40010068
0xffffffc000058738
```

Each of the values starting with `0xfff...` is a valid return address and can be used with objdump or addr2line as described above. To make matters more convenient, the addr2line utility can be used in an "interactive" fashion be running the following command in a separate terminal.

```
build/arm_v8a$ genode-aarch64-addr2line -e var/run/log.core
```

With no address specified at the command line, the tool simply waits for standard input. So we can paste multiple lines of the `Genode::backtrace()` output directly into it and get the following result:

```
0xffffffc000058738
0xffffffc00000085c
0xffffffc0000568e0
0xffffffc000056a60
0xffffffc000057e44
/.../base-hw/src/core/kernel/kernel.cc:25
:?
/.../base-hw/src/core/spec/arm/virtualization/gicv2.h:22
/.../base/include/util/reconstructible.h:56
/.../base-hw/src/core/kernel/init.cc:64 (discriminator 1)
```

We can spot both of the familiar locations *kernel.cc* line 25 and *init.cc* line 64.

As shown above, the standard GNU binutils and compiler features can bring us quite far without using a debugger. We have gathered a lot of input for investigating the error. Our next job will be using this information to discharge it.

2.6 Excursion to the user land

Equipped with the rudimentary debugging skills presented in the previous section, it is time to conquer the remaining stumbling blocks on our way to the user land.

To quickly recall, the starting point of our investigation was the following error message.

```
Error: Assertion failed: id < _count && _cpus[id].constructed()
Error:   File: /.../repos/base-hw/src/core/kernel/cpu.cc:205
Error:   Function: Kernel::Cpu& Kernel::Cpu_pool::cpu(unsigned int)
```

By following the call chain leading to this message in reverse, we ultimately arrived at *base-hw/src/core/kernel/init.cc* at line 64 right in the middle of the function `kernel_init`:

```
  pool_ready = cpu_pool().initialize();
```

To double check that the error indeed occurs somewhere in the `initialize` method, let's wrap the call with a bit of instrumentation.

```
 Genode::log("call cpu_pool().initialize()");
 pool_ready = cpu_pool().initialize();
 Genode::log("pool_ready=", pool_ready);
```

The resulting output confirms our hypothesis.

```
Starting kernel ...

call cpu_pool().initialize()
Error: Assertion failed: id < _count && _cpus[id].constructed()
```

It is always good to have the reassurance about still being on the right track. As we suspected, `cpu_pool().initialize()` is called but never returns. So let's look at its implementation in *base-hw/src/core/kernel/cpu.cc*.

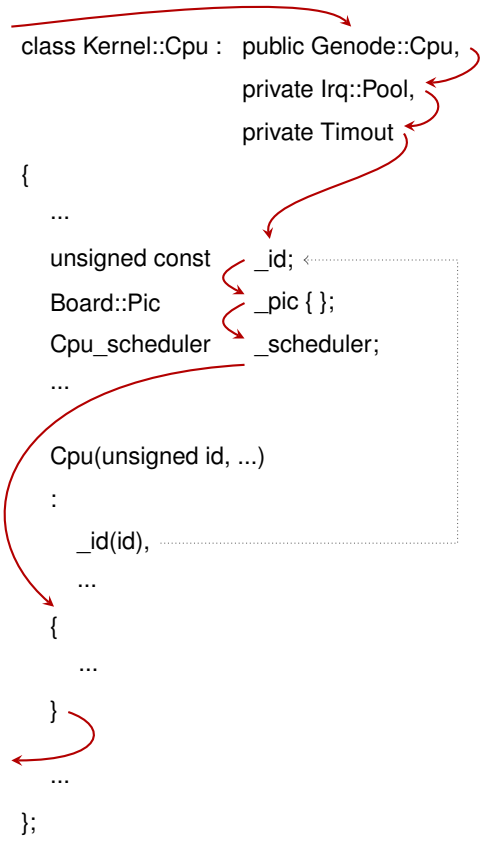
```
bool Cpu_pool::initialize()
{
    unsigned id = Cpu::executing_id();
    _cpus[id].construct(id, _global_work_list);
    return --_initialized == 0;
}
```

Each element of the `_cpus` array is a `Constructible<Cpu>` object. The `Constructible` pattern is used throughout Genode. It allows for the static allocation of dynamically created objects. The `construct` method triggers the construction of a `Cpu` object. We are ultimately faced with a general question: How to instrument the construction of C++ objects?

Debugging the construction of C++ objects The lowest-hanging fruit is adding a message right at the beginning of the constructor's body:

```
Cpu::Cpu(unsigned const id, Inter_processor_work_list & global_work_list)
:
  ... plenty of initializers ...
{
  Genode::log(__PRETTY_FUNCTION__);
  _arch_init();
}
```

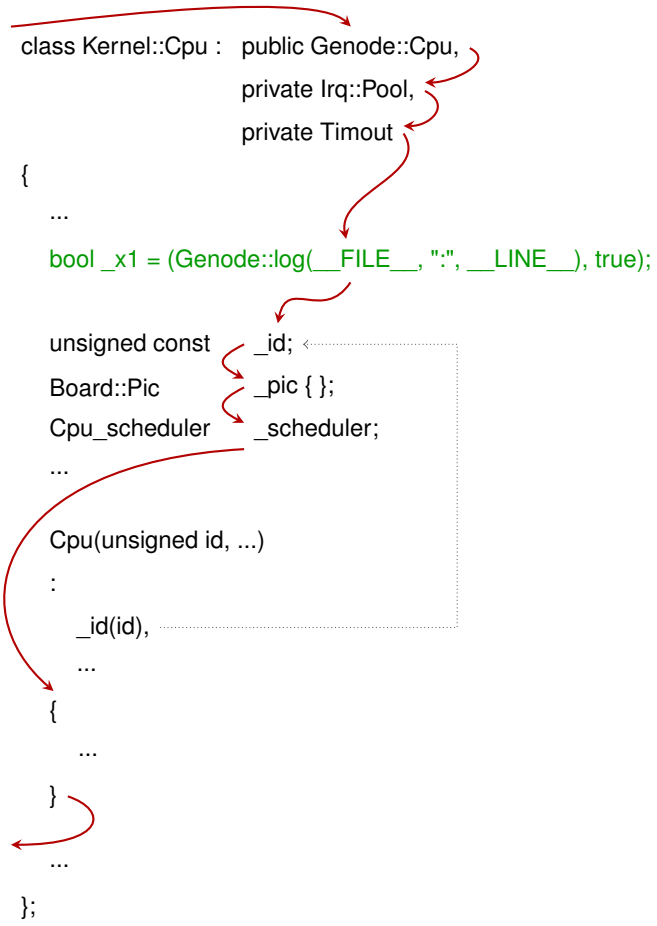
Upon the next run, we see no such message. So we can conclude that we get stuck in the middle of the construction of one of the base classes or aggregated members. As illustrated by the following picture, the body of the constructor is called pretty late in the process of constructing an object.



Placing debug messages gets a little bit more cumbersome now. We have to disguise such messages as object attributes. For example, by placing the following line right at the start of the class body, we can see whether we get stuck in the construction of one of the base classes or - later - during the construction of a member.

```
bool _x1 = ( Genode::log(__FILE__, ":", __LINE__), true );
```

The effect of this instrumentation looks as follows.



The trick is to wrap the `log` call into an expression that can be used as initialization of a dummy member. When the construction of the `Cpu` object reaches the point of the `_x1` member, we see the message as a side effect. The member `_x1` is never actually used.

On the next run, we see the following:

```
Starting kernel ...

call Cpu_pool::initialize()
/.../repos/base-hw/src/core/kernel/cpu.h:77
Error: Assertion failed: id < _count && _cpus[id].constructed()
```

Since we see the message, we know that the problem occurs not in any of the base classes but during the construction of a subsequent member. To find out which one, we can spill dummy members in-between the various members, like so:

```
    unsigned const _id;
    bool _x2 = ( Genode::log(__FILE__, ":", __LINE__), true );
    Board::Pic     _pic {};
    bool _x3 = ( Genode::log(__FILE__, ":", __LINE__), true );
    Timer          _timer;
    bool _x4 = ( Genode::log(__FILE__, ":", __LINE__), true );
    Cpu_scheduler  _scheduler;
    bool _x5 = ( Genode::log(__FILE__, ":", __LINE__), true );
    Idle_thread    _idle;
    bool _x6 = ( Genode::log(__FILE__, ":", __LINE__), true );
    Ipi            _ipi_irq;
    bool _x7 = ( Genode::log(__FILE__, ":", __LINE__), true );
```

If this looks unsophisticated, it's because it is. The next run reveals the following.

```
call cpu_pool().initialize()
bool Kernel::Cpu_pool::initialize()
/plain/no/genode.git/repos/base-hw/src/core/kernel/cpu.h:78
/plain/no/genode.git/repos/base-hw/src/core/kernel/cpu.h:117
Error: Assertion failed: id < _count && _cpus[id].constructed()
```

From this message, we can conclude that the construction of the `_pic` member is the problem. Does that ring a bell? In the backtrace we obtained in Section 2.5.3, observed the following line.

```
/.../base-hw/src/core/spec/arm/virtualization/gicv2.h:22
```

We could have saved some time by following the output of the backtrace utility more closely, but we would have missed our little excursion to the C++ constructor instrumentation.

By continuing the manual instrumentation work, we end up in the `Gicv2` constructor, specifically in the initialization of the `_max_irq` member. The `max_irq` function interacts with memory-mapped registers of the interrupt controller. Recalling that we have merely provided dummy values of the register addresses, the failure is no longer a mystery at all.

Let's revisit the corners that we cut while mirroring the i.MX8 EVK board support:

- We kept the definitions for memory-mapped I/O regions for the IRQ controller's CPU_BASE and DISTR_BASE untouched, knowing that the values most certainly mismatch with the Allwinner SoC.
- We pruned the `core_mmio` regions to cover only the UART. So even if core had the right numbers, it could not access the underlying hardware registers.

- We set NR_OF_CPUS to 4 but left `Board::Cpu::wake_up_all_cpus` empty.

There are quite a few uncertainties. A good way to reduce them is to first take the multi-core-related issues from the table. From experience, we know that the bring-up of secondary CPU cores can be a pain. So let us safe this topic for a later step.

By bringing up a single-processor variant of the kernel first, we will certainly reach the state of a working kernel more quickly. Subsequent user-level developments like driver-related work can then happen in parallel with the fiddly work on the kernel's multi-processor support. Disabling the kernel's multi-processor support comes down to changing the `NR_OF_CPUS` definition from 4 to 1 in the two files *lib/mk/spec/arm_v8/bootstrap-hw-pine_a64lts.mk* and *lib/mk/spec/arm_v8/core-hw-pine_a64lts.mk*.

Making the interrupt controller driver happy The ARM GIC interrupt controller consists of two parts. Similar to distinction between the I/O APIC and local APIC on x86 hardware, there exists a so-called distributor and a CPU-local interrupt controller. The distributor is responsible for routing interrupts to CPU cores whereas the CPU-local interrupt controller handles the interrupt delivery for an individual CPU. So on a 4-core SoC, there are one distributor and four CPU-local interrupt controllers. The memory-mapped registers of all CPU-local interrupt controllers are the same whereas each CPU can access only its own local controller.

To find out the addresses of both parts for the Allwinner SoC, there are two convenient sources of information. First, the U-Boot boot loader that we built in a Section 2.2.2 comes with a huge database of board specifications in the form of so-called *device tree* (dts) files inside the directory *u-boot/arch/arm/dts/*. By grepping for "pine" we find many files referring to "sun50i". By grepping for "gic" in all files named "sun50i", we end up at *sun50i-a64.dtsi*. In there, the following snippet catches our attention:

```
u-boot/arch/arm/dts$ vim sun50i-a64.dtsi

    gic: interrupt-controller@1c81000 {
      compatible = "arm,gic-400";
      reg = <0x01c81000 0x1000>,
            <0x01c82000 0x2000>,
            <0x01c84000 0x2000>,
            <0x01c86000 0x2000>;
      interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(4) | IRQ_TYPE_LEVEL_HIGH)>;
      interrupt-controller;
      #interrupt-cells = <3>;
    };
```

By looking at the numbers, we unfortunately still don't know which register ranges refers to the distributor and the CPU local controller. We could consult ARM's official documentation.

Alternatively, we find the answer in the Allwinner A64 user manual[1] on page 74. It states the following:

```
GIC_DIST: 0x01C80000 + 0x1000
GIC_CPUIF:0x01C80000 + 0x2000
```

With this knowledge gained, we can change the definitions in our *pine_a64lts_board.h* file to the following.

```
IRQ_CONTROLLER_DISTR_BASE = 0x01c81000,
IRQ_CONTROLLER_DISTR_SIZE = 0x1000,
IRQ_CONTROLLER_CPU_BASE   = 0x01c82000,
IRQ_CONTROLLER_CPU_SIZE   = 0x2000,
```

Additionally, those resources must be registered as core's memory-mapped I/O regions in *board/pine_a64lts/platform.cc*.

```
Bootstrap::Platform::Board::Board()
:
  early_ram_regions(Memory_region { ::Board::RAM_BASE, ::Board::RAM_SIZE }),
  late_ram_regions(Memory_region { }),
  core_mmio(Memory_region { ::Board::UART_BASE, ::Board::UART_SIZE },
            Memory_region { ::Board::Cpu_mmio::IRQ_CONTROLLER_DISTR_BASE,
                            ::Board::Cpu_mmio::IRQ_CONTROLLER_DISTR_SIZE },
            Memory_region { ::Board::Cpu_mmio::IRQ_CONTROLLER_CPU_BASE,
                            ::Board::Cpu_mmio::IRQ_CONTROLLER_CPU_SIZE })
{
  ::Board::Pic pic {};
}
```

When building and running the run/log system image the next time, we get filled with joy:

[1] https://linux-sunxi.org/images/b/b4/Allwinner_A64_User_Manual_V1.1.pdf

```
Starting kernel ...


kernel initialized
ROM modules:
 ROM: [000000004012c000,000000004012c156) config
 ROM: [0000000040006000,0000000040007000) core_log
 ROM: [00000000401eb000,000000004022c260) init
 ROM: [0000000040134000,00000000401eacb0) ld.lib.so
 ROM: [0000000040004000,0000000040005000) platform_info
 ROM: [000000004012d000,00000000401331e8) test-log

Genode 20.11-197-g635985f542 <local changes>
2010 MiB RAM and 64533 caps assigned to init
[init -> test-log] hex range:          [0e00,1680)
[init -> test-log] empty hex range:    [0abc0000,0abc0000) (empty!)
[init -> test-log] hex range to limit: [f8,ff]
[init -> test-log] invalid hex range:  [f8,08) (overflow!)
[init -> test-log] negative hex char:  0xfe
[init -> test-log] positive hex char:  0x02
[init -> test-log] floating point:     1.70
[init -> test-log] multiarg string:    "parent -> child.7"
[init -> test-log] String(Hex(3)):     0x3
[init -> test-log] Very long messages:
[init -> test-log -> log] 1.....................................................................
[init -> test-log] 3.....................................................................
[init -> test-log] 5.....................................................................
[init -> test-log]
[init -> test-log] Test done.
```

We just witnessed the first successful excursion to the user land. The kernel started the user-level init component, which in turn started the test-log program as child component. The output of test program looks just perfect! To truly appreciate what just happened, consider that the simple system scenario already entails most of Genode's fundamental mechanisms:

- Transition between kernel and user land and vice versa
- Multiple protection domains protected by virtual memory
- Synchronous inter-component communication calls (RPC)
- Asynchronous notifications
- Shared memory between components
- The ELF loading of programs

- Handling of the system's configuration
- Multi-threading and inter-thread synchronization
- Dynamic linking

The simple log-test scenario above is just the beginning. In the next section, we take the board through the entire test suite of the Genode base framework.

2.7 Kernel packaging and testing

With our toes still a bit frozen from testing the waters of the user land, we now take the remaining steps towards a cultivated Genode life, largely automating our work flow, packaging the kernel, and testing the platform like there is no tomorrow.

During the initial user-land bring-up described in the previous Section 2.6, the process of building a system image, loading the image onto the board, and obtaining log output required quite a few manual steps: Starting picocom, issuing the `make run/log` command, copying the system image to the TFTP directory, resetting the board, scanning the log output with our eyes. Some parts of this process can be streamlined.

2.7.1 Accelerating our run-script workflow

First, instead of manually instructing the run tool to produce a uImage instead of an ELF image at the command line, we can place the following line into our *etc/built.conf* file.

```
RUN_OPT += --include image/uboot
```

Second, we can let the run tool manage the execution of the `picocom` command instead of manually starting the it by adding the following line:

```
RUN_OPT += --include log/serial
```

This way, we can skip the step of spawning of picocom. But more importantly, the run tool becomes able to detect the success of run scripts automatically! So the part about "scanning the log output with our eyes" becomes much more relaxing.

Third, the copying of the uImage file into the TFTP directory can be automated by adding the following lines.

```
RUN_OPT += --include load/tftp
RUN_OPT +=   --load-tftp-base-dir /var/lib/tftpboot
RUN_OPT +=   --load-tftp-offset-dir /$(BOARD)
```

Upon the next execution of the `make run/log KERNEL=hw BOARD=pine_a64lts` command, a new symbolic link appears at */var/lib/tftpboot*.

```
$ ls -la /var/lib/tftpboot/pine_a64lts
lrwxrwxrwx ... /var/lib/tftpboot/pine_a64lts -> /.../build/arm_v8a/var/run/log/uImage
```

The symlink is updated each time a run script is executed. It always points to the most recently built system image. By setting the U-Boot bootcmd to load */var/lib/tftpboot/pine_a64lts*, the board will automatically fetch the most recently built system image.

```
=> env edit bootcmd
edit: bootp 10.0.0.32:/var/lib/tftpboot/pine_a64lts ; bootm
=> env save
Saving Environment to FAT... OK
```

With these little tweaks, the work flow with run scripts becomes almost fully automated. The only remaining manual steps are:

1. Issuing the `make run/...` command at the build directory.
2. Once the message `Terminal ready` appears, pressing the reset button on the board.

This is a perfectly acceptable level of convenience. If you want to go even further, you may find the following two articles inspiring.

Remote-control your test target via power scripts

https://genodians.org/chelmuth/2019-03-13-powerplug

Exploring Genode Base HW with Raspberry Pi - further workflow automation

https://genodians.org/tomga/2019-08-13-rpi-automation

There is no better way to celebrate the new level of efficiency than to test-drive a few hand-picked run scripts.

2.7.2 Stress-testing the init component

The log scenario that we executed so far already employed Genode's init component, which is the first (and only) component immediately started by core. Init constructs a subsystem of components according to a configuration in XML form. The init configuration for the log scenario was rather primitive. There exists comprehensive test that exercises the entire feature set of init by running a dynamically configured instance of init as a child of init. The test is hosted at the *repos/os/* repository and has the form of a deployable package (more on that later). You can find the test's ingredients at *recipes/pkg/test-init/* (package with the runtime description) *src/test/init/* (driver for executing a test sequence) *recipes/raw/test-init/test-init.config* (sequence executed by the test driver).

That's probably be a bit overwhelming. Let's better just try it out. To run a test package, the *os/run/test.run* script becomes handy. We can use it as follows, passing the name of the test package as `PKG` argument.

```
build/arm_v8a$ make run/test PKG=test-init KERNEL=hw BOARD=pine_a64lts
...
...
genode build completed
...
Terminal ready
```

At this point, we have to press the reset button of the board.

```
...
... log output of more than 200 test steps
...
[init -> test -> test-init] --- test complete ---
[init -> test] child "test-init" exited with exit value 0

Run script execution successful.
```

What else could we ask for! When examining the log output, you can get a glimpse of the feature set at work: Addition and deletion of subsystems, changing access-control policies in the fly, reconfiguring child components, chaining services, balancing resources among the components, heartbeat monitoring, and exit handling.

As another noteworthy detail, in contrast to the simple log test, the init test employs a timer at the user level. Since the test passed, we have the confirmation that the in-kernel timer driver and interrupt-controller driver work in principle.

2.7.3 Timer accuracy test

Speaking of the timer, it is generally not enough to know that the timer works in principle but also that it is precise, which comes down to its correct calibration. Genode provides a ready-to-use test that compares the notion of time as observed by the Genode system with the wall-clock time as known on your host system. The run script for this low-level test is located at *repos/base/run/timer_accuracy.run*.

```
build/arm_v8a$ make run/timer_accuracy KERNEL=hw BOARD=pine_a64lts
...
...
Genode 20.11-197-g635985f542 <local changes>
2010 MiB RAM and 64533 caps assigned to init
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
[init -> test-timer_accuracy]
Good: round 1, host measured 1000 ms, test measured 1008 ms
Good: round 2, host measured 2000 ms, test measured 2000 ms
Good: round 3, host measured 3000 ms, test measured 3006 ms
Good: round 4, host measured 4000 ms, test measured 3997 ms
Good: round 5, host measured 5000 ms, test measured 5003 ms
Good: round 6, host measured 6000 ms, test measured 6007 ms
Good: round 7, host measured 7000 ms, test measured 6995 ms
Good: round 8, host measured 8000 ms, test measured 7984 ms
Good: round 9, host measured 9000 ms, test measured 9005 ms

Run script execution successful.
```

Be patient, the test can take up to 40 seconds. The output looks just perfect.

2.7.4 Testing the dynamic linker

The ldso test exercises the functionality of the dynamic linker, including the execution of global constructors, transitive library dependencies, exception handling across libraries, and cross-library symbol resolution.

```
build/arm_v8a$ make run/test PKG=test-ldso KERNEL=hw BOARD=pine_a64lts
...
... build libc
...
[init -> test] child "test-ldso" exited with exit value 123

Run script execution successful.
```

Given that the init test succeeded, which already employed the dynamic linker, the result is not surprising but reassuring.

2.7.5 Packaging the kernel

All but the most basic run scripts leverage Genode's package management[1], often referred to as *depot*. A run script can conveniently incorporate packaged components into a system scenario via the `import_from_depot` function. When reviewing the various existing run scripts in the Genode source tree for this function, one can spot the following pattern.

```
import_from_depot [depot_user]/src/[base_src] \
                  [depot_user]/...
```

Each argument denotes a path of a depot archive. The `depot_user` and `base_src` are function calls. The `depot_user` function returns the name of the originator/creator of the given depot archive. It returns `genodelabs` by default and can be customized in the *etc/build.conf* file.

The `base_src` function returns the archive name of the so-called "base" source archive for a given combination of board and kernel. It contains the lowest-level and kernel-specific fundamentals any system scenario relies on, namely the kernel/core, the dynamic linker, and a timer driver. In order to execute any of the run scripts that follow this pattern, we need to create such a depot archive for our version of the kernel. When using the "hw" kernel, the `base_src` function can be found at *tool/run/boot_dir/hw*.

```
$ grep -r base_src tool/run/
...
run/boot_dir/hw:proc base_src { } { return "base-hw-[board]" }
...
```

In the case of our `pine_a64lts` board, the source archive would hence be named `base-hw-pine_a64lts`. The *tool/depot/create* tool can be used to populate the depot. Even though we have not yet provided any declaration for our base archive, let's call the tool and see how it breaks:

```
$ ./tool/depot/create x/src/base-hw-pine_a64lts UPDATE_VERSIONS=1 FORCE=1
Error: incomplete or missing recipe (x/src/base-hw-pine_a64lts)
```

The following things are worth noting about the command-line arguments.

- I supply `x` as depot user, which is just a dummy name that is good enough while pursuing the packaging work. Once the work is finished, it allows me to just remove the *depot/x/* directory and all testing artifacts are gone.

[1] https://genode.org/documentation/genode-foundations/22.05/development/Package_management.html

- The UPDATE_VERSIONS=1 argument tells the tool to automatically increase the version of the depot archive whenever the content differs from the previously packaged version. During the packaging work, I always set it to 1.
- The FORCE=1 argument tells the tool to perform all packaging steps from scratch instead of reusing artifacts from previous runs.

Now, let's address the error message. It tell us that the tool expected a so-called recipe for the given depot archive, which does not exist yet. The so-called recipes describe how a depot archive can be extracted from the source tree. They are searched in the *<repo>/recipes/* directories of all repositories. E.g., the recipe for the base source archive for the i.MX8 EVK board resides at *repos/base-hw/recipes/src/base-hw-imx8q_evk/*. This is a suitable template for out pine_a64lts recipe.

```
$ mkdir -p repos/allwinner/recipes/src/base-hw-pine_a64lts
$ cp -r repos/base-hw/recipes/src/base-hw-imx8q_evk/* \
        repos/allwinner/recipes/src/base-hw-pine_a64lts/
```

The directory hosts three files:

content.mk

This is Makefile snippet with rules for gathering the content of the archive from the source tree. The copied file, however, merely includes rules from a file called *base-hw_content.inc*. We can keep this line.

```
include $(GENODE_DIR)/repos/base-hw/recipes/src/base-hw_content.inc
```

used_apis

This file contains a list of APIs required to build a binary archive from the source archive. The copied template contains merely two lines, which we can keep that way. Naturally, the base-hw kernel requires the definitions of the generic Genode API (base API) and the supplements that are specific for the base-hw kernel (base-hw API).

```
base-hw
base
```

hash

The hash file tells the depot tools about the current version of the archive and draws the connection to the corresponding archive content by specifying a hash value.

```
2021-02-24 d122ddee70f0b075de8cec50a41c9f4783702e05
```

The hash value is computed over the entire content of the archive. Should the hash of a freshly created archive deviate from the hash stored at the recipe, we know that the version should better be updated. Of course, it would be tiresome to calculate such hash values manually. Thankfully, the depot tools do this job for us. While keeping in mind that the hash is most certainly wrong for our the pine_a64lts source archive, we leave it as is because we don't know any better value anyway at this point.

With the new source recipe in place, let's give the package creation another try.

```
$ ./tool/depot/create x/src/base-hw-pine_a64lts UPDATE_VERSIONS=1 FORCE=1
```

This time, the output looks different:

```
$ ./tool/depot/create x/src/base-hw-pine_a64lts UPDATE_VERSIONS=1 FORCE=1
created x/api/base/2021-02-22
created x/api/base-hw/2021-02-22

Error: CPU architecure for board pine_a64lts undefined

  missing file /.../repos/allwinner/board/pine_a64lts/arch
```

The tool has successfully created the API archives for the dependencies we stated in the *used_apis* file. However, the source-archive creation still backs out, missing the information of the board's CPU architecture. The CPU architecture dictates the subset of files of the base-hw repository that are relevant for the given board. This information is expected at the path printed by the error message. That's our call!

```
$ mkdir -p repos/allwinner/board/pine_a64lts
$ echo arm_v8a > repos/allwinner/board/pine_a64lts/arch
```

Upon the next attempt to create the source archive, the creation-process succeeds.

```
$ ./tool/depot/create x/src/base-hw-pine_a64lts UPDATE_VERSIONS=1 FORCE=1
created x/api/base/2021-02-22
created x/api/base-hw/2021-02-22
created x/src/base-hw-pine_a64lts/2021-03-04 (new version)
```

The last line tells us that the tool has detected the inconsistency of our recipe's hash file with the assembled archive content and has automatically adjusted the version (taking the current date) and hash in the hash file. Now it looks as follows:

```
$ cat repos/allwinner/recipes/src/base-hw-pine_a64lts/hash
2021-03-04 4589eb3b4d816d17a2f2a539031a54eff9dd3712
```

You may like to have a look at the resulting archive.

```
$ ls depot/x/src/base-hw-pine_a64lts/2021-03-04/
etc  include  lib  LICENSE  src  used_apis
```

Finally, let us check that it is possible to create a binary archive from our source archive by specifying `x/bin/arm_v8a/base-hw-pine_a64lts` as depot-archive path to the depot-create tool. To accelerate the build, we can append `-j8` as argument to enable the use of multiple CPUs.

```
$ ./tool/depot/create x/bin/arm_v8a/base-hw-pine_a64lts \
                      UPDATE_VERSIONS=1 FORCE=1 -j8
created x/api/base-hw/2021-02-22
created x/api/base/2021-02-22
created x/src/base-hw-pine_a64lts/2021-03-04
checking library dependencies...
...
... many build steps
...
    LINK     timer
created x/bin/arm_v8a/base-hw-pine_a64lts/2021-03-04
```

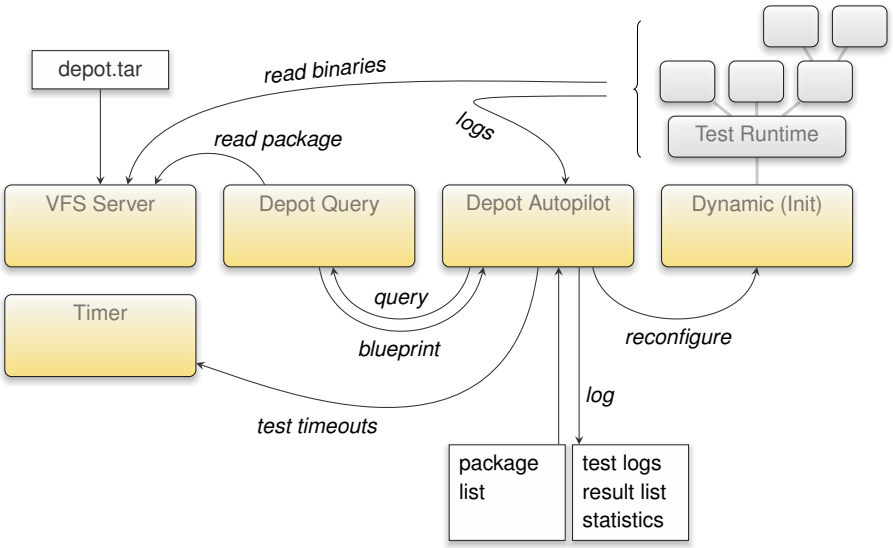
This time, the tool was satisfied with the current hash of our source recipe, the build process ran to completion, and we can inspect the results at the printed location within the depot.

```
$ ls -1 depot/x/bin/arm_v8a/base-hw-pine_a64lts/2021-03-04/
bootstrap-hw-pine_a64lts.o
core-hw-pine_a64lts.a
ld.lib.so
timer
```

2.7.6 Combined test suite of over 80 system scenarios

With the kernel packaged, a whole new world of run scripts opens up for us. The most intriguing one is *repos/gems/run/depot_autopilot.run*. It is a system that does not only execute a single test scenario but orchestrates the execution of more than 80 system scenarios one after the other. The init test we executed earlier is just one of these scenarios. Combined, the scenarios form the comprehensive test suite for Genode's base framework covering the following topics.

- Low-level data structures and allocators
- Parsing and generating XML, UTF-8
- Integration of Ada/SPARK with Genode's C++ API
- Publisher-subscriber mechanism
- Management of dynamic subsystems
- Fault detection mechanism
- Synthetic tests for low-level components and interfaces such as init, timer, VFS, block access, terminal
- VFS infrastructure
- C runtime (I/O, execve, fork, pthreads)
- Standard C++ library
- TCP/IP
- Network routing
- Tracing
- On-target deployment of depot packages



The diagram gives an overview of the architecture of the system scenario. It is described in great detail in the release documentation of Genode 18.11[1].

As a prerequisite for executing the depot-autopilot scenario, the depot packages for the whole arsenal of tests must be made available. We can instruct the build system to automatically create the depot content as needed, by enabling the following option at the *etc/build.conf* file:

```
RUN_OPT += --depot-auto-update
```

Furthermore, we need to make sure to have the following repositories enabled in the *etc/build.conf* file.

```
REPOSITORIES += $(GENODE_DIR)/repos/libports
REPOSITORIES += $(GENODE_DIR)/repos/dde_linux
REPOSITORIES += $(GENODE_DIR)/repos/gems
```

The dde_linux repository is solely needed for the TCP/IP stack ported from the Linux kernel (lxip). The gems repository hosts the depot-autopilot. With these precautions taken, we can kick off the *depot_autopilot.run* script as usual.

```
build/arm_v8a$ make run/depot_autopilot KERNEL=hw BOARD=pine_a64lts
```

[1] https://genode.org/documentation/release-notes/18.11#Automated_test_infrastructure_hosted_on_top_of_Genode

You will most likely encounter an error like the following.

```
Error: Ports not prepared or outdated:
  ada-runtime dde_linux expat gcov gmp libc lwip sanitizer stdcxx

You can prepare respectively update them as follows:
  /.../tool/ports/prepare_port ada-runtime dde_linux expat gcov \
                               gmp libc lwip sanitizer stdcxx
```

The printed ports of 3rd-party software are required. They can be imported into Genode's *contrib/* directory by executing the command as suggested by the error message.

Once the prepare_port command has completed, we can give the depot_autopilot.run script another try. This time, we can lay back and enjoy tons of build output scroll by, take a nip at a cup of coffee, maybe stretch our back a little, continue watching the build output, relax, not to forget to keep breathing. Have I mentioned looking at the build output?

When finally loading the resulting uImage on the board, we are greeted with a shocking message:

```
TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/uImage'.
Load address: 0x42000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##
         3.5 MiB/s
done
Bytes transferred = 8608974 (835cce hex)
## Booting kernel from Legacy Image at 42000000 ...
   Image Name:
   Image Type:   AArch64 Linux Kernel Image (gzip compressed)
   Data Size:    8608910 Bytes = 8.2 MiB
   Load Address: 40010000
   Entry Point:  40010000
   Verifying Checksum ... OK
   Uncompressing Kernel Image
Error: inflate() returned -5
Image too large: increase CONFIG_SYS_BOOTM_LEN
Must RESET board to recover
resetting ...
```

Just at the climax of our expectations, U-Boot's ELF loader went on strike. The ELF image is apparently too large. We take the mental note to adjust `CONFIG_SYS_BOOTM_LEN` and re-build and re-install U-Boot taking the steps of the [**?**].

To satisfy our urgent need of a reward for our patience during the build of the depot-autopilot system image, we can side step U-Boot's image-size constraint by loading a raw binary image. Since we need an ELF image instead of a uImage, we have to temporarily disable the `-include image/uboot` RUN_OPT in the *etc/build.conf* file and rebuild the image.

```
build/arm_v8a$ make run/depot_autopilot KERNEL=hw BOARD=pine_a64lts
....
```

A quick look at the result reveals that the uncompressed ELF image is quite large compared to the uImage file of 8 MiB. Here we can see the benefit of uImage files.

```
build/arm_v8a$ ls -lh var/run/depot_autopilot/boot/image.elf
-rwxrwxr-x 1 ... 49M ... var/run/depot_autopilot/boot/image.elf
```

The ELF image is swiftly converted to a raw binary placed in our TFTP directory.

```
build/arm_v8a$ /usr/local/genode/tool/current/bin/genode-aarch64-objcopy \
                  -Obinary \
                  var/run/depot_autopilot/boot/image.elf \
                  /var/lib/tftpboot/depot_autopilot.img
build/arm_v8a$ ls -lh /var/lib/tftpboot/depot_autopilot.img
-rwxrwxr-x 1 ... 49M ... /var/lib/tftpboot/depot_autopilot.img
```

Now, we can use the following U-Boot command to load it on the board.

```
=> bootp 0x40010000 10.0.0.32:/var/lib/tftpboot/depot_autopilot.img
BOOTP broadcast 1
DHCP client bound to address 10.0.0.178 (109 ms)
Using ethernet@1c30000 device
TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/depot_autopilot'.
Load address: 0x40010000
Loading: ##################################################################
...
         ##########################################################
         3.2 MiB/s
done
Bytes transferred = 51351552 (30f9000 hex)
```

... and run it!

```
=> go 0x40010000
## Starting application at 0x40010000 ...

kernel initialized
...
```

... massive amount of log output scrolls by for about 9 minutes ...

```
...
[init -> depot_autopilot] --- Finished after 519.179 sec ---
[init -> depot_autopilot]
[init -> depot_autopilot]  test-spark                 ok   0.239
[init -> depot_autopilot]  test-spark_exception       ok   0.161
[init -> depot_autopilot]  test-spark_secondary_stack ok   0.518
[init -> depot_autopilot]  test-block                 ok   1.124
[init -> depot_autopilot]  test-block_cache           ok   0.692
[init -> depot_autopilot]  test-clipboard             ok   3.676
[init -> depot_autopilot]  test-depot_query_index     ok   0.289
[init -> depot_autopilot]  test-ds_ownership          ok   0.227
[init -> depot_autopilot]  test-dynamic_config        ok   3.154
[init -> depot_autopilot]  test-dynamic_config_loader ok   3.211
[init -> depot_autopilot]  test-dynamic_config_slave  ok   2.640
[init -> depot_autopilot]  test-entrypoint            ok  40.117
[init -> depot_autopilot]  test-weak_ptr              ok   2.900
...
...
...
[init -> depot_autopilot]  test-xml_generator         ok   0.570
[init -> depot_autopilot]  test-xml_node              ok   1.028
[init -> depot_autopilot]  gcov                       ok  39.064
[init -> depot_autopilot]
[init -> depot_autopilot] succeeded: 82 failed: 0 skipped: 4
[init -> depot_autopilot]
[init] child "depot_autopilot" exited with exit value 0
```

The entire test suite succeeded with no errors!

It took 519 seconds. To cross-correlate this duration with the depot-autopilot test on the i.MX8q EVK board: The i.MX board takes 465 seconds, which makes the Pine-A64-LTS around 10% slower than the i.MX8 EVK. This is of course no benchmark to draw meaningful conclusions from. But the fact that both values are in the same ballpark reassures us that nothing fundamental (like the low-level CPU or memory configuration) went wrong with our port.

2.8 Device access from the user level

Genode's peripheral device drivers live outside the kernel and have the form of regular user-level components. This article presents how the device-hardware access works under these conditions, while taking the general-purpose I/O pins of the Pine-A64-LTS single-board computer as playground.

In the previous section, we reached a solid base line of functionality for the kernel and Genode framework on the Pine-A64-LTS board. Now it is time to turn out attention to the main topic of our SoC porting effort, which is the interaction with peripheral devices.

As a warm-up, there is no better peripheral than a general-purpose-I/O pin (GPIO) controller. It is a relatively simple device while enabling us to observe very satisfying physical effects. Despite the simplicity, we are faced with the two most important device-driver-related topics, namely accessing device registers and dispatching interrupts.

The investigation starts with the quest of finding a suitable pin at one of the various connectors present on the board. The board schematics[1] as found in the PINE64 Wiki[2] are our guide. While skimming the 19 pages of the document and glancing at the headlines above the very technical looking drawings, the so-called Euler connector at page 12 catches my attention because this name appears besides a prominently visible 34-pin header on the board.

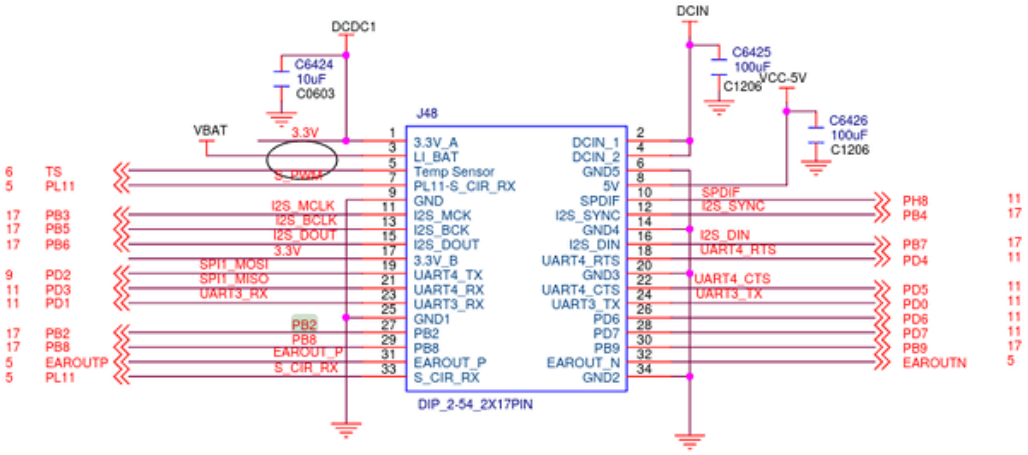


Figure 3

By looking at the schematics, it is easy to guess that the box with the 34 connectors corresponds to this pin header. The pins have labels, which give us clues about their

[1] https://files.pine64.org/doc/SOPINE-A64/PINE%20A64-TLS-20180130.pdf
[2] https://wiki.pine64.org/wiki/PINE_A64-LTS/SOPine

designated purposes. E.g., some pins are wired to fixed voltages like 5V or 3.3V or ground. Some others hint at specific functionality present in the SoC or another component on the board, e. g., those prefixed with I2S or UART or EAROUT. Some pins however, stand out by being named PB2, PB8, PD7 and such. The prefix P presumably stands for pin. Other usual signal-labeling schemes as found in schematics documents contain the pattern "IO" or "GPIO". Let's settle on the pin PB2 and see where this leads us. By searching the document for "PB2", we can see that the same signal appears at a box labeled "R18" (on the page for the Pi-2 connector). By searching for the ominous component "R18", we quickly learn that this label refers to the Allwinner SoC. So the pin is directly connected to the SoC. Did we ask for more? To sum up our findings, the following pins of the Euler connector are of interest to us:

- Pin 8: 5V
- Pin 27: PB2 (wired to the SoC)
- Pin 34: ground

The label PB2 has to have a meaning for the SoC, which is hopefully cleared up in the SoC's documentation[1]. For SoCs with no public documentation, the most compelling alternative source for such information are device-tree source (dts) files as usually provided by the SoC vendors for the Linux kernel and U-Boot. But let us save the device-tree topic for later. Being lucky that the Allwinner A64 SoC documentation is public, we can search it for "PB2", which brings us to Page 377, specifically to the description of a bit field named "PB2_SELECT" at a so-called "PB Configure Register 0".

3.21.2.1. PB Configure Register 0 (Default Value: 0x77777777)

Offset: 0x24			Register Name: **PB_CFG0_REG**
Bit	R/W	Default/Hex	Description
31	/	/	/
11	/	/	/
10:8	R/W	0x7	PB2_SELECT 000: Input 001: Output 010: UART2_RTS 011: Reserved 100: JTAG_DO0 101: SIM_VPPEN 110: PB_EINT2 111: IO Disable
7	/	/	/

Figure 4

[1] https://linux-sunxi.org/images/b/b4/Allwinner_A64_User_Manual_V1.1.pdf

The surrounding Section 3.21 “Port Controller(CPUx-PORT)" gives us the insights we need. PB2 is apparently one of the 10 input/output pins of Port B of the PIO peripheral, which presumably stands for Pin I/O. There exist plenty of device registers that are mirrored for different ports (B, C, D, ...).

2.8.1 Using a GPIO pin for sensing a digital signal

As a first exercise, let’s write a little program at *allwinner/src/test/pin_state/main.cc* that accesses the PB Configure Register 0.

```
#include <base/component.h>
#include <base/log.h>
#include <base/attached_io_mem_dataspace.h>
#include <util/mmio.h>

namespace Test {
    using namespace Genode;
    struct Main;
}


struct Test::Main
{
    Env &_env;

    Attached_io_mem_dataspace _pio_ds { _env, 0x1c20800u, 0x400u };

    struct Pio : Mmio
    {
        struct Pb_cfg0 : Register<0x24, 32>
        {
            struct Pb2_select : Bitfield<8, 3> { };
        };

        Pio(addr_t base) : Mmio(base)
        {
            log("PB2_SELECT: ", read<Pb_cfg0::Pb2_select>());
        }
    };

    Pio _pio { (addr_t)_pio_ds.local_addr<void>() };

    Main(Env &env) : _env(env) { }
};


void Component::construct(Genode::Env &env)
{
    static Test::Main main(env);
}
```

The following details are worth noting.

- The program comes in the form of a `Main` object as opposed to a `main()` function. To learn more about this structure, please refer to the article Genode's Conscious

C++ dialect[1].

- The `Env` interface allows the code to interact with the environment of the Genode component such as allocating memory, or opening a connection to a service provided by another component.
- The `_pio_ds` member opens a connection to an IO_MEM service and obtains a virtual-memory mapping of the specified range of the system bus. The numbers are taken from the Allwinner A64 manual.
- The `Pio` struct represents a memory-mapped I/O region, inheriting the `Mmio` type. The `Mmio` constructor takes the base address of the underlying device-register range as argument. The structs defined in the scope of the `Pio` struct mirrors the register structure of the memory-mapped I/O range: There exists a 32-bit wide register `Pb_cfg0` at offset 0x24.

  ```
  struct Pb_cfg0 : Register<0x24, 32>
  ```

  The bits 8 to 10 of this register correspond to the bit field `Pb2_select`.

  ```
  struct Pb2_select : Bitfield<8, 3> { };
  ```

  These declarations correspond one-to-one with the register definitions as found in the SoC user manual.
- In tho `Pio` constructor, we print the value of the `Pb2_select` bitfield by using the `Mmio::read` method.

  ```
  log("PB2_SELECT: ", read<Pb_cfg0::Pb2_select>());
  ```

  Note that the code is completely free of (often bug-prone) bit-masking/shifting operations.

To build the program, we have to accompany it with a *target.mk* file as follows.

```
TARGET := test-pin_state
SRC_CC := main.cc
LIBS   += base
```

Finally, we need to embed the program into a Genode system scenario. The following run script accomplishes this.

[1] https://genodians.org/nfeske/2019-01-22-conscious-c++

```
build { core init test/pin_state }

create_boot_directory

install_config {
  <config>
    <parent-provides>
      <service name="LOG"/>
      <service name="PD"/>
      <service name="CPU"/>
      <service name="ROM"/>
      <service name="IO_MEM"/>
      <service name="IRQ"/>
    </parent-provides>

    <default caps="100"/>

    <start name="test-pin_state">
      <resource name="RAM" quantum="1M"/>
      <route> <any-service> <parent/> </any-service> </route>
    </start>
  </config>
}

build_boot_image { core ld.lib.so init test-pin_state }

run_genode_until forever
```

When executing this run script, we can observe the following output:

```
kernel initialized
ROM modules:
 ROM: [000000004012c000,000000004012c17f) config
 ROM: [0000000040006000,0000000040007000) core_log
 ROM: [00000000401eb000,000000004022c260) init
 ROM: [000000004012d000,00000000401e4bd0) ld.lib.so
 ROM: [0000000040004000,0000000040005000) platform_info
 ROM: [00000000401e5000,00000000401ea0d0) test-pin_state

Genode 21.02-61-g446df00d0d8
2010 MiB RAM and 64533 caps assigned to init
[init -> test-pin_state] PB2_SELECT: 7
```

The PB2_SELECT bits have the value 7, which is the default value (I/O disable) according to the documentation. You may ask, what's behind those bits? The number

of connectors of a chip is physically limited by the space of the chip's package and the practicalities of PCB routing. To make one SoC applicable to a wide variety of products, SoC vendors implement a feature set much larger than the pin count would allow and leave the selection of a board-specific subset of those features to the board vendor. So different boards can use the same SoC but with different functionality exposed. The ultimate meaning of the physical pins is left to a software configuration. This multiplexing of pins to multiple SoC functionalities is often referred to as I/O muxing or pin muxing. On some SoCs, the I/O mux configuration is presented as a distinct device. On the Allwinner A64, it is part of the PIO device. For the pin PB2, the SoC provides the following options.

```
000: Input
010: UART2_RTS
100: JTAG_DO0
110: PB_EINT2
001: Output
011: Reserved
101: SIM_VPPEN
111: IO Disable    <- default
```

To sample the state of pin 27 of the Euler connector, we have to change the configuration value to 0 (input). Let's set the configuration value and validate that the change has the desired effect by changing the body of the `Pio` struct as follows.

```
struct Pb_cfg0 : Register<0x24, 32>
{
    struct Pb2_select : Bitfield<8, 3>
    {
        enum { IN = 0 };
    };
};

Pio(addr_t base) : Mmio(base)
{
    log("PB2_SELECT: ", read<Pb_cfg0::Pb2_select>());

    write<Pb_cfg0::Pb2_select>(Pb_cfg0::Pb2_select::IN);

    log("PB2_SELECT: ", read<Pb_cfg0::Pb2_select>());
}
```

Note the `enum` value definition for `IN`, which helps us to self-document the code as opposed to just writing the value 0. The output looks as expected. We read back the value that we have just written.

```
[init -> test-pin_state] PB2_SELECT: 7
[init -> test-pin_state] PB2_SELECT: 0
```

With the PB2 pin configured as input, let's see if we can observe a signal change at the Euler connector pin 27. The pin state is captured by the so-called PB Data Register (PB_DATA_REG) at offset 0x34. The register hosts one bit for each pin of the port B. For the PB2 pin, we have to poll bit 2. Or, to put it in other words:

```
...
struct Pb_data : Register<0x34, 32>
{
    struct Pb2 : Bitfield<2, 1> { };
};

Pio(addr_t base) : Mmio(base)
{
    write<Pb_cfg0::Pb2_select>(Pb_cfg0::Pb2_select::IN);

    while (true)
        log("PB2_STATE: ", read<Pb_data::Pb2>());
}
```

This gives us the following output:

```
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
```

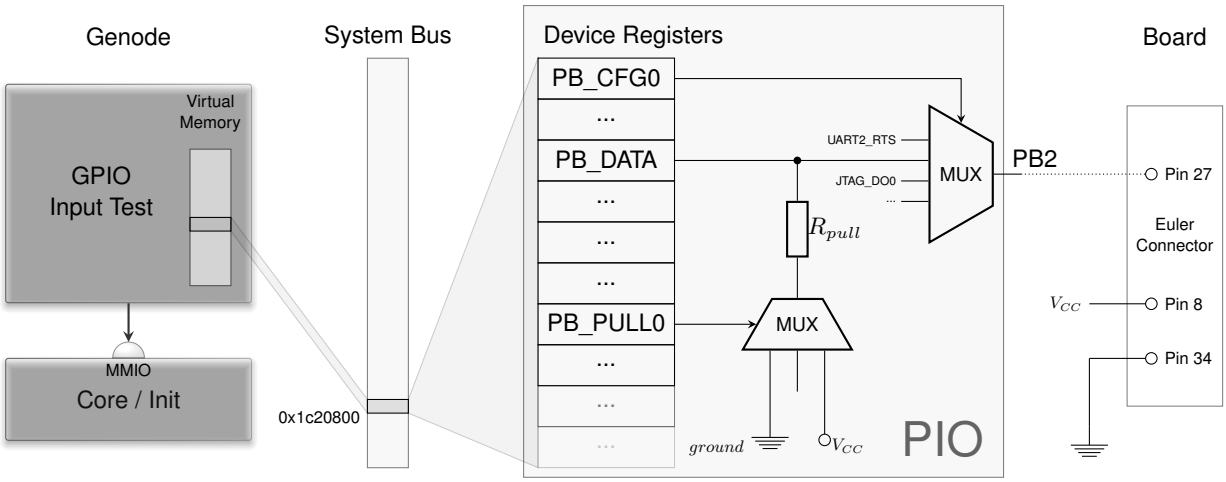
The pattern looks interesting, like if the PB2 pin is not quite sure about its state. For the experiment, let's try to connect the PB2 pin to ground. That is shorting the pins 27 (PB2) with 34 (GND). As a matter of courtesy, it is good to avoid connecting the pins directly but instead placing a resistor of a few hundred Ohm between both pins. Should we have done a mistake along our way and accidentally connect a 5V pin to GND, the current will flow nicely through our resistor instead of producing a short circuit. So what happens when connecting both pins?

```
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
[init -> test-pin_state] PB2_STATE: 0
...
```

That looks clean! What about connecting pin 27 (PB2) to pin 8 (5V)?

```
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
[init -> test-pin_state] PB2_STATE: 1
...
```

Isn't that wonderful?
The following picture summarizes our scenario.



The pin 27 of the Euler connector goes to the PB2 pin of the SoC. Via the PB_CFG0 register, we configure this pin to be used as general-purpose I/O pin reflected by bit 2 in the PB_DATA register. The register set of the PIO device unit is visible at physical address 0x1c20800 at the system bus. Thanks to the MMIO service of Genode's core, our test component becomes able to access this register range as part of its virtual address space. So what's this PB_PULL0 register shown in the picture?

This register can be used to prevent the fluctuating state when leaving the PB2 pin unconnected. Jargon speaks of *high impedance*, which sounds super educated but means the same thing. In real-world applications, this floating state is often not wanted. After all, digital means 0 or 1 but not maybe. Fortunately, the state can easily be avoided by connecting the PB2 pin via a very high resistor to ground (or 5V). This resistor pulls the floating potential *down* to ground (or *up* to 5V). Since this is such a common need, the SoC comes readily equipped with pull-down or pull-up resistors. We just need to enable either option, which can be done via the PB PULL Register 0 (PB_PULL0).

```
...
struct Pb_pull0 : Register<0x40, 32>
{
    enum { PULL_DOWN = 2 };

    struct Pb2 : Bitfield<4, 2> { };
};

Pio(addr_t base) : Mmio(base)
{
    ...
    write<Pb_pull0::Pb2>(Pb_pull0::PULL_DOWN);
    ...
}
```

With this little change, the output stays at 0 even when leaving the pin 27 (PB2) disconnected.

2.8.2 Driving an LED via a GPIO pin

Let's try the reverse, using the PB2 pin as a digital output signal. At this point, it is easy to connect the dots at the software side.

1. Configure the PB2_SELECT bits of the PB_CFG0 register to operate the pin in output mode, which is value 1.
2. Write the desired state to the bit 2 of the PB_DATA register.

The following code sets up the PB_CFG0 register and equips the `Pio` struct with a `toggle_pb2` method that reads the PB2 state from the PB_DATA register and writes back the inverted state.

```
struct Pio : Mmio
{
    struct Pb_cfg0 : Register<0x24, 32>
    {
        struct Pb2_select : Bitfield<8, 3>
        {
            enum { OUT = 1 };
        };
    };

    struct Pb_data : Register<0x34, 32>
    {
        struct Pb2 : Bitfield<2, 1> { };
    };

    Pio(addr_t base) : Mmio(base)
    {
        /* configure PB2 pin to output mode */
        write<Pb_cfg0::Pb2_select>(Pb_cfg0::Pb2_select::OUT);
    }

    void toggle_pb2()
    {
        bool const value = read<Pb_data::Pb2>();

        /* write back inverted value */
        write<Pb_data::Pb2>(!value);
    }
};
```

To let the test program blink the LED at a visible rate, we need a timer mechanism. Here, Genode's `Timer::Connection` becomes handy. By adding following few lines to the `Main` object, the `toggle_pb2` method gets called every 250 milliseconds.

```
#include <timer_session/connection.h>
...
struct Main
{
    Timer::Connection _timer { _env };

    void _handle_timeout(Duration)
    {
        _pio.toggle_pb2();
    }

    Timer::Periodic_timeout<Main> _timout_handler {
        _timer, *this, &Main::_handle_timeout, Microseconds { 250*1000 } };
};
```

Until now, the simple test scenario lack a timer service. So we have to extend the run script a bit.

1. Adding the timer service to the list of components to build.

   ```
   build { ... timer }
   ```

2. Adding a start node to the static system configuration.

   ```
   <start name="timer">
     <resource name="RAM" quantum="1M"/>
     <route> <any-service> <parent/> </any-service> </route>
     <provides> <service name="Timer"/> </provides>
   </start>
   ```

3. Routing the timer-session request by the test program to the timer service.

   ```
   <start name="test-pin_control">
     <resource name="RAM" quantum="1M"/>
     <route>
       <service name="Timer"> <child name="timer"/> </service>
       <any-service> <parent/> </any-service>
     </route>
   </start>
   ```

4. Adding the timer executable to the boot image.

   ```
   build_boot_image { ... timer }
   ```

At the hardware side, we need to connect an LED in series with a resistor dimensioned such that the potential difference over the LED will be approximately 2V. When connecting a 5V pin over the LED and the resistor to ground, the resistor should hence take away 3V. Most LEDs draw a current of 20mA. Hence, Ohm's law (R = U / I) tells us that the resistor should have a value of 3V / 0.02 A = 15O Ohm. Picking a higher resistor cannot hurt. It will just reduce the brightness of the LED. Long story short, a resistor of a few hundred Ohm should be fine.

If any electrical engineer is reading this and finds I'm writing nonsense, please contact me.

To see if the LED is able to light up in principle when connected with the resistor in series, the pins 8 (5V) and 34 (GND) become handy. The anode contact (the long one) of the LED must face the 5V side.

Now its time to bring software and hardware together by connecting the LED's anode to pin 27 (PB2) and starting the test program. The final setup looks like this. What's not captured in the photo is that the LED is indeed blinking.

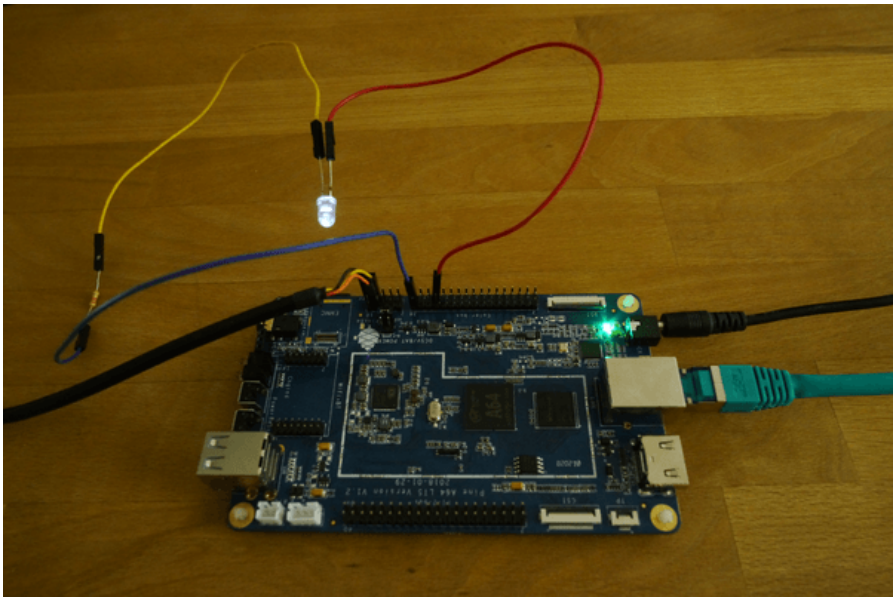


Figure 5

2.8.3 Responding to device interrupts

Besides sensing and driving digital signals, GPIO pins are often used as an interrupt source. So some external circuity can trigger a sporadic response by the software.

To explore the interrupt facility, let's first ignore the ARM GIC interrupt controller for a moment and just focus on the PIO device. In the PB_CFG0 register, the value 6 configures the pin as operating in PB_EINT2 mode. Whatever the meaning of the E or the 2, the pattern "INT" hints at what we want.

```
...
struct Pb_cfg0 : Register<0x24, 32>
{
    struct Pb2_select : Bitfield<8, 3>
    {
        enum { EINT2 = 6 };
    };
};
...
Pio(addr_t base) : Mmio(base)
{
    write<Pb_pull0::Pb2>(Pb_pull0::PULL_DOWN);

    write<Pb_cfg0::Pb2_select>(Pb_cfg0::Pb2_select::EINT2);
}
```

The PB External Interrupt Status Register (PB_EINT_STATUS_REG) reflects the interrupt state.

```
struct Pb_eint_status : Register<0x214, 32> { };
```

As an intermediate test, we can poll this register and see what happens when we connect the pin 27 (PB2) to pin 8 (5V). The polling loop can be directly added to the `Pio` constructor.

```
while (true)
    log("PB_EINT_STATUS: ", read<Pb_eint_status>());
```

After starting the program, we see the following output scrolling by.

```
[init -> test-pin_interrupt] PB_EINT_STATUS: 0
[init -> test-pin_interrupt] PB_EINT_STATUS: 0
[init -> test-pin_interrupt] PB_EINT_STATUS: 0
...
```

Once after connecting PB2 to 5V, the output changes to:

```
[init -> test-pin_interrupt] PB_EINT_STATUS: 4
[init -> test-pin_interrupt] PB_EINT_STATUS: 4
[init -> test-pin_interrupt] PB_EINT_STATUS: 4
...
```

The 4 corresponds to the bit 2 set, which is what we anticipated. The status bit never returns to the original state. To clear the bit, a 1 must be written to the status bit. This can be tested by slightly changing the while loop.

```
while (true) {
    if (read<Pb_eint_status::Pb2>()) {
        log("PB2 EINT status went high");
        write<Pb_eint_status::Pb2>(1);
    }
}
```

The scrolling log output is no more. Now, we see only one message each time we fiddle with the PB2 pin.

```
[init -> test-pin_interrupt] PB2 EINT status went high
```

The clearing of the interrupt status works as advertised.

Until now, we have observed the PIO device behavior via a polling loop, which is of course not in the spirit of using interrupts. To complete the scenario, we have to tell the PIO to inform the CPU's interrupt controller (GIC) whenever the EINT status goes high. The connection between the PIO and the GIC can be established via the PB External Interrupt Control Register.

```
struct Pb_eint_ctl : Register<0x210, 32>
{
   struct Pb2 : Bitfield<2, 1> { };
};
```

When setting bit 2 in this register, the GIC will see a device interrupt from the PIO device. The GIC interrupt numbers are documented in the Allwinner A64 manual at page 211. PB_EINT is interrupt number 43.

To obtain an interrupt in our component, we can use core's IRQ service as follows.

```
#include <irq_session/connection.h>
...

struct Test::Main
{
    ...

    enum { PB_EINT = 43 };

    Irq_connection _irq { _env, PB_EINT };


    unsigned _count = 0;

    void _handle_irq()
    {
        log("interrupt ", _count++, " occurred");

        _pio.clear_pb2_status();

        _irq.ack_irq();
    }

    Signal_handler<Main> _irq_handler { _env.ep(), *this, &Main::_handle_irq };

    Main(Env &env) : _env(env)
    {
        _irq.sigh(_irq_handler);
        _handle_irq();
    }
};
```

The following details about this code fragment are worth highlighting.

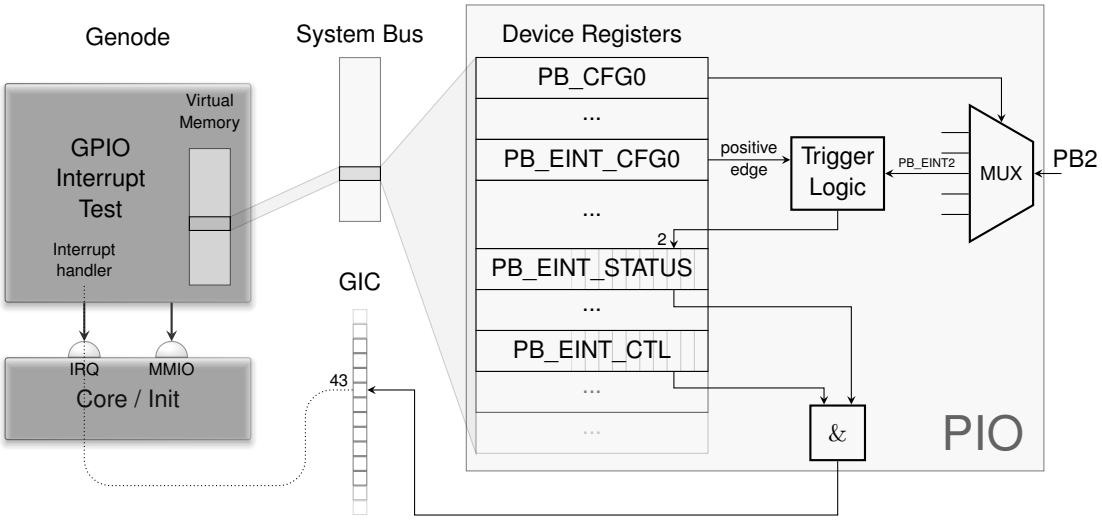
- The GIC interrupt number is passed as argument to the IRQ connection to core.
- Interrupts are delivered as signals. The `_irq_handler` is a signal handler that is registered at the IRQ session via the `_irq.sigh` method. Each time the signal occurs, the `Main::_handle_irq` method is executed.
- The `_pio.clear_pb2_status` method performs the clearing of the PB2 interrupt status.

```
struct Pio
{
      ...
      void clear_pb2_status()
      {
          write<Pb_eint_status::Pb2>(1);
      }
};
```

The `_irq.ack_irq` call acknowledges the interrupt at the GIC.

- The `_handle_irq` method is manually called once after registering the signal handler at the IRQ session. This pattern ensures that an initially pending interrupt that occurred just before the call of `_irq.sigh` is processed before the component goes into idle state.

The following illustration summarizes the scenario.



The exact conditions for triggering an interrupt can be configured for the pin using the PB External Interrupt Configure Register 0 (PB_EINT_CFG0). By default, the status goes to 1 as soon as a rising edge is detected. The other alternatives are falling edge, level-high (interrupt stays pending as long as the signal is high), level-low, and double edge (interrupt on any change of the signal).

Only if the bit 2 of the status register (PB_EINT_STATUS) and the bit 2 of the control register (PB_EINT_CTL) are set, the interrupt controller (GIC) receives an interrupt. This GIC interrupt (number 43) is propagated via core's IRQ service to our user-level component, which implements the interrupt handler.

Thanks to the interrupt mechanism, we can now respond to sporadic hardware events without active polling. When executing the scenario, we can see that a single

message occurs each time when fiddling with the PB2 pin. The system stays completely idle otherwise.

```
[init -> test-pin_interrupt] interrupt 0 occurred
[init -> test-pin_interrupt] interrupt 1 occurred
[init -> test-pin_interrupt] interrupt 2 occurred
[init -> test-pin_interrupt] interrupt 3 occurred
[init -> test-pin_interrupt] interrupt 4 occurred
```

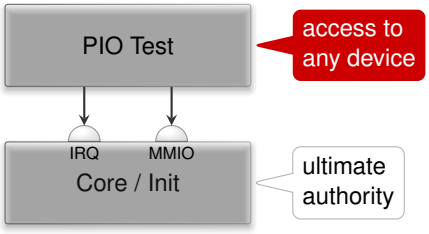
Pointers to the corresponding code The test programs described above can be found at the genode-allwinner[1] Git repository. The C++ code is located at src/test/pin_state/[2], src/test/pin_control/[3], and src/test/pin_interrupt/[4]. These test programs are accompanied with matching run scripts located at the run/[5] directory.

[1] https://github.com/genodelabs/genode-allwinner
[2] https://github.com/genodelabs/genode-allwinner/tree/master/src/test/pin_state
[3] https://github.com/genodelabs/genode-allwinner/tree/master/src/test/pin_control
[4] https://github.com/genodelabs/genode-allwinner/tree/master/src/test/pin_interrupt
[5] https://github.com/genodelabs/genode-allwinner/tree/master/run

2.9 One Platform driver to rule them all

In the previous section, we exercised direct-device access from user-level components. In Genode systems beyond such toy scenarios, however, it would be irresponsible to follow the path of allowing arbitrary drivers to access any device willy-nilly. Our call for discipline and rigidity is answered by the *(rising drum roll)* platform driver.

Let's recap the scenario of the previous article.

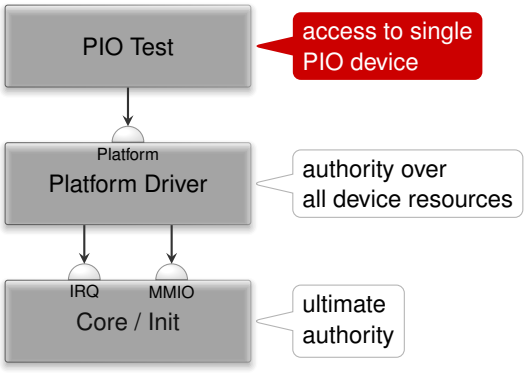


Our user-level test program created connections to core's services for accessing memory-mapped I/O registers and receiving notifications for device interrupts. The choice of physical register addresses and the GIC interrupt number was up to the test program. So in principle, our program could access any part of the platform by just requesting it. Hence, the mere fact that the driver code has the form of a regular user-level component does not buy us a security gain per se. To benefit from Genode's architecture, we need to rigidly limit the reach of each individual driver to the smallest-possible set of device resources.

Remember, even though we want to use drivers, we distrust them. Consequently, besides enforcing access control, we generally don't want to expose system-global information to such untrusted components, asking questions like: Does a driver even need to know the physical address of a memory-mapped I/O register? Does it need to know the GIC interrupt number of the device it is driving? The perhaps surprising answer is that - no! - many drivers can happily do their job without any knowledge about these technicalities. All a driver needs to know is *how to speak* to a device of a certain type, not where a particular instance of a device is located and how it is wired up. This principled approach leads to a clear-cut separation of driver logic from parametrization.

2.9.1 Platform driver

To separate the concerns of parametrization and access control from the device drivers, Genode employs the so-called *platform driver* as a level of indirection between core's services and the individual drivers. The platform driver has a global view over all device resources and follows a configured policy to partition those resources between its clients. Each session to the platform service can comprise potentially multiple devices, depending on the configured policy.



To integrate the notion the platform driver into our existing scenario of accessing general-purpose I/O pins via a to-be-developed PIO device driver, it is best to first sketch a run script that mirrors the picture above. We have to find a suitable name and location within our source tree for our designated driver component.

The naming of driver components within Genode follows the pattern

```
<device-or-platform>_<driver-type>_drv
```

For example, the `imx8_fb_drv` is a framebuffer (fb) driver for the `i.MX8` SoC. In our case of a PIO driver for the Allwinner A64 SoC, the name `a64_pio_drv` is a sensible choice.

Even though there is no strict convention of the directory where a driver is located, drivers usually reside in a subdirectory of *src/drivers/* that corresponds to the primary purpose of the driver. E.g., framebuffer drivers are located at *src/drivers/framebuffer/*. Our designated driver drives GPIO pins. So I settled on placing it at *src/driver/pin/a64/* within the genode-allwinner repository[1]. With the complicated naming-things-topics behind us, let's turn our attention to the run script, appropriately named *a64_pio_drv.run*.

1. Building the components including the platform driver along with our new custom driver.

   ```
   build { core init drivers/platform drivers/pin/a64 }
   ```

2. Creating a boot directory with the configuration of the init component.

[1] https://github.com/genodelabs/genode-allwinner

```
create_boot_directory

install_config {
  <config>
    <parent-provides>
      <service name="LOG"/>
      <service name="PD"/>
      <service name="CPU"/>
      <service name="ROM"/>
      <service name="IO_MEM"/>
      <service name="IRQ"/>
    </parent-provides>

    <default caps="100"/>
    ...

  </config>
}
```

At this time the scenario consists of only two components, namely the platform driver and our PIO driver. The <start> node for the platform driver is particularly interesting.

```
<start name="platform_drv">
  <resource name="RAM" quantum="1M"/>
  <provides> <service name="Platform"/> </provides>
  <route> <any-service> <parent/> </any-service> </route>
  <config devices_rom="config">
    <device name="pio">
      <io_mem address="0x1c20800" size="0x400"/>
      <irq number="43"/>
    </device>
    <policy label="a64_pio_drv -> ">
      <device name="pio"/>
    </policy>
  </config>
</start>
```

The routing rule states that the platform driver is permitted to open arbitrary sessions to core, including IRQ and IO_MEM. There are no restrictions.

The <provides> declaration states that this component offers a “Platform” service.

The <config> node tells the platform driver to determine the information about the present devices from the ROM module labeled “config”, which allows us to

specify both the device information and the policy inside the same <config> node. Without the `devices_rom="config"` attribute, the platform driver would request a "devices" ROM session instead. We declare the existence of a single "pio" device that features one memory-mapped I/O range and the GIC interrupt 43. You may recall those values from Section 2.8.

The <config> node also tells the platform driver about the access-control policy applied to clients that connect to the platform service. In the case at hand, we dictate that a client labeled as "a64_pio_drv → " gets access to the "pio" device. You may wonder about the trailing → part of the label. The part before the arrow is hard-wired by the parent of the `a64_pio_drv` and thereby reflects the identity of the client in a way that cannot be forged by the client. The part after the arrow is controlled by the client. The client can use this part to provide hints about the purpose of the session. So a client that creates multiple sessions to the same server can express the intention behind those sessions. In our case, this client-controlled part remains unused.

The <start> node for our designated PIO driver looks as follows.

```
<start name="a64_pio_drv">
  <resource name="RAM" quantum="1M"/>
  <route>
    <service name="ROM"> <parent/> </service>
    <service name="CPU"> <parent/> </service>
    <service name="PD">  <parent/> </service>
    <service name="LOG"> <parent/> </service>
    <service name="Platform"> <child name="platform_drv"/> </service>
  </route>
  <config/>
</start>
```

Let me bring the <route> node to your attention. In contrast to the wildcard rule <any-service> used for the platform driver, the rules for the PIO driver state explicit permissions. From these rules, we can immediately infer the potential reach of the component.

The driver is permitted to connect to the platform driver. That's what we want. It is also able to use core's ROM, CPU, PD, and LOG services, which provide the fundamental ingredients for executing the program.

Most importantly, no other service is reachable. In particular, the direct use of core's IRQ and IO_MEM is out of question. The only way to access a device is the platform driver that imposes its policy.

3. Building the boot image containing the ELF binaries for the components and executing the scenario.

```
build_boot_image { core ld.lib.so init platform_drv a64_pio_drv }

run_genode_until forever
```

For reference you can find a commit for this step here[1].

To exercise the interplay between the designated PIO driver with the platform driver, it is a good idea to transplant the test/pin_state[2] program of the previous section from the use of core's services to the use of the platform driver. The following snippet highlights the important changes.

```
#include <platform_session/device.h>
...

struct Pio_driver::Main
{
   Env &_env;

   Platform::Connection _platform { _env };

   Platform::Device _device { _platform };

   struct Pio : Platform::Device::Mmio
   {
      struct Pb_cfg0 : Register<0x24, 32>
      {
         ...
      };

      ...

      Pio(Platform::Device &device) : Mmio(device)
      {
         ...
      }
   };
   ...
   Pio _pio { _device };
   ...
};
```

- The API for using the platform driver becomes available via

[1] https://github.com/genodelabs/genode-allwinner/commit/febf53b8ad6819757eeeaf20eeb1b634ade0b668
[2] https://github.com/genodelabs/genode-allwinner/blob/pio/src/test/pin_state/main.cc

```
#include <platform_session/device.h>
```

- A session to the platform service is established by creating an instance of a `Platform::Connection` passing the Genode environment as argument.

  ```
  Platform::Connection _platform { _env };
  ```

  By passing the `_env`, we explicitly give our consent that the `Platform::Connection` can have global side effects such as the communication with the outside world.

- Access to one particular device of the platform session can be obtained by creating an instance of a `Platform::Device`.

  ```
  Platform::Device _device { _platform };
  ```

  When called with only the `Platform::Connection` as argument, the device refers to the first - and in our case only - device of the platform session. In cases where multiple devices appear grouped in one platform session, a second argument allows for the selection of the device.

- The memory-mapped registers of the PIO device are represented by a custom `Pio` type that inherits the `Platform::Device::Mmio` type.

  ```
  struct Pio : Platform::Device::Mmio
  ```

  The constructor takes a `Platform::Device` and an optional index as arguments.

  ```
  Pio _pio { _device };
  ```

  If no index is provided, it refers to the first <io_mem> resources as declared in the platform-driver's configuration.

- Thanks to the inherited `Platform::Device::Mmio` type, the individual registers can be accessed in the same way as we did in the previous article.

Note that in contrast to the previous examples, the code is void of physical addresses. Now, those addresses are the business of the platform driver only.

2.9.2 Session interfaces for accessing pins

We ultimately want to allow multiple programs to interact with different GPIO pins. So our PIO driver must evolve into a server component that allows clients to interact with pins. Analogously to how the platform driver safeguards the access to device resources by different - mutually distrusting - device drivers, the PIO driver's job will be the safeguarding of GPIO pins.

Traditionally, Genode features the "Gpio" session interface for this purpose. This interface allows a client to access an individual pin. Once assigned to a pin, the session grants the client the full responsibility for the pin. In particular the direction of the I/O pin is laid into the hands of the client. We later realized that the wiring and thereby the direction of a pin is ultimately a board-level decision. Wrongly operating an input pin in output mode can easily result in a short-circuit. Therefore, the client of an individual pin should better not be burdened with the responsibility to control the pin direction or pull resistors. To address this concern, it is best to split the roles of GPIO pins into clear-cut session interfaces. Those roles are:

1. The sensing of the state of a GPIO pin, e. g., detecting whether a button is pressed or not: operating a pin as an input signal. This role is now covered by the "Pin_state" session interface with the single RPC function

   ```
   bool state() const;
   ```

   By calling this function, the client can request the state of the pin. That's it.

2. Controlling the signal level of a pin: operating a pin as an output signal. This role is now addressed by the "Pin_control" session interface that provides an interface of only one rather unsurprising RPC function

   ```
   void state(bool);
   ```

3. Receiving a notification of a change of the signal level of a GPIO pin: operating a pin as an interrupt source. This role can be represented by Genode's existing IRQ session interface - the same interface as provided by Genode's core for GIC interrupts.

2.9.3 PIO device driver

The A64 PIO driver implements the three session interfaces outlined above. It resides at src/drivers/pin/a64[1] within the genode-allwinner repository. The accompanied README covers the details about its use and configuration.

Similar to how the platform-driver configuration declares device resources like IRQs and memory-mapped I/O regions, the PIO driver's configuration declares pins.

[1] https://github.com/genodelabs/genode-allwinner/tree/master/src/drivers/pin/a64

```
<config>
  <out name="led"    bank="B" index="2" default="on"/>
  <in  name="button" bank="H" index="8" pull="up" irq="edges"/>
  ...
</config>
```

Here we see the declaration of an output pin named “led” and an input pin “button”. The `bank` and `index` denote the physical location of the pin at the SoC. Further pin parameters are expressed as attributes. For example, in the absence of a “Pin_control” client for the “led”, the led is set to state “on” according to the `default` attribute.

Since the A64 PIO device subsumes GPIO functionality as well as I/O MUX functionality, the driver also offers the selection of pin functions beyond <in> and <out>.

For reference, the commit for the driver implementation can be found here[1]. A few technical tidbits and caveats I encountered during its development are worth sharing:

Device-register interaction

The actual interplay of the driver with the hardware registers is completely covered by the code found in pio.h[2]. Genode’s `Mmio` framework API makes this code strikingly simple, almost self-describing. There is no manual bit fiddling to be found, thanks to the wonderful `Register_array`.

Code organization

I deliberately split the code into a boring and an interesting part.

The boring part models the SoC-specific terminology as a bunch of corresponding C++ types. In types.h[3] one can find types for any term we deal with - however boring it is. Most of these types have a local `Value` type that is as rigid as possible. E.g., the `Pull` type contains an enum with the values DISABLE, UP, and DOWN as the `Value` type. The degrees of freedom mirror the information found in the SoC manual. Each type is equipped with a class function `from_xml` that encodes the knowledge of how values of the type relate to XML representation. Some of the types go as far as deliberately disabling any means to construct instances of the type without using `from_xml` by deleting the default constructor. This way, program-global invariants of the type can be enforced at a single place. The boring code makes up the biggest part of the driver. This is good because with “boring” I mean simple and easy to assess for correctness.

The interesting part lives in the main.cc[4] file where all the strings are coming together.

[1] https://github.com/genodelabs/genode-allwinner/commit/11ec1ae2b8fe39de4f8059eb1e28c9a1bcb263a5
[2] https://github.com/genodelabs/genode-allwinner/blob/master/src/drivers/pin/a64/pio.h
[3] https://github.com/genodelabs/genode-allwinner/blob/master/src/drivers/pin/a64/types.h
[4] https://github.com/genodelabs/genode-allwinner/blob/master/src/drivers/pin/a64/main.cc

Stumbling blocks

Quite a bit of time went wasted because of silly mistakes of mine.

Sometimes I went too hastily over the SoC documentation without double checking. In particular, I allowed myself be become misled by a table in the SoC documentation[1] at page 376 where I wrongly identified patterns that do not exist. In one part of the table, the symbol `n` *seemingly* refers to a zero-indexed value corresponding to GPIO banks in alphabetic order. Some lines below (at the Pn_INT_*) definitions, the `n` refers only to a few banks, namely `B`, `G`, `H`. I wrongly assumed the same linearity of register layouts to apply for both parts of the table. In reality, `n` must just be read as a shorthand of "some value". Note to myself: Double check my assumptions each time I'm overconfident that *I got it*.

Because of my prolonged intimacy with pin 2 at bank B, I lost sight of the other banks, in particular the fact that each bank is wired up with a distinct GIC interrupt. Once I tried to receive interrupts for pin 8 at bank H, I first struggled to get the interrupt mechanism to work, until I realized that bank H interrupts end up at GIC IRQ 53, not 43. In fact, the "pio" device in the platform driver configuration now looks like this:

```
<device name="pio">
  <io_mem address="0x1c20800" size="0x400"/>
  <irq number="43"/> <!-- Port B -->
  <irq number="49"/> <!-- Port G -->
  <irq number="53"/> <!-- Port H -->
</device>
```

Implementation of dynamic re-configurability

For maintaining the internal data model of the pin-state configuration, the driver employs Genode's `List_model` utility. By using this utility, the creation and updating of such a data model from XML data becomes very simple. It comes down to providing hook functions for creating, destroying, matching, and updating model items.

It is worth noting that the driver configuration is not static but it can be dynamically adjusted during runtime. So in principle, we can attain a blinking LED by the sole means of re-configuring the driver.

2.9.4 Dynamic configuration testing

Wait what!?

[1] https://linux-sunxi.org/images/b/b4/Allwinner_A64_User_Manual_V1.1.pdf

If blinking an LED by reconfiguring the PIO driver sounds as irresistible to you as to me, follow me for a moment.

For test-driving the dynamic configuration handling of components like the A64 PIO driver, there exists a handy utility component called *dynamic_rom*, which provides a ROM service that feeds the client with different version of ROM content over time. Here is how a <start> node of a dynamic_rom server looks like.

```
<start name="dynamic_rom">
  <resource name="RAM" quantum="1M"/>
  <provides> <service name="ROM"/> </provides>
  <route>
    <service name="Timer"> <child name="timer"/> </service>
    <any-service> <parent/> </any-service>
  </route>
  <config>
    <rom name="config">
      <inline description="LED off">
        <config>
          <out name="led" bank="B" index="2" default="off"/>
        </config>
      </inline>
      <sleep milliseconds="1000"/>
      <inline description="LED on">
        <config>
          <out name="led" bank="B" index="2" default="on"/>
        </config>
      </inline>
      <sleep milliseconds="1000"/>
    </rom>
  </config>
</start>
```

The <rom> node within its configuration defines a PIO <config>. After 1 second, the <config> is replaced with a new version where the `default` attribute of the <out> pin is toggled. After one more second, the first <config> becomes active again.

The remaining piece of the puzzle is feeding the ROM provided by the dynamic_rom server as config ROM to the a64_pio_drv driver. This can be achieved by the following routing rule in the <start> node of the a64_pio_drv component.

```
<start name="a64_pio_drv">
  ...
  <route>
    <service name="ROM" label="config">
      <child name="dynamic_rom"/> </service>
    ...
  </route>
</start>
```

By wiring up the driver configuration to the dynamic_rom we can see the LED happily blinking even without any "Pio_control" client present.

The dynamic_rom server is handy utility in many testing situations. Besides issuing time-triggered configuration updates, it can be used to mock system-state changes that are normally driven by real components or sensory input that is difficult to fabricate manually.

2.9.5 Cascaded authorities

Similarly to the configuration concept of the platform driver, the pin-declarations of the PIO driver configuration are followed by a policy part of the configuration that associates clients with pins.

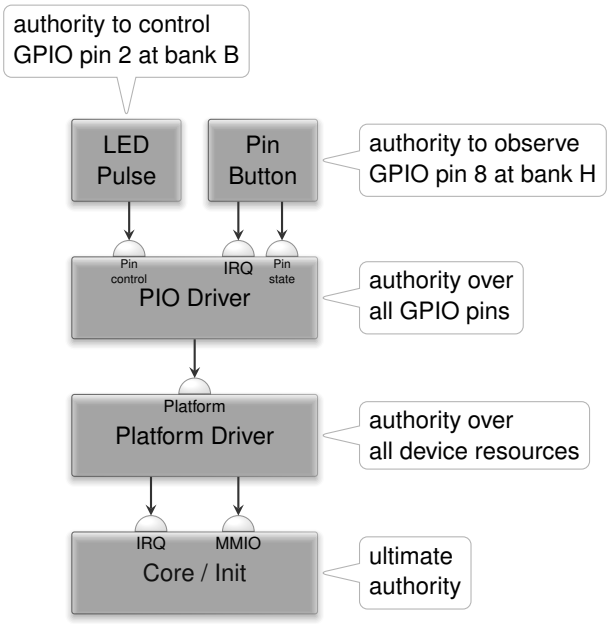
```
<config>
  ...
  <policy label_prefix="pin_event ->" pin="button"/>
  <policy label_prefix="pin_pulse ->" pin="led"/>
</config>
```

This configuration assigns the "led" pin to the program "pin_pulse", and the "button" to the program "pin_event". Note that - like the pin declarations - these assignments can be dynamically changed by the means of configuration updates.

The "pin_pulse" component uses the "Pin_control" session to drive the digital signal of an LED with a pulse-width-modulated pattern. Effectively, the program toggles the LED 200 times per second while adjusting the relation of the durations of the low and high signal levels over time. The result is a nice breathing effect.

The "pin_event" component watches the state of a pin using a combination of an IRQ session and a "Pin_state" session. Each time when the signal changes, an IRQ is triggered, which prompts the component to obtain the pin state by calling `Pin_state::state`.

The component composition of the scenario looks as follows.



The higher up we get, the less influential the components become. Whereas the kernel has ultimate authority over everything, the reach of the pin-pulse component is limited to the control of the output signal of a single GPIO pin only.

As indicated by the coloring of the components, policy and mechanisms are nicely separated. The pin-pulse component does not even know which pin it is driving. It merely contains the logic needed to modulate the PWM pattern on a digital output signal. At the bottom end of the picture, the core / kernel component does have no interpretation of physical device addresses or IRQ numbers. It is indifferent regarding GIC IRQ number 43 and free from policy. The policy is encapsulated in the forms of the platform and PIO driver components, each respectively applying a policy at a *useful* level of abstraction.

2.9.6 Integrated test scenario

The final version of the a64_pio_drv.run[1] script contains the combinations of the various fragments discussed above. It test-drives the dynamic re-configurability of the PIO driver along with the "Pin_state", "Pin_control", and IRQ session interfaces.

For the test of the GPIO input, I selected pin 8 of bank H. This pin is accessible at the Euler connector at pin 10 of the Pine-A64-LTS board. The board has a button labeled "power" just besides the reset button. Although this "power" button is connected to the AXP803 power management chip, it doesn't appear to have any effect when pressed while the board is on. According to the board schematics[2], the button

[1] https://github.com/genodelabs/genode-allwinner/blob/master/run/a64_pio_drv.run
[2] https://files.pine64.org/doc/SOPINE-A64/PINE%20A64-TLS-20180130.pdf

happens to be also wired to pin 5 of the smaller 10-pin Euler header. I figured that I can thereby feed the button state to the GPIO pin H8 be connecting pin 5 of the small Euler header with pin 10 of the large Euler header. The signal is active-low, which can be explained by the schematics that shows that the button pulls the PWR_ON signal to ground when pressed. Long story short, with this wiring in place, the power button can be observed via GPIO H8. The GPIO pin B2 can be connected to an LED as we did for the test/pin_control example described in the previous article.

2.10 Taking Linux out for a walk

In preparation of the porting of drivers from Linux to Genode, we have to gather knowledge about the drivers' natural habitat. This article goes through the steps of building a custom Linux system that is tailored to a driver of our choice.

Most of Genode's device drivers are not written from scratch specifically for Genode but ported from time-tested code bases such as the Linux kernel. In the case of most ARM SoC drivers, the driver code is provided by the SoC vendors and is often the only reference of how the hardware works. Given the staggering complexity of the devices, their rapid product cycles, and the diversity of employed IP cores, developing the drivers from scratch is out of question.

Motivation In the past, we sometimes directly jumped into the driver-porting work guided by mere intuition, transplanting chewable bites of Linux kernel code into Genode components, and reanimating the code to life. In some cases like our our first port[1] of the Intel GPU driver, we were lucky with bringing up a driver on Genode without even testing the driver code on Linux beforehand. However, this remained a rare exception. In most cases, especially when a predictable success and development schedule is desired, we had to engage with the building and integration of custom Linux systems as part of the driver-porting work. Even though jumping thought the hoops of Linux is additional work, it gives us three invaluable benefits.

First, it gives us **reassurance** how the driver works on the reference hardware. We can set our expectations regarding the feature set, stability, and performance. Once the driver is ported, we can reflect on the success by the means of benchmarking the driver in both systems Linux and Genode.

Second, it allows us to **study** and **instrument** the driver's interaction with the user-level interfaces and the hardware. For getting hold of the userland's interaction with the driver, it is useful to exercize the driver interfaces - think of `ioctl` calls - by walking the beaten tracks.

Third and most importantly, it allows for **cross-correlating** the behavior of the driver in its natural environment against its execution within an isolated Genode component. We have to mimick Linux kernel APIs so that the driver is happy, after all. The comparison of the driver running in both environments is instructive for crafting the driver environment.

Picking a tangible goal Among the various classes of peripherals, *network* devices are an attractive target for porting a first driver. In contrast to the PIO device discussed in Section 2.9, network devices are *complicated enough* to get a tangible benefit out of the porting approach. Conversely, network drivers are *simple enough* to not get

[1] https://genode.org/documentation/release-notes/10.08#Gallium3D_and_Intel_s_Graphics_Execution_Manager

overwhelmed. Furthermore, network connectivity is *easy to test* in an automated way, avoiding distractions from fiddly manual workflows. Finally, the *reward* of enabling networking support is quite substantial. A single driver opens up a whole world of Genode system scenarios.

In the following, we will walk towards a minimally complex custom Linux system by taking the following steps.

- Learning how to use U-Boot to start an *arbitrary* Linux-based OS,
- Using a custom initrd based on Busybox,
- Booting a *custom-built* kernel,
- Getting the *network device* to work,
- Stripping down the *kernel configuration*,
- Making the findings *reproducible*

2.10.1 Bootstrapping Linux using U-Boot

For our development work flow, we'd like to retain the convenience of network boot as explored in Sections 2.2 and 2.3. Booting Linux differs from booting a single binary as we did so far. For grokking those differences, it is nice to rely on known-to-work ingredients.

Remembering from Section 2.2 how well the image of the Armbian[1] distribution worked on the board, we know where to look. There are various ways for accessing the content of the image. When using Gnome, you may be able to mount the image by clicking on it. In my case, I loop-mounted the image manually. First, looking in the partition table to find out where the file system starts.

```
$ fdisk -l Armbian_21.05....img
Disk Armbian_21.05....img: 1,7 GiB, 1782579200 bytes, 3481600 sectors
...
Armbian_21.05....img1  8192 3481599 3473408  1,7G 83 Linux
```

It starts at block `8192`. Given the usual block size of 512 and the help of a multiplication device, we get to know the byte offset as an argument for loop-mounting the file system.

```
$ python -c "print 8192*512"
4194304
$ sudo mount -oloop,offset=4194304 Armbian_21.05....img /mnt
```

[1] https://www.armbian.com

In the file-system's */boot/* directory, the files of interest are:

Image is a symlink to the Linux kernel. In my case, it refers to *vmlinuz-5.10.34-sunxi64*, which is a file of 21 MiB.

config-5.10.34-sunxi64 is the kernel configuration, which will become handy once we build the kernel from source. Save it for later.

dtb/allwinner/ is a directory of so-called device-tree files, which we will cover in a minute. For now, it suffices to know that each dtb file contains a hardware description of a specific board. Among the 48 dtb files present in the directory, I identified the one for the Pine-A64-LTS board via

```
$ ls -1 /mnt/boot/dtb/allwinner/ | grep a64 | grep lts
sun50i-a64-pine64-lts.dtb
```

initrd.img-5.10.34-sunxi64 is the initial ram disk used for bootstrapping the userland.

For booting Linux, we have to supply the kernel (Image), the dtb file (pine64.dtb) for our board, and the initrd via our TFTP directory and can use the following U-Boot commands to bring the combination of the pieces to life. The addresses are picked such that they do not overlap, using U-Boot's default load address for the kernel.

1. Load the kernel.

   ```
   => bootp 0x42000000 10.0.0.32:/var/lib/tftpboot/Image
   ```

2. Load the initial RAM disk.

   ```
   => bootp 0x41000000 10.0.0.32:/var/lib/tftpboot/initrd
   ```

3. Load the device-tree file

   ```
   => bootp 0x41f00000 10.0.0.32:/var/lib/tftpboot/pine64.dtb
   ```

4. Modifying the device tree to supply kernel parameters. First, telling U-Boot's FDT tool where the location of our DTB data.

   ```
   => fdt addr 0x41f00000
   ```

   Making space for adding additional content.

   ```
   => fdt resize 0x1000
   ```

Modifying the `chosen` device-tree node that contains the kernel parameters such as the location of the initrd (start and end address)

```
=> fdt chosen 0x41000000 0x41996388
```

or the kernel command line

```
=> fdt set /chosen bootargs "rdinit=/bin/sh"
```

5. Start the kernel by specifying the kernel's address and DTB address. We can specify `-` as the second (initrd) argument because we already tell the kernel the location of the initrd via the `chosen` DTB node.

```
=> booti 0x42000000 - 0x41f00000
```

It goes without saying that manually executing this sequence of commands would be error-prone and unnerving. Fortunately, U-Boot allows us to store a sequence of commands in an environment variable, like so:

```
=> setenv lx 'command; another command; ...'
```

The variable named `lx` now contains the sequence of the commands separated by semicolons. The value of `lx` variable can be made persistent via the mechanism discussed in Section 2.2.2.

```
=> saveenv
```

The sequence of commands stored in the variable `lx` can be executed via the `run` command:

```
=> run lx
```

The machinery takes over...

```
ethernet@1c30000 Waiting for PHY auto negotiation to complete........ done
...
Using ethernet@1c30000 device
TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/Image'.
Load address: 0x42000000
Loading: #################################################################
         #################################################################
         ....
         ####################################
         1.6 MiB/s
done
Bytes transferred = 32020992 (1e89a00 hex)
BOOTP broadcast 1
DHCP client bound to address 10.0.0.178 (3 ms)
Using ethernet@1c30000 device
TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/initrd'.
Load address: 0x41000000
Loading: #################################################################
         ....
         ###############
         2.2 MiB/s
done
Bytes transferred = 10052488 (996388 hex)
BOOTP broadcast 1
DHCP client bound to address 10.0.0.178 (2 ms)
Using ethernet@1c30000 device
TFTP from server 10.0.0.32; our IP address is 10.0.0.178
Filename '/var/lib/tftpboot/pine64.dtb'.
Load address: 0x41f00000
Loading: ##
         1.2 MiB/s
done
Bytes transferred = 28418 (6f02 hex)
## Flattened Device Tree blob at 41f00000
   Booting using the fdt blob at 0x41f00000
EHCI failed to shut down host controller.
   Loading Device Tree to 0000000049ff5000, end 0000000049ffffff ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[    0.000000] Linux version 5.10.34-sunxi64 ...
...
... a few hundred lines of boot messages ...
...
[    6.593071] Freeing unused kernel memory: 5888K
[   37.865818] vcc-1v2-hsic: disabling
```

Pressing enter...

```
#
```

A root shell awaits our commands. Could we ask for more?

2.10.2 A custom initrd based on Busybox

With the clarity gained about the network boot process of Linux via U-Boot, we can now stepwise replace each of the three ingredients.

First, we replace Armbian's initrd with a custom-built RAM disk based on Busybox[1]. This will give us a functional, reproducible, and customizable userland to play with while being ten times smaller than the initrd we scraped off the Armbian image. Since we ultimately strive for a minimalistic Linux kernel with all batteries for our board included and dynamic modules switched off, we don't need the initrd as a carrier of kernel modules after all.

Presuming that a regular ARM tool chain and the cpio utilities are installed (e. g., the Debian packages *gcc-aarch64-linux-gnu* and *cpio*), the following steps produce a custom initrd from scratch.

1. Download and extract BusyBox

   ```
   $ wget https://busybox.net/downloads/busybox-1.29.3.tar.bz2

   $ tar xjf busybox-1.29.3.tar.bz2
   $ mkdir build-busybox-aarch64
   $ cd busybox-1.29.3
   ```

2. Configure BusyBox

   ```
   $ make O=../build-busybox-aarch64 defconfig
   $ make O=../build-busybox-aarch64 menuconfig
   ```

   Check the following setting in the menu:

   ```
   [*] Setting ->  Build static binary (no shared libs)
   ```

3. Compile BusyBox with the installed cross-compile tool chain

   ```
   $ cd ../build-busybox-aarch64
   $ make CROSS_COMPILE=aarch64-linux-gnu- install -j6
   ```

[1] https://www.busybox.net/

4. Useful setups for the ram disk

```
$ cd <busbox-build-dir>/_install
```

Create fstab by editing *etc/fstab* as follows.

```
none /proc  proc   defaults 0 0
none /sys   sysfs  defaults 0 0
```

Enable DNS resolution via a predefined name server.

```
$ echo "nameserver 1.1.1.1" > etc/resolv.conf
```

Create a run-level script using mdev by creating a *etc/init.d/rcS* file with the following content.

```
#/bin/sh

mkdir -p /proc
mkdir -p /sys
mkdir -p /var/shm
mkdir -p /var/run
mkdir -p /var/tmp
mkdir -p /tmp
mkdir -p /home/tc
mkdir -p /root
mkdir -p /dev

/bin/mount -t tmpfs mdev /dev
mkdir /dev/pts
/bin/mount -t devpts devpts /dev/pts

echo "Mount everything ${1}"
/bin/mount -a

echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s
echo "Finished"
/bin/sh
```

5. Create the initial ram disk using *cpio*

```
$ find . | cpio -H newc -o | gzip > ../initrd
```

After replacing Armbian's initrd with our custom initrd in the TFTP directory, the U-Boot command for tweaking the `chosen` device-tree node must be slightly adjusted to the different size.

2.10.3 Vendor kernel source

For most SoCs, the chip vendor provides a vendor-blessed version of the Linux kernel source somewhere on the internet. Those so-called vendor kernels are usually a somewhat sour compromise. On the one hand, they contain the know-how of the vendor regarding the device drivers. The vendor kernels are usually tailored well to the hardware. On the other hand, the "tailoring" does not always follow the written and unwritten rules of Linux kernel development, standing in the way of cleanly integrating the vendor code into the upstream kernel. Some vendors don't even try. Hence, after the vendor kernel for a specific chip is publicly released, it must be assumed a dead branch of kernel development, receiving no further love. In our situation - where we are after the vendor's drivers - we have to take the SoC's vendor kernel as our reference.

In the case of the Allwinner A64 SoC, the vendor kernel[1] is based on Linux as old as version 3.10. However, thanks to the tremendous efforts of the Sunxi[2] open-source community that works independently from the SoC vendor, the A64 is fully supported by the upstream Linux kernel by now. The instructions[3] for building a kernel suitable for the A64 SoC come down to compiling the Linux kernel for the AARCH64 architecture using its default configuration.

The bottom line is that we can pick a recent kernel from https://kernel.org and go with it.

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.12.1.tar.xz
...
$ tar xf linux-5.12.1.tar.xz
```

This will extract the kernel source to the *linux-5.12.1* subdirectory. In the following, we will refer to this directory as LX_DIR. Let's remember the path in a environment variable. That's just for convenience.

```
export LX_DIR=$PWD/linux-5.12.1
```

[1] https://linux-sunxi.org/Pine64#BSP
[2] https://linux-sunxi.org
[3] https://linux-sunxi.org/Pine64#Linux_Kernel

2.10.4 Building and booting a custom-built kernel

Let's give the default kernel configuration a try. For hygienic reasons, I prefer using a build directory separate from the source tree. Let's call the build directory LX_BUILD_DIR pointing at some place that does not exist yet.

```
export LX_BUILD_DIR=$PWD/lx_build
```

When using the Linux build system targeting the AARCH64 architecture, one always has to specify `ARCH=arm64` as argument to make. Note that this argument even changes the output of `make help`. So we should apply it consistently. Let's stuff it into an environment variable so that we don't need to repeat it over and over again.

```
export ARCH=arm64
```

Similarly to the `ARCH` argument, we need to tell the kernel's build system about the tool chain used for cross compiling the kernel. In our case, we want to use Genode's tool chain, which is usually installed at */usr/local/genode/tool/<version>/bin/* and prefixed with `genode-aarch64-`. The kernel's build system expects this information as CROSS_COMPILE argument or environment variable.

```
export CROSS_COMPILE=/usr/local/genode/tool/21.05/bin/genode-aarch64-
```

With these precautions taken, we can initialize the build directory with the default configuration via

```
make -C $LX_DIR O=$LX_BUILD_DIR defconfig
```

The kernel configuration can be found at the .config file inside the build directory. To get an idea what a "default" configuration entails, the number of enabled options is telling.

```
$ grep =y $LX_BUILD_DIR/.config | wc -l
2482
```

I translate this number to *not processable by my mind*.

A further feel of defeat sets in when trying to compare the default configuration with the *config-5.10.21-sunxi64* found on the Armbian image.

```
$ diff $LX_BUILD_DIR/.config config-5.10.21-sunxi64 | wc -l
10363
```

We seem to be out of luck if we think of correlating both configurations with each other.

Without further ado, the kernel image can be built as follows.

```
$ make -C $LX_BUILD_DIR Image -j8
  ...

  LD      vmlinux
  SORTTAB vmlinux
  SYSMAP  System.map
  OBJCOPY arch/arm64/boot/Image
```

On my laptop, this takes about 20 minutes. As indicated by the build-system messages, the resulting kernel `Image` can be found at the *arch/arm64/boot/*. It has a size of 32 MiB.

By the way, the kernel's build system offers a number of other useful targets such as a compressed kernel image. It is worth taking a look at the options.

```
$ make -C $LX_BUILD_DIR help
```

Here, we can learn that there exists a convenient target for creating DTB files.

```
$ make -C $LX_BUILD_DIR dtbs
```

So we can pick the dtb file matching our board from *arch/arm64/boot/dts/*, in our case *allwinner/sun50i-a64-pine64-lts.dtb* replacing the third magical puzzle piece of the boot process by a variant created from source.

When trying out the fresh baked kernel on the board, we can see the kernel booting up, showing the kernel log over the serial line, and presenting us with the root shell. We have a solid ground to walk on!

Next up, let us turn our attention to networking. The first impulse is to issue `ifconfig` to see the presence of network devices. Well, the bad news is that there are none.

We can look at this problem from several angles.

- Grep'ing the *.config* file for patterns like ALLWINNER, SUNXI, NET, etc.
- Wandering through the menus of `make menuconfig` on the hunt for cues.

- Comparing the boot logs of the Armbian kernel and the custom built kernel, looking out for network-related messages.

Of these points, the last one is probably the least futile. However, there is an alternative route towards success and happiness: Let's have a closer look at the device tree for our board.

2.10.5 Device-tree treasure trove

The so-called device trees[1] are semi-formal descriptions of a hardware platforms, which may consist of several peripherals, buses, interrupt controllers, clocks, and so on. In the context of Linux, it serves two purposes. First, it fills the gap of dynamic device discovery on platforms where devices cannot be probed in a reliable way at runtime. This applies for most ARM SoCs. By looking at the device tree, the kernel learns which drivers are to use. Second, the device tree parametrizes the individual drivers. This allows drivers to stay clear from vendor-specific parameters hard-coded in the source code. By taking parameters from the device tree, drivers can more easily re-targeted to other SoC revisions.

For our aspiration to port drivers to Genode, device trees are a blessing. Since they are curated by the SoC vendors, they contain a form of hardware documentation that can often not be found anywhere else. Moreover, in contrast to sketchy documentation - if available at all - the information present in the device trees can be assumed to be correct because it plays a role in driving the hardware. In a way, device trees look like a social engineering trick to wrest hardware docs from vendors. To lift the treasure, we have to look in the source tree at *arch/arm64/boot/dts/*.

```
$ find $LX_DIR/arch/arm64/boot/dts
```

We find almost 800 files there. But the forest is well organized. It is straight-forward to spot the vendor and narrow the search. In my case:

```
$ find $LX_DIR/arch/arm64/boot/dts | grep allwinner
...
$ find $LX_DIR/arch/arm64/boot/dts | grep allwinner | grep pine
...
$ find $LX_DIR/arch/arm64/boot/dts | grep allwinner | grep pine | grep lts
.../arch/arm64/boot/dts/allwinner/sun50i-a64-pine64-lts.dts
```

When looking into this file, we see that it uses the C preprocessor to include a bunch of includes. To get the entire picture, we have to apply the C-preprocessing step.

[1] https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf

```
$ cpp -I $LX_DIR/include -x assembler-with-cpp -P \
      $LX_DIR/arch/arm64/boot/dts/allwinner/sun50i-a64-pine64-lts.dts \
      > flat_pine64lts.dts
```

The `-x assembler-with-cpp` argument is needed to prevent the C preprocessor from misinterpreting lines with a leading # as preprocessor directive. The `-P` argument removes line markers from the output. It is useful to save the result in a file (*flat_pine64lts.dts*), which we can consult at any time later. For the Pine-A64-LTS board, the file comprises 1600 lines of insights: the relationships of many important numbers to human-readable terminology.

2.10.6 Enabling network support

Refreshed from studying the device tree for our board, now is a good time to come back to the topic of enabling the networking for our board. By skimming over the flattened dts file, the following node featuring the term "ethernet" catches our attention.

```
emac: ethernet@1c30000 {
 compatible = "allwinner,sun50i-a64-emac";
 syscon = <&syscon>;
 reg = <0x01c30000 0x10000>;
 interrupts = <0 82 4>;
 interrupt-names = "macirq";
 resets = <&ccu 13>;
 reset-names = "stmmaceth";
 clocks = <&ccu 36>;
 clock-names = "stmmaceth";
 status = "disabled";
 mdio: mdio {
  compatible = "snps,dwmac-mdio";
  #address-cells = <1>;
  #size-cells = <0>;
 };
};
```

Let me draw your attention to the two lines defining a property called `compatible`. Those properties denote the driver that should be used to talk to this piece of hardware. It actually draws a direct connection to the source code. To put the device-tree node in other words:

To use ethernet on our board, the kernel configuration must include the source code related to "allwinner,sun50i-a64-emac" and "snps,dwmac-mdio".

Let's start with "allwinner,sun50i-a64-emac".

```
$ grep -r "allwinner,sun50i-a64-emac" $LX_DIR/drivers/net
.../stmmac/dwmac-sun8i.c: { .compatible = "allwinner,sun50i-a64-emac",
```

Apparently, a sledgehammer is sometimes the perfect device for singling out a needle from a haystack. Let's not waste too much time with looking at the code. The only information important to us is the name of the compilation unit `dwmac-sun8i.c`. As a matter of convention, the corresponding object file is usually present in the makefile for the subsystem.

```
$ grep "dwmac-sun8i.o" $LX_DIR/drivers/net/ethernet/stmicro/stmmac/Makefile
obj-$(CONFIG_DWMAC_SUN8I)  += dwmac-sun8i.o
```

What a revelation! We have just drawn the connection from a node found it a device tree to a kernel-configuration option. When looking at the kernel's `.config` file, it becomes clear that the kernel does not drive the ethernet device with builtin drivers because the driver is configured as a module.

```
$ grep CONFIG_DWMAC_SUN8I $LX_BUILD_DIR/.config
CONFIG_DWMAC_SUN8I=m
```

Often, kernel options have dependencies. To get a picture of the dependencies of the `CONFIG_DWMAC_SUN8I` driver, one can consult the accompanied *Kconfig* files or show the `Help` for the corresponding item in the menus of the kernel's `make menuconfig` interactive configuration tool.

```
$ make -C $LX_BUILD_DIR menuconfig
```

Use search (`/`) to search for the desired option (e. g., CONFIG_DWMAC_SUN8I), learn about the location of the option in the menu hierarchy, and look out for dependencies not marked with `y`. In our concrete case, the two options STMMAC_ETH and STMMAC_PLATFORM must be enabled to satisfy DWMAC_SUN8I.

To enable the options, one may use the interactive menu config tool. But I prefer a non-interactive way that can be scripted. There exists a handy tool at *scripts/config* in the kernel source tree that allows us to enable and disable options, like so:

```
$ $LX_DIR/scripts/config --file $LX_BUILD_DIR/.config \
       --enable STMMAC_ETH --enable STMMAC_PLATFORM --enable DWMAC_SUN8I
```

The tool merely sets or removes individual options. It must be followed by an invocation of `make olddefconfig` to resolve possible inconsistencies and dependencies.

```
$ make -C $LX_BUILD_DIR olddefconfig
```

By looking at the resulting `.config` we get the reassurance that all dependencies are indeed met.

```
$ grep CONFIG_DWMAC_SUN8I $LX_BUILD_DIR/.config
CONFIG_DWMAC_SUN8I=y
```

The steps above may appear long-winded. But in contrast to hit-and-miss juggling of kernel configurations, the steps form a deterministic and explainable process.

With the driver for the "allwinner,sun50i-a64-emac" device-tree node covered, we are left to repeat the process for the "snps,dwmac-mdio" node. It turns out this node is covered by STMMAC_PLATFORM already.

As a convenient way to quickly test-drive our network-equipped Linux kernel, the kernel can be instructed to issue a DHCP request as part of the boot process. This can be enabled by specifying the argument `ip=dhcp` to the kernel command line. For reference, when using U-Boot's `fdt` command, the `chosen` device-tree node can be adjusted as follows:

```
=> fdt set /chosen bootargs "rdinit=/bin/sh ip=dhcp"
```

Once all driver dependencies are resolved, `ifconfig` reports the Ethernet device with its IP address. At this point we know that network packets were successfully transmitted in both directions.

```
/ # ifconfig
ifconfig: /proc/net/dev: No such file or directory
eth0      Link encap:Ethernet  HWaddr 02:BA:FE:7B:59:38
          inet addr:10.0.0.178  Bcast:10.0.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:39

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
```

2.10.7 Stripping down the kernel configuration

Even though the custom built kernel works in principle, the situation it not satisfactory. First, the kernel image of 32 MiB carries a lot of excess weight when considering that

we merely want to take the network driver for a joyride. Second, we learned that the default configuration deliberately excludes features that we deem interesting.

To overcome this situation, we have to dive into the world of Linux kernel configuration more than knee deep. But first, let's backup the known-to-work `.config` file.

```
cp $LX_BUILD_DIR/.config $LX_BUILD_DIR/config_works
```

In addition to the `defconfig` build target that results in 2482 options enabled in the `.config` file, the kernel's build system also features a `tinyconfig` target that leaves only 323 options enabled. By using this configuration, the `Image` build time goes down from 20 minutes to only 1.5 minutes with the image weighting only 1.6 MiB. That is much more to my taste!

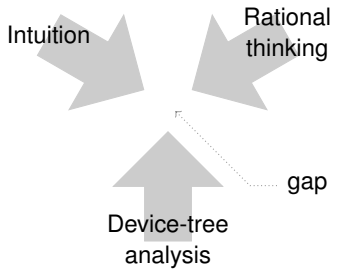
The only downside of the `tiny` kernel is that it does not show a life sign when booted on our board. But that is expected because the configuration does not accommodate any specific SoC out of the box. In order to bring it to life, all we need to do is to enable the right options, don't we? The first goal should be to find the options needed to get the boot log over the serial line. Once we accomplish that, we can turn our attention to supporting our initrd. Once that works, we can revive the networking support.

At this point, we know that serial output works with the `defconfig` kernel. The 2482 options enabled for the defconfig have to contain the right ones. Most obviously, we need to enable the platform support for our SoC (ARCH_SUNXI). We can let our intuition guide us for a bit. Which kernel options found in the defconfig could possibly contribute to serial output of the kernel console? Searching the *config_works* file for terms like "SERIAL".

```
$ grep =y $LX_BUILD_DIR/config_works | grep SERIAL
...
```

E.g., SERIAL_EARLYCON looks certainly useful to have.

The second strategy for finding the right options is the device-tree-driven approach we used for the network device. Searching the DTS file for "serial" leads us to a node referring to "snps,dw-apb-uart", which relates to *8250_dw.c* and *8250_early.c*, which in turn brings CONFIG_SERIAL_8250_DW and CONFIG_SERIAL_8250_CONSOLE to our attention. Fast forward, the following options can be directly inferred from the device tree: SERIAL_8250, SERIAL_8250_16550A_VARIANTS, SERIAL_8250_DW, SERIAL_8250_CONSOLE.



Looking from above, looking from below, however we look, there is still a gap of config options to fill. To uncover the missing options, we can apply brute force, bisecting the configurations as illustrated by the following commands:

1. Create a fresh tinyconfig

```
$ make -C $LX_BUILD_DIR tinyconfig
```

2. Enable options we already know we need for sure

```
$ $LX_DIR/scripts/config --file $LX_BUILD_DIR/.config \
   --enable SERIAL_8250 ...
```

3. Enable half of the candidates found in the backed up defconfig

```
$ grep =y $LX_BUILD_DIR/config_works |\
   head -n 1200 |\
   sed "s/=y//" |\
   xargs -ixxx $LX_DIR/scripts/config \
                  --file $LX_BUILD_DIR/.config --enable xxx
```

   This command adjusts the *.config* file by enabling the first 1200 config options we find enabled in the *config_works* file, which are the upper half of the enabled options.

4. Sanitize the *.config* file

```
$ make -C $LX_BUILD_DIR olddefconfig
```

5. Build the kernel and copy the resulting image the TFTP directory

```
$ make -C $LX_BUILD_DIR dtbs Image -j8 \
  && cp $LX_BUILD_DIR/arch/arm64/boot/Image /var/lib/tftpboot/
```

6. Boot the board. If the kernel shows a life sign over serial, we know that all the needed options are covered by the first 1200 ones. Otherwise, we know that a missing option is somewhere beyond those 1200. So we can continue with the first step while cutting the search space in half for each iteration.

For example, in the case of the Pine-A64-LTS board, I found that the first 1200 options sufficed for the serial output. So I went for the first 600 options in the next iteration.

Since the search space is cut into half in each iteration, resolving the mystery of missing kernel options comes down to about 10 iterations and a bit of patience. Using this process, I uncovered the need for CONFIG_PRINTK, CONFIG_BINFMT_ELF, or BLK_DEV_INITRD, which look obvious in hindsight but are very unlikely to find by the means of grep, intuition, or device-tree analysis.

By following this process, I eventually came up with a kernel has networking and serial output enabled. The bisecting work is not taxing but rather mechanic. It is nice knowing that it leads to predictable success. With about 500 options enabled and an image size of 4.2 MiB (uncompressed), the resulting kernel is a workable basis for the upcoming porting and instrumentation work.

2.10.8 Making the findings reproducible

As with any Genode-related working topic, I'm trying to make the essence of the above findings easily reproducible, for others and me. This way, the next developer can pick up a topic where I left it.

To download the kernel using Genode's ports tool, we can start with the following initial ports file placed at *allwinner/ports/a64_linux.port*. The prefix `a64` refers to the name of the Allwinner SoC, which expresses our intent that the downloaded version of the Linux kernel is blessed for the use of this particular SoC.

```
LICENSE   := GPLv2
VERSION   := 5.12.1
DOWNLOADS := a64_linux.archive

URL(a64_linux) := https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-$(VERSION).tar.xz
SHA(a64_linux) := 123
DIR(a64_linux) := src/linux
```

The SHA hash is not known at this point, just putting an arbitrary number 123 there. The accompanied *allwinner/ports/a64_linux.hash* hash file can be created with a made-up number.

```
456
```

With the port-description file and hash file in place, we can give the `a64_linux` port a try.

```
genode$ ./tool/ports/prepare_port a64_linux
a64_linux  download https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.12.1.tar.xz
Error: Hash sum check for a64_linux failed
```

Even though the hash-sum check predictably failed, the download of the archive succeeded. It can be found in the *genode/contrib/* directory in a subdirectory named after the port file.

```
genode$ ls -lh contrib/a64_linux-456.incomplete/linux-5.12.1.tar.xz
... 113M ... contrib/a64_linux-456.incomplete/linux-5.12.1.tar.xz
```

The correct SHA hash value is just one invocation of `sha256sum` away:

```
genode$ sha256sum contrib/a64_linux-456.incomplete/linux-5.12.1.tar.xz
c0fc1cf...fe5f37  contrib/a64_linux-456.incomplete/linux-5.12.1.tar.xz
```

Now we can replace the dummy value 123 in the port description file with the correct value and retry the `prepare_port` call.

```
genode$ ./tool/ports/prepare_port a64_linux
a64_linux  extract linux-5.12.1.tar.xz (a64_linux)
a64_linux  generate a64_linux.hash
Error: allwinner/ports/a64_linux.port is out of date, expected 155f...
```

This time, the extraction step succeeded. However, the port tool rightfully argues about the port hash, which is a hash over the port description file. The hash ensures that port is consistent with the Genode source tree. This can be conveniently updated using the ports/update_hash tool.

```
genode$ ./tool/ports/update_hash a64_linux
generate a64_linux.hash
```

With the hash updated, the next attempt to `prepare_port` succeeds:

```
genode$ ./tool/ports/prepare_port a64_linux
a64_linux  extract linux-5.12.1.tar.xz (a64_linux)
a64_linux  generate a64_linux.hash
genode$ ls contrib/a64_linux-155f8b01cd911f42f23178571d70a2220612b634/
a64_linux.hash  linux-5.12.1.tar.xz  src
```

The *src/linux/* subdirectory contains the source tree of the kernel.

```
genode$ ls contrib/a64_linux-155f8b01cd911f42f23178571d70a2220612b634/src/linux/
arch     CREDITS        fs       Kbuild   LICENSES     net      security  virt
block    crypto         include  Kconfig  MAINTAINERS  README   sound
certs    Documentation  init     kernel   Makefile     samples  tools
COPYING  drivers        ipc      lib      mm           scripts  usr
```

Finally, to conserve the information about configuring and building a Linux kernel tailored to our porting work, I added a Genode build target at allwinner/src/a64_linux/target.mk[1] / target.inc[2], which applies the kernel configuration and builds the kernel image. Thanks to this *target.mk* file, the custom Linux kernel can be built from within Genode's build directory via:

```
build/arm_v8a$ make a64_linux
```

[1] https://github.com/genodelabs/genode-allwinner/blob/master/src/a64_linux/target.mk
[2] https://github.com/genodelabs/genode-allwinner/blob/master/src/a64_linux/target.inc

2.11 Pruning device trees

We briefly touched the treasure trove called device trees in the previous section. To leverage the wealth of information for the development and porting of Genode device drivers, this article introduces a handy new tool set.

As summarized in the previous article, device-tree files as found at Linux source tree under *arch/<arch>/boot/dts/* provide both a structural description of an SoC and parametrization data for individual device drivers. It goes without saying that this information is extremely valuable. On the other hand, the encoding of the information in the form of so-called *Devicetree Specification* (PDF[1]) files is not ideal for us.

The authors of DTS files anticipate a monolithic kernel where a global view of the system is natural. In contrast, Genode fosters a strict separation of drivers from each other where each driver gets to see only a tiny part of the picture. With a DTS file of more than 1600 lines (as for the Pine-A64-LTS) board given, it is really hard to see to see clear lines of responsibilities between drivers. This is where Genode's tool at *tool/dts/extract* comes into play. Just for reference, usage information are provided by executing the tool without arguments.

Let's assume we have generated an all-encompassing DTS file *flat_pine64lts.dts* for our board via the C preprocessor as described in Section 2.10.5.

The *tool/dts/extract* utility allows us to generate a dot graph from the source, which can be processed by the Graphviz[2] dot tool to generate a PNG file.

```
tool/dts$ ./extract --dot-graph flat_pine64lts.dts > pine64.dot
tool/dts$ dot -Tpng pine64.dot > pine64.png
```

[1] https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf

[2] https://graphviz.org/

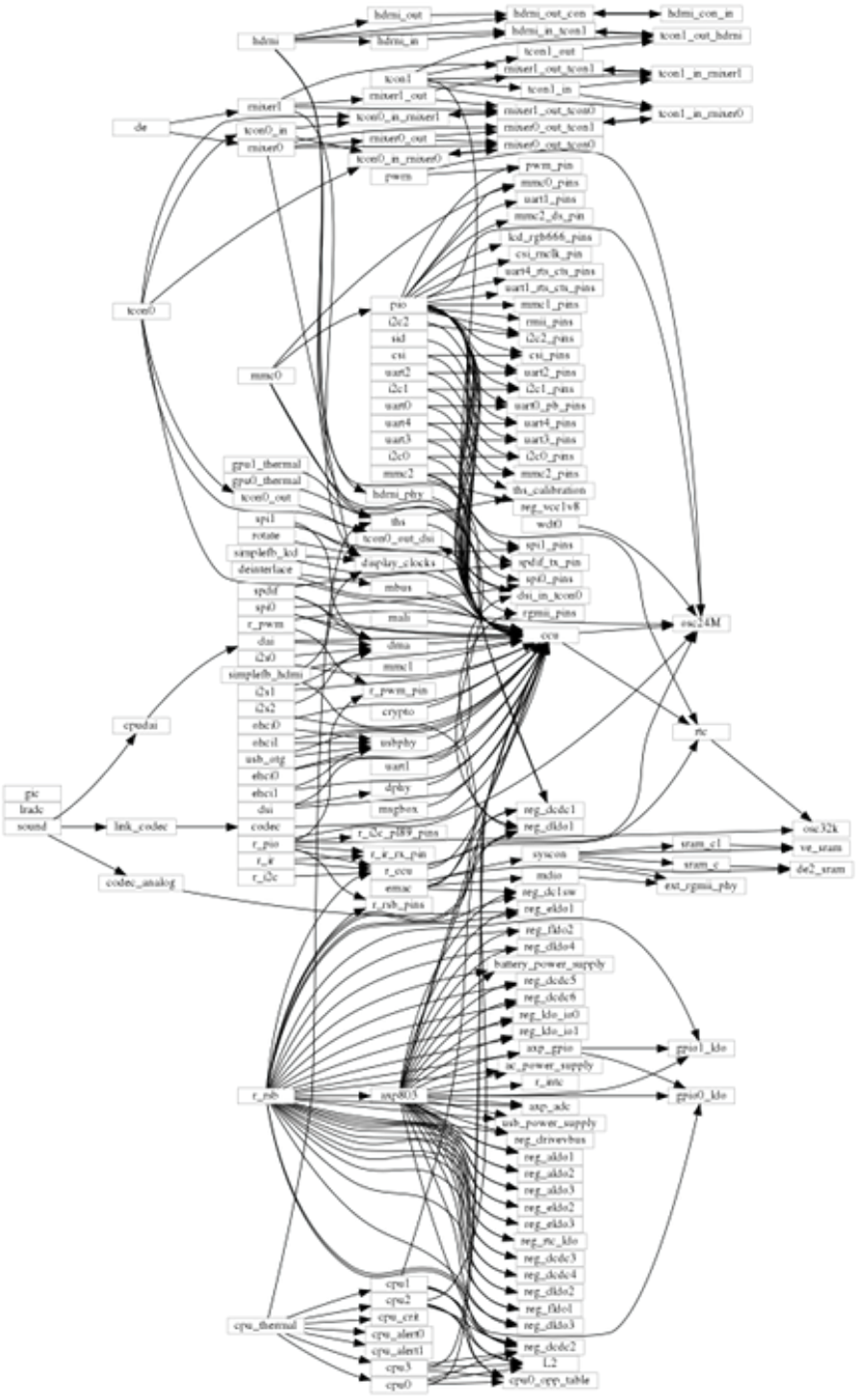


Figure 6: Does this count as generative art?

Even though the picture presents only a tiny fraction of the information present in the DTS file - neither any properties, nor device addresses, nor unlabeled nodes are shown - it is too overwhelming to be useful.

Let's say we are interested in the porting of the ethernet driver. In the previous article we already manually walked the DTS tree and spotted the corresponding node along the way. With the `-labels` option, the *extract* tool provides a convenient way to get an overview of the nodes present in the tree.

```
tool/dts$ ./extract --labels flat_pine64lts.dts
...
uart1 /soc/serial@1c28400
spi0_pins /soc/pinctrl@1c20800/spi0-pins
ve_sram /soc/syscon@1c00000/sram@1d00000/sram-section@0
reg_aldo1 /soc/rsb@1f03400/pmic@3a3/regulators/aldo1
emac /soc/ethernet@1c30000
uart2 /soc/serial@1c28800
lradc /soc/lradc@1c21800
...
```

Each line presents a label accompanied with the corresponding path of the device node. Of course, the command is best combined with grep.

```
tool/dts$ ./extract --labels flat_pine64lts.dts | grep ether
emac /soc/ethernet@1c30000
mdio /soc/ethernet@1c30000/mdio
ext_rgmii_phy /soc/ethernet@1c30000/mdio/ethernet-phy@1
```

The `emac` label should ring a bell from the previous article. To find out about the interaction of the emac device with the other parts of the device tree, the extract tool allows us to generate a new DTS tree with only a selection of devices and their dependencies present.

```
tool/dts$ ./extract --select emac flat_pine64lts.dts > emac.dts
```

From the more of 1600 lines of the original DTS file, the result comprises only about 200 lines. This amount of information can be digested without choking.

```
tool/dts$ wc -l emac.dts
213 emac.dts
tool/dts$ ./extract --dot-graph emac.dts > emac.dot
tool/dts$ dot -Tpng emac.doc > emac.png
```

With a few final manual tweaks of the layout parameters, one can get a picture as nice as this.

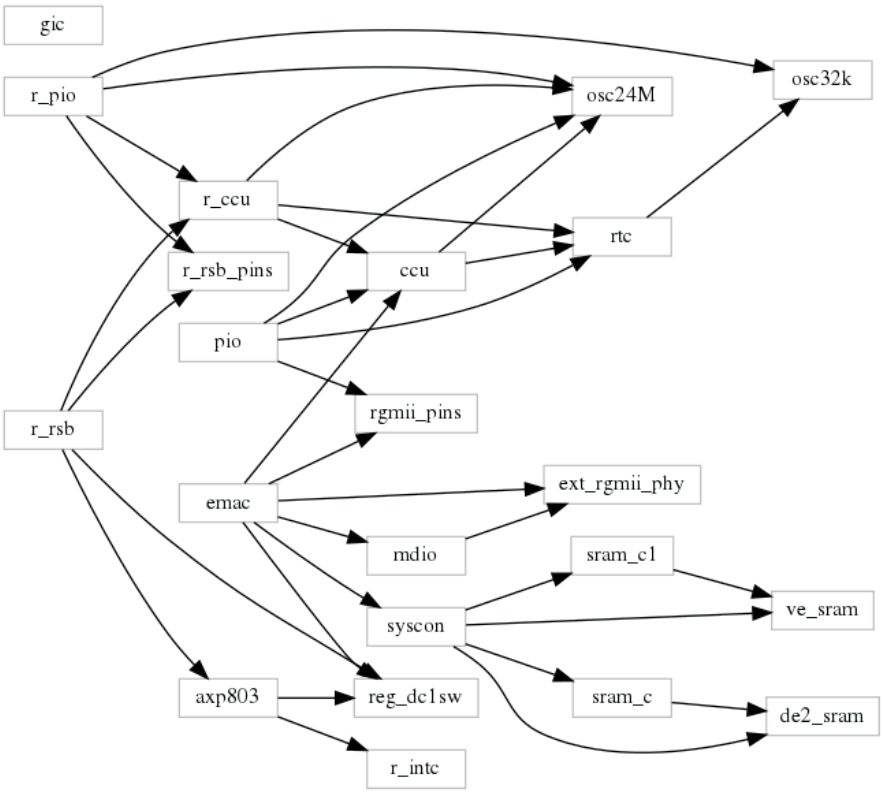


Figure 7: A sudden moment of clarity.

Finally, we can create a device-tree binary out of the pruned device-tree source.

```
tool/dts$ dtc -Idts emac.dts > emac.dtb
```

The device-tree compiler does not complain, which gives us the reassurance that the tree is in a healthy state after the brutal pruning.

Test-driving Linux with the tuned device tree In order to successfully boot the Linux kernel, the supplied device tree needs a few mandatory ingredients. First, we need to supply the information about the timer to be used by the kernel, which is provided by the `/timer` node. Furthermore, `/chosen` node contains the `stdout-path` property, which tells the kernel where messages should go. In the device tree for the Pine-A64-LTS board, it is defined as

```
stdout-path = "serial0:115200n8";
```

The `serial0` part of the string refers to an entry of the `/aliases` node, which is defines as follows. Note that it contains an alias referring to our Ethernet device `emac`.

```
aliases {
  ethernet0 = &emac;
  serial0 = &uart0;
  serial1 = &uart1;
  serial2 = &uart2;
  serial3 = &uart3;
  serial4 = &uart4;
};
```

The following command extracts a device tree featuring those mandatory nodes. Since the `emac` device is implicitly pulled in by the `/alias` node, we don't need to explicitly specify the `-select emac` argument.

```
tool/dts$ ./extract --select /chosen --select /aliases --select /timer \
                    flat_pine64lts.dts
```

The resulting device tree, once compiled into its dtb representation, suffices to boot the hand-crafted Linux kernel we built in the previous article. It looks as follows.

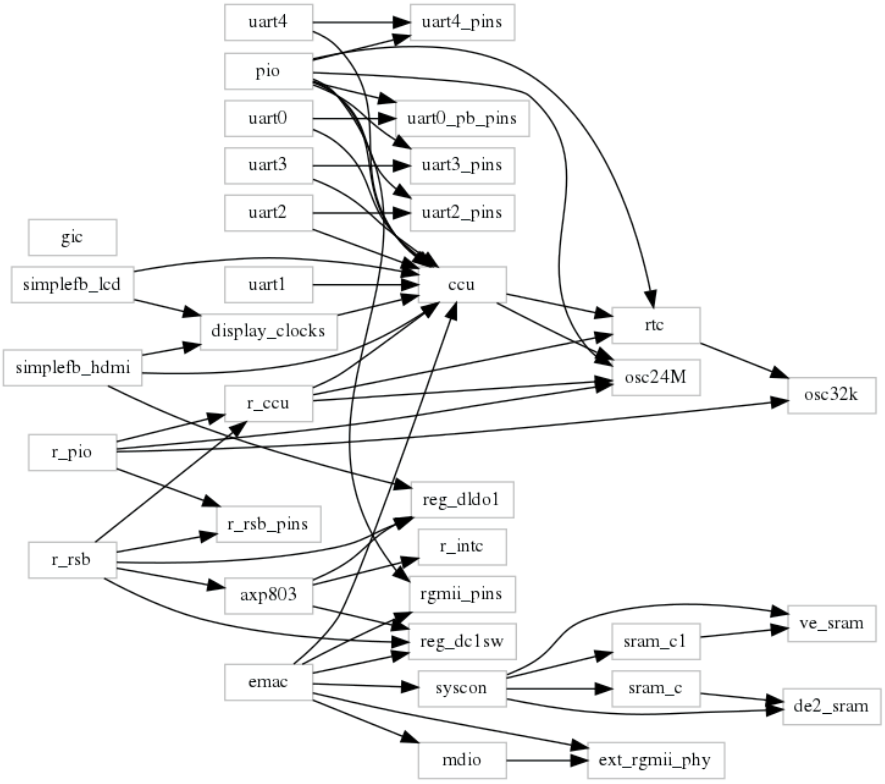


Figure 8: This fine-tuned device tree suffices for using Ethernet with Linux.

The `/timer`, `/chosen`, and `/aliases` nodes are not shown because the graph omits unlabeled nodes. It still contains a few obviously unneeded parts such as the ones relates to the `simplefb` nodes (defined inside the `/chosen` node) or the `uart1` to `uart4` nodes (referenced by the `/aliases` node). To remove those, one may consider cutting off those dependencies by commenting out those parts in the *flat_pine64lts.dts* file.

Prospects Even though the primary motivation behind the new tooling is the pruning of device trees to attain driver-specific miniature device trees to be fed to ported Linux driver code, I already see myself using the graph feature as an aid for understanding SoC hardware. As of now, the graph is admittedly just a quick hack. The dot language allows for generating nicely structured images, e. g., presenting child nodes contained in parent nodes. It's also tempting to generate XML configuration data for Genode's platform driver from the device-tree information.

2.12 Networking

Given the insights gained by driving a device using a tailored bare-bones Linux system as discussed in the previous sections, we are ready to take the next step, namely transplanting Linux driver code into self-sufficient Genode components. This section walks through the challenge of porting a network driver from the Linux kernel to Genode. It thereby exemplifies Genode's device-driver environment approach for the reuse of unmodified Linux kernel code, touches crucial technicalities of the Linux kernel, and provides practical clues.

For reference, the result of the work described herein can be found at the *genode-allwinner* repository.

Git repository of the Allwinner board support

https://github.com/genodelabs/genode-allwinner

Overview The activity of porting a Linux driver is too elaborate for one swift step. To get a sense of measurable progress, it is useful and motivating to define intermediate goals that can be wrapped up one by one.

1. Creating a **minimal Linux kernel configuration** that accommodates barely more than the single driver we are interested in, similar to what we did in the Section 2.10. While pursuing this goal, one can solely keep the focus on the Linux kernel configuration.

2. Getting a tangible feeling for the targeted device and the interplay with other devices. Looking at the **pruned device tree** as described in Section 2.11 is a good aid. Look at the cobweb of device nodes and try to make a mental picture out of it. E.g., in the case of the network driver, we have to consider the ethernet PHY, the actual network controller (emac), and spot the dependency from certain clocks and voltages as potential risks. To double-check the findings, it is recommended to test-drive Linux with the pruned device tree to see if it is still able to operate the device.

3. Creating the initial source skeleton of the driver component and successfully compiling and linking Linux code into a **first executable ELF binary**. During this step, one can focus solely on the build system, symbols, and compilation units. One doesn't have to understand the code in order to link it. This step is described in Sections 2.12.1 and 2.12.2.

4. Creation of a **test scenario** (run script) and a convenient work flow to execute and quickly update the binary. At this step, which is covered by Section 2.12.3, we are merely concerned with the relationships between components without looking inside them. For the work flow, it is satisfying to string together a few convenient shell commands to ease ones life.

5. Complete the execution of **low-level Linux initcalls** until the first device resources are requested. Here, one has to get close to the Linux code but can ignore any hardware-specific concerns.

6. Moving the work flow over to the **real hardware** leveraging the capabilities of Genode's run tool, and possibly tweaking it using custom plugins. The focus is on the use of shell commands and glueing them together using Tcl script snippets, possibly even automating the powering[1] of the target hardware to make one's life better.

7. Successively **completing all Linux initcalls** including those of the driver code. One has to iteratively boot the target hardware, look at the log output, compare the log with the Linux kernel log obtained beforehand. Whenever both logs diverge in non-plausible ways, investigate and instrument the code of both native Linux and Genode. In other words, taking a deep dive into the Linux kernel code, adding additional Linux subsystems, curating dummy functions, supplementing custom emulation code, and extending the platform-driver's configuration whenever encountering the driver's request for additional hardware resources. This step is addressed by Sections 2.12.4 and 2.12.5.

8. Once all initcalls are executed, **provoke a small operation** of the driver. For example, exercising the link detection of the network driver by plugging/unplugging the cable, comparing the resulting log output with native Linux. For the first time, the driver's actual functionality and its interplay with the physical world comes into focus. During the iterative debugging and learning, the tips and tricks given by Sections 2.12.7 and **[?]** may be of help.

9. Get the **driver's core functionality** in the form of a self-sufficient program to work. In the network driver's case, this would be the determination of the device's MAC address as well as the transmission and reception of network packets. At this stage, the driver remains co-located with the test code in one program. In other words, the test program uses the Linux kernel's internal APIs directly. For the network driver, one can conveniently use Linux' builtin DHCP support as test program as described in below in Section 2.12.13.

10. Adding the **Genode service interface** to the driver, e. g., by using the building blocks of the `genode_c_api`. To test this integration, the test scenario must be enhanced by a separate component that interacts with the driver component using a Genode session interface. Knowing that the Linux code and the device is operational, one can focus solely on the Genode integration at this stage. This step is covered in more detail in Section 2.12.17.

[1] https://genodians.org/chelmuth/2019-03-13-powerplug

11. Once the driver works reliably in a minimalistic setting, it is time to expose it to regular networking scenarios by **packaging** it into a form that is digestible by the arsenal of existing run scripts. The packaging step is covered in Section 2.12.18.
12. With non-trivial work loads at hand, one can take a critical look at the driver's behavior, in particular at its performance, and **optimize** it if desired.
13. Finally, one can wrap up the work by cleaning up the code, potentially consolidating parts shared with other drivers, reviewing the result, and documenting the component.

When broken up into these steps - each with a different focus and a tangible intermediate result - the work can be conducted in manageable pieces and can even passed-on between developers.

2.12.1 Directory structure of the driver component

Being a network driver, it naturally should reside somewhere under *src/drivers/nic/*. To avoid possible path ambiguities with network drivers hosted in other Genode source repositories, placing the driver inside a uniquely named subdirectory is a good practice. In our case - with the driver being referred to as "EMAC" - we settle on the directory *src/drivers/nic/emac/*. This directory will host the following files, for which we can initially create skeleton versions based on one of the already existing drivers. E.g., one may take the emac driver[1] as reference when porting a new network driver.

target.mk The build-description file of the driver

main.cc The main program, which initializes the Linux emulation (`Lx_kit::initialize`) along with the Genode service frontend (`genode_uplink_init`), hosts the component's central I/O signal handler, defines the interplay between the execution of Linux and Genode code (`lx_emul_start_kernel` and `Lx_kit::env().scheduler`), and supplies device-tree-binary information to the Linux code (`_dtb`). It is recommended to take a *main.cc* of an existing driver as starting point.

generated_dummies.c Dummy implementations of symbols normally provided by the Linux kernel. As the name suggests, the content won't be manually maintained but generated. So we best start with an empty file.

dummies.c Whenever a generated dummy will be called, the execution will stop with a message along with a backtrace, which will prompt us to closely inspect the situation and decide whether the call of the dummy can be ignored by returning an appropriate return code or must be replaced by an actual implementation. In the former case, the dummy implementation must be moved from *generated_dummies.c* to the manually curated *dummies.c* file.

[1] https://github.com/genodelabs/genode-allwinner/tree/master/src/drivers/nic/emac

lx_emul.c This file contains the implementation of symbols that require more thought than merely being a dummy.

lx_emul.h This header is included by both *dummies.c* and *generated_dummies.c*. It can thereby be used as a manually maintained supplement to the automatically generated *generated_dummies.c*.

lx_user.c This file contains the implementation of a representative of a Linux user task. It provides the symbol `lx_user_init` that is called once the Linux kernel initialization is completed. As a skeleton, an empty implementation suffices.

```
#include <lx_user/init.h>

void lx_user_init(void)
{
}
```

At a later stage, *lx_user.c* will be our hook for connecting the Linux kernel world with a Genode service interface.

source.list This file contains the selection of Linux source codes to be included in our driver. Each line refers to file specified relative to the root of the Linux kernel tree. Most of the development work revolves around the curation of this list. As a starting point, it is useful to take an existing source list as starting point, in particular the selection of *lib/*, *kernel/*, and *arch/* files.

src/include/lx_emul/initcall_order.h In contrast to the files above, which reside local to the driver's directory, the *initcall_order.h* header is used across the Linux drivers. It equips the Linux emulation with the information about the correct initcall sequence. Later, we will see how to generate this file automatically. Until then, the following empty skeleton will do.

```
 static const char * lx_emul_initcall_order[] = {
   "END_OF_INITCALL_ORDER_ARRAY_DUMMY_ENTRY"
};
```

Build magic When reviewing the *target.mk* file of the emac driver, it is obvious that there must be more to the build rules than those few dull lines. The magic happens in the lib-import file import-a64_lx_emul.mk[1]. This file is supplemented to the build process because the *target.mk* specifies *a64_lx_emul* as a library dependency.

[1] https://github.com/genodelabs/genode-allwinner/blob/master/lib/import/import-a64_lx_emul.mk

```
LIBS = base a64_lx_emul
```

The content of *import-a64_lx_emul.mk* is worth studying. It lists several building blocks of the *lx_kit* and *lx_emul* libraries that are used across all Linux drivers ported for the A64 SoC, it defines the compiler flags used for building Linux C code, and it obtains the list of C source files from the driver's *sources.list* file. Furthermore, it evaluates the `BOARDS` and `DTS_EXTRACT` variables to generate driver-specific device-tree binary files. Given this mechanism, the *target.mk* of a driver solely needs to declare the supported `BOARDS` and the driver-specific selection of device-tree nodes to produce a ready-to-use dtb file, e. g.,

```
BOARDS := pine_a64lts
DTS_EXTRACT(pine_a64lts) := --select emac
```

Finally, the import file contains a number of tweaks and quirks such as disabling certain warnings for individual compilation units or generating build artifacts that are implicitly generated by the Linux build system (crc32table.h).

Generated Linux headers The Linux build system generates a number of header files when preparing a build directory. When compiling Linux code outside the Linux build system - as we do - those headers are badly missed. This is where the a64_linux_generated[1] library comes in. This pseudo library has the sole purpose of creating a Linux build directory with the generated headers we need. It takes the same Linux kernel configuration[2] as used for the a64_linux target we discussed earlier in Section [?].

2.12.2 Identifying Linux source codes of interest

How to spot the files needed to drive our network device among the many thousands of C files found in the Linux source tree?

Taking the device tree as our guide In the remainder of this article, we refer an all-encompassing device-tree source (DTS) file *flat_pine64lts.dts* for our board. This file can be extracted from the Linux kernel source via the C preprocessor as described in Section 2.10.5.

Given the *flat_pine64lts.dts* file, let Genode's *dts/extract* tool (Section 2.11) guide our attention:

[1] https://github.com/genodelabs/genode-allwinner/blob/master/lib/mk/spec/arm_v8/a64_linux_generated.mk

[2] https://github.com/genodelabs/genode-allwinner/blob/master/src/a64_linux/target.inc

```
$ .../tool/dts/extract --select emac flat_pine64_lts.dts \
                       | grep "compatible = "
  compatible = "fixed-clock";
  compatible = "fixed-clock";
  compatible = "simple-bus";
   compatible = "allwinner,sun50i-a64-system-control";
    compatible = "mmio-sram";
     compatible = "allwinner,sun50i-a64-sram-c";
    compatible = "mmio-sram";
     compatible = "allwinner,sun50i-a64-sram-c1",
   compatible = "allwinner,sun50i-a64-ccu";
   compatible = "allwinner,sun50i-a64-pinctrl";
   compatible = "allwinner,sun50i-a64-emac";
    compatible = "snps,dwmac-mdio";
   compatible = "arm,gic-400";
   compatible = "allwinner,sun50i-a64-rtc",
   compatible = "allwinner,sun50i-a64-r-intc",
   compatible = "allwinner,sun50i-a64-r-ccu";
   compatible = "allwinner,sun50i-a64-r-pinctrl";
   compatible = "allwinner,sun8i-a23-rsb";
  compatible = "x-powers,axp803";
 compatible = "pine64,sopine-baseboard", "pine64,sopine",
  compatible = "ethernet-phy-ieee802.3-c22";
 compatible = "pine64,pine64-lts", "allwinner,sun50i-r18",
```

Note that some `compatible` attributes span multiple lines (the lines ending with a comma). So it's probably best to manually inspect the device-tree source to get the full information.

Recap that we already used the `compatible` device-node attributes in Section 2.10.6 to connect the dots between the device tree and kernel-configuration options. Analogously, we can use those attribute values to look up the associated source codes.

Let's take "snps,dwmac-mdio" as an pattern to grep Linux source tree:

```
linux$ grep -rl "snps,dwmac-mdio" drivers
drivers/net/ethernet/stmicro/stmmac/stmmac_platform.c
```

By looking at the Makefile where the driver code is located, we can immediately spot the set of driver sources declared by looking at the listed object files. This information is all we need to expand our *sources.list* file.

```
...
drivers/net/ethernet/stmicro/stmmac/stmmac_platform.c
drivers/net/ethernet/stmicro/stmmac/stmmac_main.c
drivers/net/ethernet/stmicro/stmmac/stmmac_ethtool.c
drivers/net/ethernet/stmicro/stmmac/stmmac_mdio.c
drivers/net/ethernet/stmicro/stmmac/ring_mode.c
drivers/net/ethernet/stmicro/stmmac/chain_mode.c
drivers/net/ethernet/stmicro/stmmac/dwmac_lib.c
drivers/net/ethernet/stmicro/stmmac/dwmac1000_core.c
drivers/net/ethernet/stmicro/stmmac/dwmac1000_dma.c
drivers/net/ethernet/stmicro/stmmac/dwmac100_core.c
drivers/net/ethernet/stmicro/stmmac/dwmac100_dma.c
drivers/net/ethernet/stmicro/stmmac/enh_desc.c
drivers/net/ethernet/stmicro/stmmac/norm_desc.c
drivers/net/ethernet/stmicro/stmmac/mmc_core.c
drivers/net/ethernet/stmicro/stmmac/stmmac_hwtstamp.c
drivers/net/ethernet/stmicro/stmmac/stmmac_ptp.c
drivers/net/ethernet/stmicro/stmmac/dwmac4_descs.c
drivers/net/ethernet/stmicro/stmmac/dwmac4_dma.c
drivers/net/ethernet/stmicro/stmmac/dwmac4_lib.c
drivers/net/ethernet/stmicro/stmmac/dwmac4_core.c
drivers/net/ethernet/stmicro/stmmac/dwmac5.c
drivers/net/ethernet/stmicro/stmmac/hwif.c
drivers/net/ethernet/stmicro/stmmac/stmmac_tc.c
drivers/net/ethernet/stmicro/stmmac/dwxgmac2_core.c
drivers/net/ethernet/stmicro/stmmac/dwxgmac2_dma.c
drivers/net/ethernet/stmicro/stmmac/dwxgmac2_descs.c
```

Taking the Linux build directory as our guide Alternatively to taking the device-tree as the starting point, the build directory of our bare-bones Linux kernel contains instructive information, specifically the object files that went into the kernel.

```
build/arm_v8a$ find a64_linux/ -name "*.o"
```

Remember that we have previously slimmed down the Linux kernel configuration as far as we could, keeping only the bare minimum needed for networking. So the list of object files found in the Linux build directory serves as a reasonably small superset of the compilation units that are of interest to us. Any compilation unit not listed cannot be important.

By combining both perspectives, taking the `compatible` device-nodes attributes as cues while using the Linux kernel's object files as plausibility check, our mental picture of the driver code and its dependencies becomes more and more clear and our *sources.list* file grows.

Build test With the *sources.list* enriched with the list of Linux source codes we are interested in, let's have the Genode build system chew a bit on our driver:

```
build/arm_v8a$ make drivers/emac
 ...
 Program drivers/nic/emac/emac_nic_drv
   COMPILE  arch/arm64/lib/memchr.o
   COMPILE  arch/arm64/lib/memcmp.o
   ...
   COMPILE  drivers/net/ethernet/stmicro/stmmac/stmmac_main.o
   COMPILE  drivers/net/ethernet/stmicro/stmmac/stmmac_platform.o
   COMPILE  drivers/net/ethernet/stmicro/stmmac/stmmac_mdio.o
   COMPILE  drivers/net/ethernet/stmicro/stmmac/stmmac_ptp.o
   COMPILE  drivers/net/ethernet/stmicro/stmmac/stmmac_tc.o
   COMPILE  dummies.o
   COMPILE  main.o
   LINK     emac_nic_drv
drivers/net/ethernet/stmicro/stmmac/stmmac_main.o: in function 'stmmac_cmdline_opt':
                       stmmac_main.c:5379: undefined reference to 'strsep'
...
... many more undefined references
...
```

We see the Linux source code being picked up and compiled! The build system backs out not before the linking stage. During linking, however, the many inter-dependencies of the driver code from the rest of the Linux kernel become visible in the form of "undefined reference" errors.

Tying the loose ends to make the linker happy In principle, we could inspect and resolve each of those linking errors manually. But given the volume of error messages, this would be tedious. In this situation, Genode's *dde_linux/create_dummies* tool comes to the rescue.

Let's remember the location of our driver's *generated_dummies.c* file in an environment variable named `DUMMY_FILE`, which will later be evaluated by the *create_dummies* tool:

```
build/arm_v8a$ export DUMMY_FILE=/path/to/src/drivers/emac/generated_dummies.c
```

With the `DUMMY_FILE` defined, we can instruct the tool to fill the file with a dummy implementation for each of the unresolved references reported by the linker:

```
build/arm_v8a$ echo > $DUMMY_FILE ;\
               ../../tool/dde_linux/create_dummies generate \
                                    LINUX_KERNEL_DIR=a64_linux \
                                    TARGET=drivers/nic/emac ;\
               make drivers/nic/emac
```

In the command line above, we first wipe any prior content from the file, then invoke the create_dummies tool, followed by another build of the driver with the new version of generated file. The *create_dummies* tool is described in more detail at a dedicated article[1].

With the dummies generated, the linking of the executable binary succeeds! This won't be the last time of issuing the command. In fact, each time after adding or removing any content of the *sources.list* file, it is best to update *generated_dummies.c* using the command above.

2.12.3 Executable testbed

With a first executable binary built, it is time to give it a first run. At this point, we are not yet concerned about accessing any actual hardware. We merely want to obtain the first life sign of the component and see any hint of Linux initialization code being executed. The following run script named after the driver - in our case *a64_emac_drv.run* is an appropriate name - can serve as our initial test scenario.

[1] https://genodians.org/skalk/2021-04-08-dde-linux-experiments-1#Generate_missing_implementations

```
build { core init timer drivers/platform drivers/nic/emac }

create_boot_directory

install_config {
  <config>
    <parent-provides>
      <service name="LOG"/>
      <service name="PD"/>
      <service name="CPU"/>
      <service name="ROM"/>
      <service name="IO_MEM"/>
      <service name="IRQ"/>
    </parent-provides>

    <default caps="100"/>

    <start name="timer">
      <resource name="RAM" quantum="1M"/>
      <route> <any-service> <parent/> </any-service> </route>
      <provides> <service name="Timer"/> </provides>
    </start>

    <start name="platform_drv">
      <resource name="RAM" quantum="1M"/>
      <provides> <service name="Platform"/> </provides>
      <route> <any-service> <parent/> </any-service> </route>
      <config devices_rom="config">
        <policy label="emac_nic_drv -> ">
        </policy>
      </config>
    </start>

    <start name="emac_nic_drv">
      <resource name="RAM" quantum="1M"/>
      <route>
        <service name="ROM"> <parent/> </service>
        <service name="CPU"> <parent/> </service>
        <service name="PD">  <parent/> </service>
        <service name="LOG"> <parent/> </service>
        <service name="Timer">    <child name="timer"/> </service>
        <service name="Platform"> <child name="platform_drv"/> </service>
      </route>
      <config/>
    </start>

  </config>
}

build_boot_image { core ld.lib.so init timer platform_drv
                   emac_nic_drv emac-pine_a64lts.dtb }

run_genode_until forever
```

The emac_nic_drv is connected to the platform_drv but no device resources are assigned to the corresponding policy yet. In fact, the device hardware remains completely untouched, which allows us to execute the run script for an arbitrary boards, in particular Qemu. The use of Qemu instead of the targeted board at this early stage is convenient.

The dtb file emac-pine_a64lts.dtb as integrated into the boot image is a side-product of building the emac driver. It is created according to the definition of the BOARDS and DTS_EXTRACT variables in the driver's target.mk file.

2.12.4 Linux initcalls

When executing the run script exemplified above, we are greeted with the following log output.

```
kernel initialized
ROM modules:
 ROM: [0000000040120000,0000000040120469) config
 ROM: [0000000040009000,000000004000a000) core_log
 ROM: [0000000040149000,000000004014a094) emac-pine_a64lts.dtb
 ROM: [0000000040207000,000000004023cbe8) emac_nic_drv
 ROM: [000000004023d000,00000000402841f8) init
 ROM: [0000000040159000,0000000040206880) ld.lib.so
 ROM: [0000000040121000,0000000040148d30) platform_drv
 ROM: [0000000040007000,0000000040008000) platform_info
 ROM: [000000004014b000,0000000040158710) timer

Genode 21.05-10-g51f02a668d9
2010 MiB RAM and 64533 caps assigned to init
[init -> emac_nic_drv] Error: Initcall __initcall_initialize_ptr_randomearly unknown in init
[init -> emac_nic_drv] Error: Initcall __initcall_init_jiffies_clocksource1 unknown in initca
[init -> emac_nic_drv] Error: Initcall __initcall_stmmac_init6 unknown in initcall database!
[init -> emac_nic_drv] Error: Initcall __initcall_sync_state_resume_initcall7 unknown in init
[init -> emac_nic_drv] Error: Initcall __initcall_devlink_class_init2 unknown in initcall dat
[init -> emac_nic_drv] Error: Function kmem_cache_init not implemented yet!
[init -> emac_nic_drv] Backtrace follows:
[init -> emac_nic_drv] 0x1024034
[init -> emac_nic_drv] 0x1016898
[init -> emac_nic_drv] 0x1022498
[init -> emac_nic_drv] Will sleep forever...
```

The messages "Error: Initcall ... unknown in initcall database!" tell us that the Linux code we incorporated into our component features initcalls that are unknown to the lx_emul execution environment. Hence, lx_emul won't know the order, in which those

calls must be executed. More information about the initcall mechanism is available in a dedicated article[1].

For us, it is important to know that the initcall order is supplemented to the lx_emul library via the header file at *src/include/lx_emul/initcall_order.h*. The content of this file depends on the Linux kernel configuration. Fortunately, we won't need to maintain this header file by hand. Instead, the handy *extract_initcall_order* tool allows us to generate this file from the *System.map* of a built Linux kernel:

```
build/arm_v8a$ ../../tool/dde_linux/extract_initcall_order extract \
       LINUX_KERNEL_DIR=a64_linux
       HEADER_FILE=/path/to/drivers/src/include/lx_emul/initcall_order.h
```

Note that the directory specified at `LINUX_KERNEL_DIR` must contain a built Linux kernel, specifically the *System.map* file.

When re-running the run script with the updated *initcall_order.h*, the initcall-related errors should disappear.

2.12.5 The lx_emul building blocks

A high-level overview of the anatomy of a DDE-Linux-based driver is provided in the release documentation of Genode 21.08[2]. The lx_emul library provides three kinds of build blocks.

First, it provides a custom C interface for low-level mechanisms of the runtime. The corresponding functions are prefixed with `lx_emul_`. The interface is provided at *dde_linux/include/lx_emul/*.

Second, it provides alternative implementations of low-level Linux subsystems. Those implementations reside at *dde_linux/src/lib/lx_emul/shadow/*. For example, *shadow/mm/slub.c* is an alternative to Linux' *mm/slub.c* that provides the same binary interface but implements it by the means of the lx_emul mechanisms.

And third, it provides a few shadow headers at *dde_linux/src/include/lx_emul/shadow/* that strip away or tweak a few unpleasant parts of the Linux-internal interfaces. In particular, it redirects Linux' original initcall mechanism to the use of `lx_emul_register_initcall` and it hides low-level aspects of the memory model that are incompatible with Genode. The latter is concerned with the conversion between virtual addresses, `struct page` pointers, and DMA addresses.

[1] https://genodians.org/skalk/2021-04-08-dde-linux-experiments-1#Initcalls

[2] https://genode.org/documentation/release-notes/21.08#Linux-device-driver_environment_re-imagined

2.12.6 Iterative crafting of the driver's runtime environment

The message "Error: Function ... not implemented yet!" (in the log output above) followed by a backtrace is triggered by one of the dummy implementations in `generated_dummies.c`. It tells us that we need to replace this particular dummy with either

- A dummy implementation in the manually curated `dummies.c` with the call to `lx_emul_trace_and_stop` replaced by a meaningful return value, or
- An actual implementation of the function in `lx_emul.c`, or
- Additional Linux source codes incorporated by extending the *sources.list*.

The decision must be taken on a case-by-case basis. To take the decision, it is worthwhile to inspect the existing drivers. Drivers of the same kind (network, framebuffer) tend to show similar patterns across SoCs.

Our job at this stage is the repeated execution of the run script while resolving one unimplemented function in each iteration. Sometimes, this process requires a deep dive into parts of the Linux kernel architecture in order to asses the potential impact of the called dummy on the correct functioning of the driver. Sometimes mere intuition may guide us. In any case, the backtrace printed in the log output can be immensely helpful. You may remember the **addr2line** utility mentioned in Section 2.5.2. It accepts an arbitrary sequence of numbers as standard input when started as follows:

```
build/arm_v8a$ /usr/local/genode/tool/current/bin/genode-aarch64-addr2line \
                                          -e drivers/nic/emac/emac_nic_drv
```

To process the list of hexadecimal numbers appearing in the log, I use to copy the numbers from the terminal using a rectangular selection (by pressing the control key while selecting an area with the mouse) and pasting the content into the `addr2line` instance.

Moving to the target hardware At one point, we will ultimately reach a point where the driver tries to obtain access to device resources.

```
[init -> emac_nic_drv] Error: memory-mapped I/O resource 0x1c00000
                              (size=0x1000) unavailable
```

It's time to move the development from Qemu to the actual target hardware.

In order to grant the driver access to the requested resource, we first look up the requested address in the *flat_pine64lts.dts* file. In the example above, the range belongs to a device called `syscon`. With the information found in the device node, we can enrich the platform driver's configuration accordingly.

```
<config>
  <device name="syscon" type="allwinner,sun50i-a64-system-control">
    <io_mem address="0x1c00000" size="0x1000"/>
  </device>
  <policy label="emac_nic_drv -> " info="yes">
    <device name="syscon"/>
  </policy>
</config>
```

The change consist of two parts. First, a `<device>` is declared. Note that the `type` corresponds to the `compatible` attribute of the DTS device node. Second, the `<policy>` for the emac_nic_drv is changed to grant access of this device to the driver. It is important to set the `info` attribute to "yes", which allows the driver to read the device meta information given in the `<device>` node.

Besides memory-mapped I/O registers, device interrupts are the second type of hardware resource of interest for device drivers. Sooner or later during the driver initialization, we encounter a message like the following.

```
[init -> emac_nic_drv] Error: irq 114 unavailable
```

The driver requests GIC interrupt 114. To find out what's behind this number, the *flat_pine64lts.dts* file is the right place for seeking the ground truth. Note that interrupt numbers as found in DTS files correspond to GIC interrupts numbers minus 32. So GIC interrupt 114 appears as number 114 - 32 = **82**. A search in the DTS file for this number leads us to the matching device.

```
emac: ethernet@1c30000 {
  ...
  reg = <0x01c30000 0x10000>;
  ...
  interrupts = <0 82 4>;
  ...
};
```

This information is all we need to craft a corresponding `<device>` node for the platform-driver configuration.

```
<device name="emac" type="allwinner,sun50i-a64-emac">
  <io_mem address="0x1c30000" size="0x10000"/>;
  <irq number="114"/>
</device>
```

2.12.7 Linux caveats

In the past, we repeatedly encountered two kinds of trip-wires that one should always keep in the back of the mind, namely Linux linker-script magic and global variables.

A few kernel mechanisms depend of special support at the linker-script level, most notably various flavours of initcall mechanisms, sometimes disguised as a global table (`__clk_of_table`, `__irqchip_of_table`) magically created by scattered table entries assigned to a special linker section. In contrast to Linux, we cannot rely on linker-level mechanisms if we want to keep using Genode's regular linker script. The lx_emul environment takes care of the initcall flavors that we encountered so far using the pattern described here[1]. But we know that there exist more categories of initcalls. In the event that a certain initialization function is unexpectedly not called, it is worth skimming the symbols of `generated_dummies.o` for variables with `table` in their name. Another example for linker magic is the aliasing between the `jiffies` and `jiffies64` variables. Both variables must refer to the same memory location (on little-endian architectures). This concrete issue is covered by the *lib/import/import-a64_lx_emul.mk* file.

The second trip wire is the presence of global variables that are specially initialized by compilation units not featured in *sources.list*. In this case, the *generated_dummies.c* creates default-initialized variable instances, which can break innocent library functions in subtle ways. For example, *lib/hexdump.c* contains the following global variable:

```
const char hex_asc_upper[] = "0123456789ABCDEF";
EXPORT_SYMBOL(hex_asc_upper);
```

This variable is implicitly used by *lib/vsprintf.c* when printing "%d" format strings. If default-initialized, a digit is wrongly rendered as null (terminating the string) instead of the corresponding ASCII value, producing all kinds of funny effects down the road. The global variable defined in *lib/ctypes.c* is equally important. If default-initialized, `toupper` and `strcasecmp` won't work as expected, breaking the program logic when used as condition.

As a rule of thumb, when encountering erratic behavior, one should look out for global variables in *generated_dummies.c* and investigate their purpose.

2.12.8 Enabling Linux debug messages

Several parts of the Linux kernel are garnished with debug messages that reveal valuable insights of the kernel's behavior beyond the regular log messages. The easiest way to obtain those messages for a given compilation unit is adding the following line right at the beginning of the source file, above the first `#include` directive:

[1] https://genodians.org/skalk/2021-04-08-dde-linux-experiments-1#Initcalls

```
#define DEBUG
```

When booting Linux, one has to supply "debug" as kernel-command line argument. When running the driver as Genode component, no further precaution is needed.

With respect to debug instrumentation, the following compilation units are particularly fruitful:

drivers/of/fdt.c

On ARM platforms, the kernel initialization is driven by the information of the supplied device tree. By enabling `DEBUG` in this compilation unit, one becomes able to observe the processing of the device nodes found in the device-tree and how they are matched with the available drivers.

drivers/base/dd.c

By enabling `DEBUG` in this compilation unit, the probing of devices by the various drivers becomes visible. This is particular important for devices that depend on each other. Whenever a driver finds a precondition - such as the presence of another driver - not met, it backs out of the probing via Linux' defer mechanism (`EPROBE_DEFER`). Whenever the deferral of probing diverges between native Linux and the ported driver, one should investigate the root cause of the condition that led (or did not led) to an `EPROBE_DEFER`.

2.12.9 Logging the execution of initcalls

To unveil the execution sequence of initcalls and for relating messages and backtraces printed in the log with the corresponding Linux code, two little instrumentations are of great help.

repos/dde_linux/src/lib/lx_emul/init.cc

By adding a message like the following to the `lx_emul_register_initcall` function, we become able to relate the names of initcall functions with their corresponding addresses.

```
Genode::log("lx_emul_register_initcall ", name, " call=", (void *)initcall);
```

Since `lx_emul_register_initcall` is called immediately at component construction time using the global ctors mechanism, this instrumentation gives us a complete list of initcalls. The names of those calls can easily be grep'ed in the Linux code to determine a suitable starting points for manual instrumentations.

src/lib/lx_kit/init.cc

By changing the implementation of `Lx_kit::Initcalls::execute_in_order` to print a log message in addition to `entry→call()`, we can know exactly when each initcall is executed.

```
log("exec init call ", (void *)entry->call);
```

The printed addresses correspond to those obtained in the first instrumentation. So when the kernel initialization gets stuck somewhere, one can look at the sequence of initcalls - and in particular the last initcall executed - that led to the situation.

2.12.10 Obtaining backtraces of blocked Linux tasks

The Linux kernel code is not executed in a straight linear fashion but in the form of many kernel threads that interact with each other using a variety of mechanisms such as work queues. The lx_emul runtime implements a cooperative task-execution model that folds all Linux kernel threads into a single flow of control. To see what the Linux kernel threads are doing and in particular the situation when they enter a blocking state, the following instrumentation in *dde_linux/src/lib/lx_kit/task.cc* is invaluable.

```
#include <os/backtrace.h>

void Task::block()
{
  log("Task::block: ", _name);
  backtrace();
  ...
```

The `Task::block` method is called whenever a Linux kernel thread blocks. By adding the two lines at the beginning of the method, we get hold of the situation at each single task switch. It prints the name of the blocked kernel thread along with the backtrace of the thread.

Another suitable point for an instrumentation is the `Task::run` method. By printing the `_name` after the `_setjmp` branch, one can obtain the sequence of resumed (as opposed to blocked) kernel threads.

```
void Task::run()
{
  if (_setjmp(_saved_env))
    return;

  log("Task::run: ", _name);
  ...
```

2.12.11 De-referenced null pointers

The Linux kernel is anything but short of function pointers and callbacks. Hence, sooner or later during the development, one may be faced with a de-referenced null pointer like this:

```
no RM attachment (READ pf_addr=0x18c pf_ip=0x1008fa0 from pager_object:
                       pd='init -> emac_nic_drv' thread='ep')
Warning: page fault, pager_object: pd='init -> emac_nic_drv' thread='ep'
                                   ip=0x1008fa0 fault-addr=0x18c type=no-page
```

The very small page-fault address (`pf_addr`) hints at a de-referenced null pointer. The first impulse is looking up the instruction pointer `pf_ip` in the driver's binary using `objdump`.

```
build/arm_v8a$ /usr/local/genode/tool/current/bin/genode-aarch64-objdump \
                            -lSd drivers/nic/emac/emac_nic_drv | less
```

In less, when searching for the instruction pointer value (`1008fa0`), one can see the surroundings of the offending code.

```
1008f9c:       aa0003f3        mov     x19, x0
.../src/linux/drivers/base/regmap/regmap.c:2720
        if (!IS_ALIGNED(reg, map->reg_stride))
1008fa0:       b9418c00        ldr     w0, [x0, #396]
```

Could `map` be a null pointer? If so, why? When looking into the code at the displayed coordinates *regmap.c* at line 2720, we encounter the function `regmap_read` as a suitable point for instrumentation.

```
#include <lx_emul.h>
...
int regmap_read(struct regmap *map, unsigned int reg, unsigned int *val)
{
  int ret;
  printk("regmap_read map=%p\n", map);
  if (!map)
    lx_emul_trace_and_stop(__func__);

  if (!IS_ALIGNED(reg, map->reg_stride))
    return -EINVAL;
```

The `printk` should validate our hypothesis that `map` is indeed a null pointer - just to double-check. The `lx_emul_trace_and_stop` call gives us the backtrace in this interesting situation. When running the code with these instrumentations in place, the following output appears.

```
[init -> emac_nic_drv] regmap_read map=0
[init -> emac_nic_drv] Error: Function regmap_read not implemented yet!
[init -> emac_nic_drv] Backtrace follows:
[init -> emac_nic_drv] 0x1008fc0
[init -> emac_nic_drv] 0x1009044
[init -> emac_nic_drv] 0x100a9b8
[init -> emac_nic_drv] 0x10076c0
[init -> emac_nic_drv] 0x10060f8
[init -> emac_nic_drv] 0x1006938
[init -> emac_nic_drv] 0x10069a4
[init -> emac_nic_drv] 0x1001320
[init -> emac_nic_drv] 0x1001ac4
[init -> emac_nic_drv] 0x10074a0
[init -> emac_nic_drv] 0x1056f28
[init -> emac_nic_drv] 0x10481e8
[init -> emac_nic_drv] 0x10580f8
```

Thanks to the backtrace, we can track where the `map` pointer comes from, ultimately ending up at the call of `syscon_regmap_lookup_by_phandle`. In our case, this function was (wrongly) stubbed with a dummy function returning NULL. As a way to double-check that the return value of this function indeed corresponds to the de-referenced null pointer, one can tweak the return value a little, returning a smallish magic number.

```
struct regmap * syscon_regmap_lookup_by_phandle(struct device_node * np,
                                                const char * property)
{
  return (struct regmap *)0x550;
}
```

In the next run, we can observe that the page-fault address indeed changed from 0x18c to 0x6dc.

```
no RM attachment (READ pf_addr=0x6dc pf_ip=0x1008fc0 from pager_object:
                  pd='init -> emac_nic_drv' thread='ep')
Warning: page fault, pager_object: pd='init -> emac_nic_drv' thread='ep'
                                   ip=0x1008fc0 fault-addr=0x6dc type=no-page
```

Apparently, we cannot simply shortcut the `syscon_regmap_lookup_by_phandle` function.

2.12.12 Ruling out potential cache-coherency issues

Once the driver starts to interact with the device hardware, additional uncertainties enter the picture. The most uncertain uncertainty is certainly cache coherency. Nowadays, Linux drivers preferably use cached page mappings for DMA buffers and manage the coherency between the device's perspective and the CPU's perspective on those buffers via explicit cache-management (flush, invalidate) operations. This cache management happens behind the surface of functions like `dma_map_page_attrs`.

To rule out the presence of cache coherency issues, we can force the driver to use uncached mappings only by tweaking the allocators at *dde_linux/src/include/lx_kit/env.h*. By changing the `CACHED` argument of the `memory` member to `UNCACHED`, all memory dynamically allocated by Linux kernel code will be backed by uncached memory.

Should the driver work with this tweak, one can be pretty sure to have hit a cache-coherency issue, likely missing the correct implementation of a `dma_map` / `dma_unmap` operation. Should the driver still does not work, the problem lies somewhere else. Now would be the time to suspect cosmic rays.

2.12.13 Using Linux' built-in DHCP support as networking test

Before equipping the driver with a Genode session interface, it is recommended to first execute its core functionality as a standalone program. For a network driver, the core functionality is the transmission and reception of network packets.

The Linux kernel features builtin support for obtaining an IPv4 network configuration at boot time via DHCP. The network-configuration protocol involves the successful transmission and reception of multiple network packets, and its completion is indicated

by the IP address printed in the kernel log. In other words, DHCP is the ideal first test workload for the driver. It requires the following kernel configuration options:

```
INET
IP_PNP
IP_PNP_DHCP
```

The implementation resides in *net/ipv4/ipconfig.c*, which must be added to the *sources.list* file of the driver. When being part of a regular Linux kernel, this code evaluates the kernel command line, namely the option "ip=dhcp". Since the lx_emul environment has no notion of a kernel command line, we can manually force the code to issue the DHCP request by modifying the implementation of the `ip_auto_config` function by adding calls to `skb_init` and `ip_auto_config_setup` at the beginning of the function (right after the local variable declarations).

```
static int ip_auto_config_setup(char *addrs);

static int __init ip_auto_config(void)
{
  ...
  skb_init();
  ip_auto_config_setup("dhcp");
   ...
```

2.12.14 Capturing network traffic

There exist many ways to capture network traffic for observing the interchange of DHCP protocol messages at the DHCP-server side. The tshark tool is particularly nice. For capturing the traffic related to the MAC address of my board, the following command line does an excellent job:

```
tshark -i eno1 -t ad -Y 'eth.addr == 02:ba:fe:7b:59:38'
```

2.12.15 Using flood ping as a rudimentary stability check

Once the driver has reached a seemingly operational state, having successfully completed DHCP, it is a good time to put some stress on the driver. As a litmus test, its interesting to see if a flood ping brings the driver to its knees:

```
sudo ping -f -c 1000 -s 1000 <ip-address>
```

2.12.16 Cross-correlation against the Linux kernel behavior

Sharing one Linux kernel configuration among both our bare-bones Linux kernel and the ported driver code allows for the detailed cross-correlation of the driver behavior. This consistency is fostered by the src/a64_linux/target.inc[1] file that is used by both the a64_linux target (configuring and building a Linux kernel) and each driver component (via the a64_linux_generated library). This way, any instrumentation of the Linux kernel code can be quickly tested in both execution environments.

2.12.17 Connecting the driver with a Genode session interface

Once we have validated that the driver is able to send and receive network packets via the DHCP test, the time is ripe for integrating it into the Genode environment. This integration comes down to two aspects. First, the test scenario must be changed to move the network application into a component separate from the driver, and second, the driver must interact with Genode's uplink session interface.

The first part, the network application, can be accommodated by the NIC router component. For reference, the following `<start>` node creates an instance of the NIC router that issues a DHCP request once an uplink appears, and prints the obtained IP address in the log.

```
<start name="nic_router" caps="200">
  <resource name="RAM" quantum="10M"/>
  <provides>
    <service name="Nic"/>
    <service name="Uplink"/>
  </provides>
  <route>
    <service name="Timer"> <child name="timer"/> </service>
    <any-service> <parent/> </any-service>
  </route>
  <config verbose_domain_state="yes" dhcp_discover_timeout_sec="1">
    <policy label_prefix="emac_nic_drv" domain="uplink"/>
    <domain name="uplink"/>
  </config>
</start>
```

Of course, the driver must be able to reach the NIC router, which can be achieved adding the following session route to the driver's `<start>` node.

[1] https://github.com/genodelabs/genode-allwinner/blob/master/src/a64_linux/target.inc

```
<start name="emac_nic_drv" caps="2000">
  ...
  <route>
    ...
    <service name="Uplink"> <child name="nic_router"/> </service>
    ...
  </route>
</start>
```

The second part - bridging the gap between the Linux kernel code and Genode's session interface - can best be addressed by the genode_c_api/uplink.h[1] API and an implementation of the driver's lx_user.c[2], which connects the `genode_c_api` with the Linux netdevice interface.

2.12.18 Packaging the driver

The final step is the packaging of the driver to make it available to a broad range of Genode scenarios, in particular the run scripts based on the `drivers_nic` subsystem such as *libports/run/fetchurl_lwip.run*.

The packaging is assisted by the *dde_linux/list_dependencies* tool. It determines the list of header dependencies for our driver by examining dependency (.d) files. For reference, the *nic/emac/dep.list* file is generated via following command (the paths are abbreviated).

```
build/arm_v8a$ ../../tool/dde_linux/list_dependencies \
               TARGET_DIR=drivers/nic/emac \
               LINUX_KERNEL_DIR=/path/to/linux/source/ \
               SOURCE_LIST_FILE=.../allwinner/src/drivers/nic/emac/source.list \
               DEP_LIST_FILE=.../allwinner/src/drivers/nic/emac/dep.list \
               generate
```

As reference for the recipe files needed, the depot recipes at allwinner/recipes/[3] are helpful. Their roles are as follows:

recipes/api/a64_linux/ This API recipe contains the parts of the Linux source tree that are relevant to build the drivers. It also features the parts of the Linux build system that are invoked to generate header files (for the a64_linux_generated library). Each DDE-Linux-based driver depends on this API archive. This recipe notably uses the information of the *dep.list* and *source.list* files.

[1] https://github.com/genodelabs/genode/blob/master/repos/os/include/genode_c_api/uplink.h
[2] https://github.com/genodelabs/genode-allwinner/blob/master/src/drivers/nic/emac/lx_user.c
[3] https://github.com/genodelabs/genode-allwinner/tree/master/recipes

src/a64_emac_nic_drv/ This source archive contains the Genode parts of the network drivers. The Linux sources are taken from the api/a64_linux archive.

recipes/pkg/drivers_nic-pine_a64lts/ This package aggregates all ingredients needed for a network-driver subsystem as expected by scenarios based on the convention of *drivers_nic* packages, that is, run scripts using

```
import_from_depot ... \
                  [depot_user]/pkg/[drivers_nic_pkg]
```

recipes/raw/drivers_nic-pine_a64lts

The raw archive contains the init configuration of the driver subsystem.

While crafting those recipes, it is best to use a dummy depot user "x" so that the intermediate results can easily be removed from the depot afterwards. For reference, the following command extracts the archives from the source tree and builds the binaries for the `arm_v8a` architecture. The process of developing the recipes comes down to repeatedly issuing this command and extending the recipes until the binary archives are successfully built.

```
genode$ ./tool/depot/create x/pkg/arm_v8a/drivers_nic-pine_a64lts \
                            -j8 \
                            FORCE=1 \
                            UPDATE_VERSIONS=1
```

2.13 Display

Until now, the exploration of the Allwinner A64 SoC was mainly concerned with the Pine-A64-LTS board, which offers developer conveniences like booting over the network, or easily accessible reset and GPIO pins. The upcoming topics require us to switch out development workflow from the Pine-A64-LTS board to the real deal - the PinePhone. This adjustment is covered by a dedicated article[1]. With those precautions taken, it is time to turn our attention to the arguably most challenging parts of the hardware, namely the display subsystem.

Why do I regard this part as the most challenging? The display subsystem of a mobile device is not solely one peripheral but a conglomerate of several devices that are (more or less) under software control and need to work together. The complexity of the interplay and domain-specific terminology can be quite staggering. MIPI, DSI, PLL, PHY, panel, plane, channel, connector, encoder, regulator, mixer, CRTC, RSB, TCON, LVDS, PWM. Are you still with me?

2.13.1 Driving the display with a bare-bones Linux kernel

Not knowing much about the internal structure of the display hardware, it is good to take Linux as a working starting point. When booting Armbian Linux, the display works after all. Observing the Linux boot, the following messages seem obviously be related to the display.

```
[    5.936404] Console: switching to colour frame buffer device 170x48
[    5.955920] simple-framebuffer be000000.framebuffer: fb0: simplefb registered!
[    5.959687] mmc1: new SDHC card at address 0001
[    5.967848] sun4i-drm display-engine: bound 1100000.mixer (ops 0xffff800010e340c0)
[    5.979490] sun4i-drm display-engine: bound 1200000.mixer (ops 0xffff800010e340c0)
[    5.990232] sun4i-drm display-engine: No panel or bridge found... RGB output disabled
[    6.000377] sun4i-drm display-engine: bound 1c0c000.lcd-controller (ops 0xffff800010e2f8d
[    6.012100] sun4i-drm display-engine: bound 1c0d000.lcd-controller (ops 0xffff800010e2f8d
[    6.026726] sun8i-dw-hdmi 1ee0000.hdmi: Detected HDMI TX controller v1.32a with HDCP (sun
[    6.117391] sun8i-dw-hdmi 1ee0000.hdmi: registered DesignWare HDMI I2C bus driver
[    6.130875] sun4i-drm display-engine: bound 1ee0000.hdmi (ops 0xffff800010e333f8)
[    6.200146] fb0: switching to sun4i-drm-fb from simple
[    6.210896] Console: switching to colour dummy device 80x25
[    6.216994] [drm] Initialized sun4i-drm 1.0.0 20150629 for display-engine on minor 0
[    6.603061] Console: switching to colour frame buffer device 170x48
[    6.641668] sun4i-drm display-engine: [drm] fb0: sun4i-drmdrmfb frame buffer device
```

Correlating those words with the device tree brings us to the device node of the so-called display engine.

[1] https://genodians.org/nfeske/2021-09-20-pine-fun-pinephone-boot

```
de: display-engine {
 compatible = "allwinner,sun50i-a64-display-engine";
 allwinner,pipelines = <&mixer0>, <&mixer1>;
 status = "disabled";
};
```

The device node's `compatible` string, in turn, draws the connection to the part of the Linux kernel that is of interest to us.

```
linux$ grep -r "allwinner,sun50i-a64-display-engine"
drivers/gpu/drm/sun4i/sun4i_drv.c: { .compatible = "allwinner,sun50i-a64-display-engine" },
```

So *drivers/gpu/drm/sun4i/* seems to be good starting point for exploration.

Having identified the driver code that want to execute for sure, we have to answer two questions:

1. What are the in-kernel dependencies of this driver code? All those dependencies are of interest to us because they are prerequisites.
2. Which parts of the Linux kernel are unrelated to the driver functionality? We would like to drop those parts to narrow our view on the interesting driver code as much as possible.

The investigation of those two questions is an iterative process that follows the pattern discussed in Section 2.10. In our present case, the success criterion of our custom-built bare-bones Linux kernel is the display of the little Tux at the top of the screen. Our kernel won't need anything else, Tux is enough.

To find the smallest possible selection of kernel configuration parameters, the bisecting approach that we previously used for isolating the network driver becomes handy again. Without further ado, here comes the solution as supplement for our target.inc[1].

```
# framebuffer driver
LX_ENABLE += DRM DRM_SUN4I DRM_SUN8I_MIXER DRM_SUN8I_DW_HDMI

# determined by bisecting kernel configuration options (needed by fb driver)
LX_ENABLE += CMA DMA_CMA MFD_AXP20X_RSB REGULATOR REGULATOR_AXP20X
LX_ENABLE += PROC_FS SYSFS

# to automatically set up screen mode at boot time
LX_ENABLE += FRAMEBUFFER_CONSOLE

# show Tux
LX_ENABLE += LOGO
```

[1] https://github.com/genodelabs/genode-allwinner/blob/master/src/a64_linux/target.inc

Don't ask how often I operated the reset button to find this global minimum of kernel configuration parameters.

With the bare-bones Linux kernel running, we can use Busybox to interactively poke around with the driver. It is nice to see some response, like the display going dark.

```
/ # mkdir proc
/ # mkdir sys
/ # mount -tproc proc
/ # mount -tsysfs sys
/ # cd /sys/devices/platform/display-engine/graphics
/sys/devices/platform/display-engine/graphics # cd fb0/
/sys/devices/platform/display-engine/graphics/fb0 # echo 1 > blank
/sys/devices/platform/display-engine/graphics/fb0 # echo 0 > blank
```

To further tighten our focus, the next step is the pruning of the device tree using the DTS-extract tool discussed in Section 2.11. For reference, the device tree extracted with following arguments suffices to allow Linux to drive the display. The central element is the Allwinner Display Engine[1] (DE).

```
genode$ ./tool/dts/extract --select /backlight --select de --select dsi \
                           flat_pinephone.dts
```

The resulting device-tree nodes at a glance:

[1] https://linux-sunxi.org/images/7/7b/Allwinner_DE2.0_Spec_V1.0.pdf

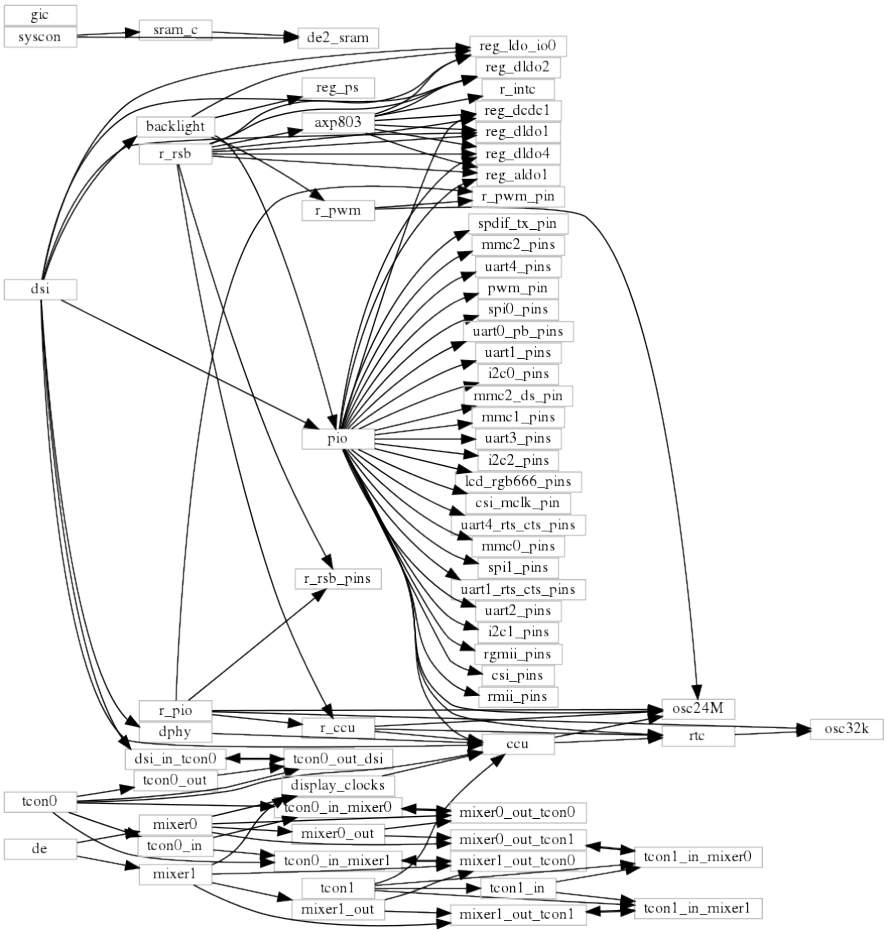


Figure 9: Device-tree nodes related to the display engine and the DSI output.

Let's not get scared. A glossary of the terminology seen the picture should lift the clouds a bit.

- The PIO device controls general-purpose I/O pins. We explored this device previously[1]. All pins are naturally related to the PIO controller but only a few of them are actually relevant for the display. So we can consider the large number of pin nodes as just noise for the most part.
- All nodes prefixed with `r_` belong to a certain part of the SoC that is referred to as "RTC" (real-time clock). Those parts are powered independently from the ARM application processor and are meant to be driven by a small microcontroller called AR100[2] that ought to manage power.

[1] https://genodians.org/nfeske/2021-04-29-platform-driver
[2] https://linux-sunxi.org/AR100

- The `r_rsb` controller is a two-wire bus similar to I2C that connects the A64 SoC with a separate power management chip (PMIC, or AXP803). This chip is responsible for generating various voltages on the board. For driving the display, this chip is important because it provides the power for the LCD display part, which is off by default. So in order to power the display, the driver needs to talk via the RSB bus to the PMIC chip.
- TCON0 and TCON1 are two interfaces of the SoC where a display can be connected. So the SoC supports dual-head scenarios like driving an HDMI display and an LCD display at the same time.
- CCU stands for central clock unit. It controls the configuration and gating of all kinds of internal clocks and reset lines. The R_CCU refers to the clocks and reset lines associated with the RTC part of the chip.
- The backlight is a separate device. Its brightness is controlled via pulse-width-modulated digital signal generated by the `r_pwm` device.
- The DPHY is responsible for the physical link of the digital connector. Electronics stuff.
- All these mixer nodes are related to the display engines ability to blend multiple images together.

When booting our bare-bones Linux kernel with the pruned device tree, Tux shows up, and the last life signs of the kernel are the following messages.

```
sun4i-drm display-engine: No panel or bridge found... RGB output disabled
sun4i-drm display-engine: bound 1c0c000.lcd-controller (ops 0xffffffc010272138)
sun4i-drm display-engine: bound 1c0d000.lcd-controller (ops 0xffffffc010272138)
sun4i-drm display-engine: bound 1ca0000.dsi (ops 0xffffffc010275810)
[drm] Initialized sun4i-drm 1.0.0 20150629 for display-engine on minor 0
sun4i-drm display-engine: [drm] Cannot find any crtc or sizes
Console: switching to colour frame buffer device 90x90
sun4i-drm display-engine: [drm] fb0: sun4i-drmdrmfb frame buffer device
sun6i-mipi-dsi 1ca0000.dsi: Attached device xbd599
panel-sitronix-st7703 1ca0000.dsi.0: 720x1440@55 24bpp dsi 4dl - ready
```

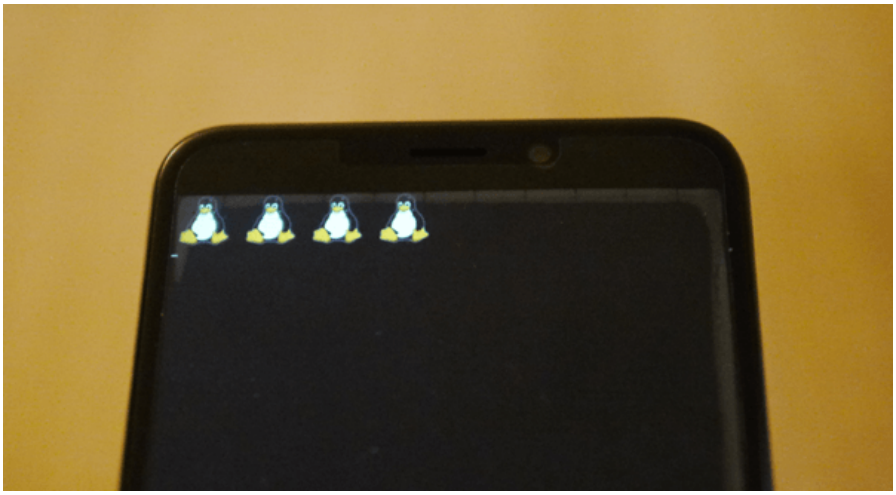


Figure 10: Hello my friends, nice to see you!

The logo. That's all we want from the Linux kernel for now.

2.13.2 A monolithic display driver running on Genode

In Section 2.12, we covered the path from a down-stripped bare-bones Linux kernel to a Genode driver component using Genode's device-driver environment. The same principle method works equally well for transplanting the display driver,

- Selecting the relevant driver sources,
- Generating dummy implementations using the *tool/dde_linux/create_dummies* tool,
- Taking cues from the existing DDE-Linux-based drivers for supplementing custom Linux emulation code,
- Using a custom run script (appropriately named *framebuffer_pinephone.run*) as a dedicated test bed for the driver,
- Resolving the access to device resources like memory-mapped I/O ranges and interrupts by enhancing the platform-driver configuration step by step.

For the test bed, the framebuffer test[1] is a handy tool. When combined with a display driver, it presents a sequence of colors and patterns, and it nicely highlights the border of the screen to verify the entirety of the framebuffer is indeed visible.

[1] https://github.com/genodelabs/genode/tree/master/repos/os/src/test/framebuffer

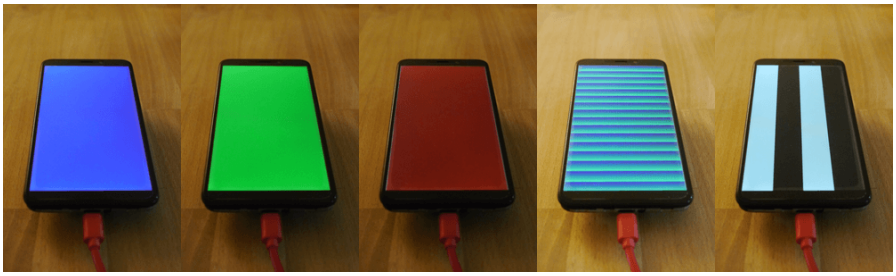


Figure 11: The *framebuffer_pinephone.run* scenario.

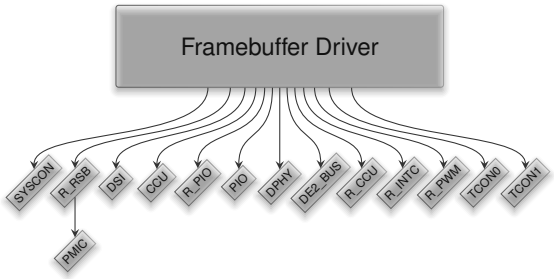
In the test scenario, the test-framebuffer component takes the place of the GUI server, providing a capture service. The framebuffer driver plays the role of a mere capture client that captures the synthetic pixel data generated by the test-framebuffer component.

For interfacing the Linux kernel code with Genode's capture session interface, the driver uses the following kernel function declared in *linux/fb.h* as a hook to get hold of the pixel data.

```
int register_framebuffer(struct fb_info *fb_info);
```

This is arguably quite primitive and does not allow the use of many driver features. However, we have to start somewhere.

The framebuffer driver interacts with device hardware through the platform driver as introduced in Section 2.9. The following picture shows all the devices the driver interacts with.



This octopus-resembling creature raises two tricky questions.

- Is it reasonable to entrust the driver with the access to all those devices? Given the complexity of the ported driver code, this seems risky, doesn't it?
- Some of the devices seem to be relevant to other drivers, too. If we grant the framebuffer driver exclusive access to those devices, how can we combine the framebuffer driver with other drivers running on the same system?

We will come to solving those questions soon. For now, let's enjoy the colorful display for a bit.

For reference, the entire commit for the monolithic framebuffer driver can be found here[1].

[1] https://github.com/genodelabs/genode-allwinner/commit/c1d088ef5d9d4c82ffa761f672c33f70431dec31

2.14 Touchscreen

In the previous Section, we went though the steps of transplanting the PinePhone's highly complex display driver from the Linux kernel into a Genode driver component. Given the lessons learned, porting over the touch screen driver should be a walk in the park, shouldn't it? Let's see.

Information gathering As a pattern established by now, we first will build a minimal Linux kernel that features only our driver of interest. Not before that works, we will transplant the relevant code into our custom Genode component as a second step. The challenge of the first step is finding the right knobs in the Linux kernel configuration to make the driver and device come alive.

Analogously to the network and display drivers covered earlier, our trail head is the device tree (Section 2.12.2) shipped with the vendor Linux kernel. By searching for "touch" in the device tree for the PinePhone, the following device node presents itself:

```
touchscreen@5d {
 compatible = "goodix,gt917s";
 reg = <0x5d>;
 interrupt-parent = <&pio>;
 interrupts = <7 4 4>;
 irq-gpios = <&pio 7 4 0>;
 reset-gpios = <&pio 7 11 0>;
 AVDD28-supply = <&reg_ldo_io0>;
 VDDIO-supply = <&reg_ldo_io0>;
 touchscreen-size-x = <720>;
 touchscreen-size-y = <1440>;
};
```

Given this anchor, we can use Genode's dts/extract[1] tool to figure out the inter-device dependencies within the SoC. First, let's determine the complete path of the device node within the device tree.

```
genode$ ./tool/dts/extract --nodes flat_pinephone.dts | grep touch

/soc/i2c@1c2ac00/touchscreen@5d
```

This path can now be supplied as `-select` argument to another call of the extract tool to create pruned device tree that contains only nodes that are related to the selected one. For further inspection, we redirect the pruned device tree to the file *touch.dts*.

[1] https://github.com/genodelabs/genode/tree/master/tool/dts

```
genode$ ./tool/dts/extract --select /soc/i2c@1c2ac00/touchscreen@5d \
                           flat_pinephone.dts > touch.dts
```

For us, the lines featuring a `compatible` string are of immediate interest because those strings draw a direct relation to Linux source codes.

```
genode$ grep compatible touch.dts

 compatible = "fixed-clock";
 compatible = "fixed-clock";
 compatible = "simple-bus";
  compatible = "allwinner,sun50i-a64-ccu";
  compatible = "allwinner,sun50i-a64-pinctrl";
  compatible = "allwinner,sun6i-a31-i2c";
  compatible = "arm,gic-400";
  compatible = "allwinner,sun50i-a64-rtc",
  compatible = "allwinner,sun50i-a64-r-intc",
  compatible = "allwinner,sun50i-a64-r-ccu";
  compatible = "allwinner,sun50i-a64-r-pinctrl";
  compatible = "allwinner,sun8i-a23-rsb";
 compatible = "goodix,gt917s";
 compatible = "x-powers,axp803";
compatible = "pine64,pinephone-1.2", "pine64,pinephone", "allwinner,sun50i-a64";
```

The actual touchscreen device is the “goodix,gt917s”. A few other nodes are already familiar from the work with the display driver. All devices prefixed with “r-" belong to the so-called RTC-related part of SoC, which can be powered independently from the application processor and are designated for always-on functionality. The “rsb” (reduced serial bus) is the two-wire bus that interconnects the SoC with the power-management IC “x-powers,axp803”. The “i2c” and “pinctrl” relation becomes apparent when looking at the schematics of the PinePhone or the datasheet of the Goodix touchscreen controller.

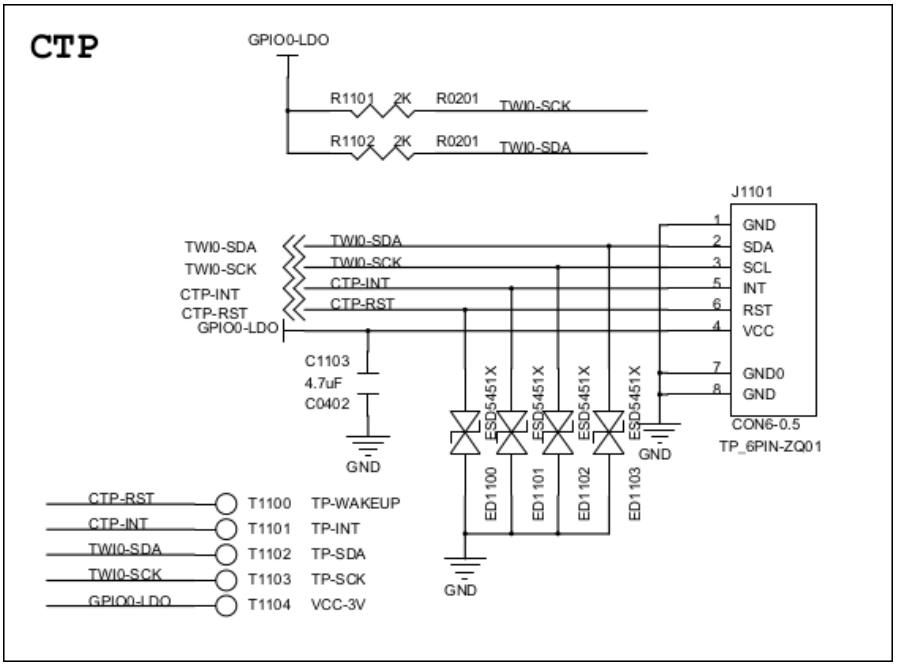


Figure 12

"CTP" presumably stands for capacitive touch panel. It is powered by same "GPIO-LDO" voltage that we encountered for the display driver before. As this signal comes from the AXP803 power-management IC and is disabled by default, this explains the need for talking over the RSB bus with the PMIC. The actual touch data travels via the signals "TWIO-SDA" and "TWIO-SCK", which are the data and clock lines of an I2C connection. The separate "CTP-INT" signal allows the touch panel to notify the application processor whenever something interesting happens - that is - when it would be worthwhile to request new data via I2C. Finally, the "CTP-RST" signal is a reset line driven by the application processor.

So long for getting an overview of how the touch panel is integrated. Now, let's figure out the Linux source code of interest. The procedure follows the same pattern as employed for the display and network drivers.

First, we search the Linux source tree for the `compatible` strings as present in the device nodes.

```
linux$ grep -r "goodix,gt917s"

drivers/input/touchscreen/goodix.c: { .compatible = "goodix,gt917s" },
```

Having found an interesting location, look sideways, in particular at *drivers/input/-touchscreen/Makefile* in this case. The following line draws the connection to a kernel configuration option.

```
obj-$(CONFIG_TOUCHSCREEN_GOODIX)  += goodix.o
```

This gives us a new clue what to `grep` for.

```
src/linux$ grep -r TOUCHSCREEN_GOODIX

drivers/input/touchscreen/Makefile:obj-$(CONFIG_TOUCHSCREEN_GOODIX) += goodix.o
drivers/input/touchscreen/Kconfig:config TOUCHSCREEN_GOODIX
```

The `Kconfig` file mentioning the kernel option gives us a few more hints.

```
config TOUCHSCREEN_GOODIX
        tristate "Goodix I2C touchscreen"
        depends on I2C
        depends on GPIOLIB || COMPILE_TEST
        help
          Say Y here if you have the Goodix touchscreen (such as one
          installed in Onda v975w tablets) connected to your
          system. It also supports 5-finger chip models, which can be
          found on ARM tablets, like Wexler TAB7200 and MSI Primo73.

          If unsure, say N.

          To compile this driver as a module, choose M here: the
          module will be called goodix.
```

Observing the driver in a minimal Linux kernel At this point, it is time to expand our bare-bones Linux configuration in *a64_linux/target.inc*.

```
LX_ENABLE += INPUT_TOUCHSCREEN
LX_ENABLE += TOUCHSCREEN_GOODIX
```

To see if and how those options come into effect, it's best to build the kernel with the changed configuration...

```
build/arm_v8a$ make a64_linux
```

and then manually inspect the *a64_linux/.config* file, in particular searching for "TOUCHSCREEN_GOODIX" to see if all config dependencies are met. If not, we have

to study the Kconfig files to see missing options. E.g., the "TOUCHSCREEN_GOODIX" option is evaluated only if "INPUT_TOUCHSCREEN" is enabled.

This procedure needs to be repeated for all `compatible` strings we identified as interesting above.

E.g., the "allwinner,sun6i-a31-i2c" compatible string leads us to *drivers/i2c/busses/i2c-mv64xxx.c*. The accompanied *Makefile* speaks of "CONFIG_I2C_MV64XXX". So we add this one to our kernel configuration.

```
LX_ENABLE += I2C_MV64XXX
```

After a few iterations of enabling kernel options, building the kernel, and test-driving it on the PinePhone, we are greeted by the driver:

```
Goodix-TS 0-005d: ID 917S, version: 0200
Goodix-TS 0-005d: Failed to invoke firmware loader: -22
Goodix-TS: probe of 0-005d failed with error -22
```

Looking into the code that prints the message "Failed to invoke firmware loader" reveals that a call to `request_firmware_nowait` fails with the error code EINVAL. This happens because the kernel falls back to the dummy function at *include/linux/firmware.h* unless the kernel option FW_LOADER is enabled. I guess, you know what comes next:

```
LX_ENABLE += FW_LOADER
```

On the next iteration, the output looks different.

```
Goodix-TS 0-005d: ID 917S, version: 0200
Goodix-TS 0-005d: Direct firmware load for goodix_917S_cfg.bin failed with error -2
input: Goodix Capacitive TouchScreen as /devices/platform/soc/1c2ac00.i2c/i2c-0/0-005d/input
```

In principle, we could add further kernel infrastructure to expose the driver as input/event interface in order to access it from the Linux user land. One useful tool is the `evbug` kernel module, which prints each occurring input event to the kernel log. It can be activated by enabling the kernel-configuration option `INPUT_EVBUG`. Alternatively, an easy way to see the immediate driver responding to touch input is to instrument the driver code directly. In the particular case, the function `goodix_process_events` is a suitable hook. Adding a `printk` as follows does the trick.

```
static void goodix_process_events(struct goodix_ts_data *ts)
{
  ...
  touch_num = goodix_ts_read_input_report(ts, point_data);
  printk("goodix_process_events got %d touch_num events\n", touch_num);
  ...
```

Upon the next try, we can see that the driver indeed receives touch events!

To crosscheck the minimal set of Linux configuration options that are required for the touchscreen driver to work, it is useful to comment out all `LX_ENABLE` lines in the a64_linux/target.inc file that are seemingly unrelated to the touchscreen device and test the resulting Linux kernel. This way, we end up reaching the following set of options.

```
LX_ENABLE += MFD_AXP20X_RSB REGULATOR REGULATOR_AXP20X
LX_ENABLE += INPUT_TOUCHSCREEN TOUCHSCREEN_GOODIX I2C I2C_MV64XXX FW_LOADER
```

Hosting the touchscreen driver code in a Genode component Equipped with the display driver as blue print, we can mirror the basic structure of a DDE-Linux-based driver component from the display driver to an appropriate location. In our case, this would be the *src/drivers/touch/goodix/* directory[1] in the genode-allwinner[2] repository.

The development procedure follows the same lines as described in Section 2.12.3. For testing the input driver in isolation without any GUI infrastructure, the event_dump[3] server is a handy tool.

Bridging Genode's C++ world with the Linux world For bridging the gap between the Linux kernel and Genode's `Event` session interface, there are two pieces needed. First, the genode_c_api/event.h[4] API provides a simple C API for generating events. As of now, the API is limited to the few event type we have actual drivers for (touch). This free-standing API depends neither on Genode nor on Linux headers. The second piece is a custom emulation code for Linux' input subsystem contained in input.c[5]. It responds to (Linux-internal) calls of the emulated input subsystem by invoking the *genode_c_api/event.h* API.

[1] https://github.com/genodelabs/genode-allwinner/tree/master/src/drivers/touch/goodix
[2] https://github.com/genodelabs/genode-allwinner
[3] https://github.com/genodelabs/genode/tree/master/repos/os/src/server/event_dump
[4] https://github.com/genodelabs/genode/blob/master/repos/os/include/genode_c_api/event.h
[5] https://github.com/genodelabs/genode-allwinner/blob/master/src/drivers/touch/goodix/input.c

Caveats During the work on the driver, I learned a few unexpected lessons that are worth sharing.

Apparently, **time-multiplexing** GPIO pins between input and output are a thing, even outside I2C. In the concrete case of the Goodix touch panel, I struggled matching the Linux driver code against the roles of the signals depicted in the Goodix documentation[1].

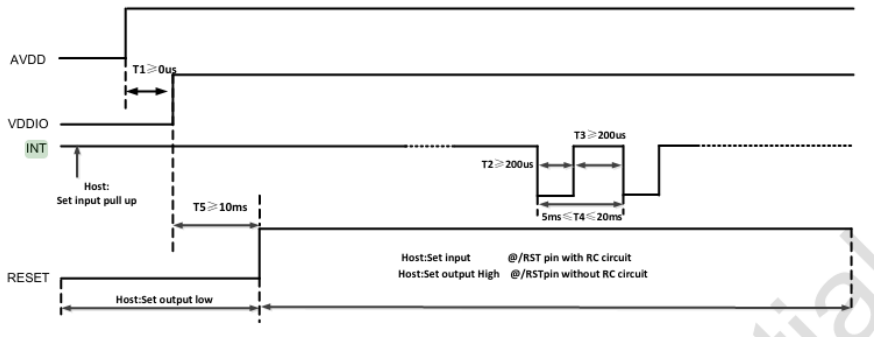


Figure 13

According to this diagram, the RESET signal is driven by the host whereas the INT signal is driven by the Goodix device, which makes perfect sense. In the driver code, however, the INT signal is driven by the host as well! It turns out that certain versions of the device scan the INT signal during reset to obtain one bit of configuration information (choice between two possible I2C addresses).

To accommodate the time multiplexed use of a pin as input or output, Genode's pin driver (a64_pio for the PinePhone) switches an output pin to output mode not before a pin-control client actually accesses the pin. This way, a driver is able to toggle the direction by controlling the lifetime of its pin-control session while sensing the pin via a separate pin-state session.

Device resources needed by the driver As described in Section 2.9, Genode's platform driver restricts access of drivers to devices. During the process of porting a Linux driver as Genode component, one is repeatedly confronted with messages like:

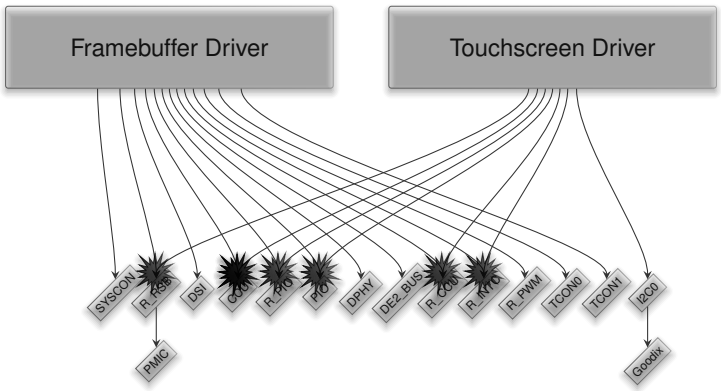
```
Error: memory-mapped I/O resource ... unavailable
```

While addressing those messages by enhancing the platform driver's configuration step by step, the following picture emerges. Note the close correlation with the device-tree information we gathered initially.

[1] https://files.pine64.org/doc/datasheet/pinephone/GT917S-Datasheet.pdf

```
<device name="r_pio" type="allwinner,sun50i-a64-r-pinctrl">
  <io_mem address="0x01f02c00" size="0x400"/>
  <irq number="77"/>
</device>
<device name="r_ccu" type="allwinner,sun50i-a64-r-ccu">
  <io_mem address="0x01f01400" size="0x100"/>
</device>
<device name="r_intc" type="allwinner,sun6i-a31-r-intc">
  <io_mem address="0x01f00c00" size="0x400"/>
  <irq number="64"/>
</device>
<device name="r_rsb" type="allwinner,sun8i-a23-rsb">
  <io_mem address="0x01f03400" size="0x400"/>
  <irq number="71"/>
</device>
<device name="ccu" type="allwinner,sun50i-a64-ccu">
  <io_mem address="0x01c20000" size="0x400"/>
</device>
<device name="pio" type="allwinner,sun50i-a64-pinctrl">
  <io_mem address="0x01c20800" size="0x400"/>
  <irq number="43"/> <!-- Port B -->
  <irq number="49"/> <!-- Port G -->
  <irq number="53"/> <!-- Port H -->
</device>
<device name="i2c0" type="allwinner,sun6i-a31-i2c">
  <io_mem address="0x01c2ac00" size="0x400"/>
  <irq number="38"/>
</device>
```

This picture is concerning because there is apparently a significant overlap of resources accessed by the display driver (Section 2.13) and those resources needed by the touchscreen driver to operate. In Linux, both drivers are part of the same program, the Linux kernel. But on Genode, we end up in the situation of having two independent programs trying to drive the same parts of the hardware.



The resolution of those conflicts is covered by the next Section.

2.15 Cutting Linux-driver competencies

The previous sections covered the challenges of transplanting complex driver code from the Linux kernel into Genode components. Once running happily in its new habitat, however, the driver code needs a heavy dose of domestication. This section shows how to curb the driver code from the overarching access of power, reset, pin, and clock controls.

At the end of the previous section, we encountered conflicting hardware access as a hard problem when integrating multiple driver components into one system. It naturally arises on the attempt to combine the framebuffer driver with the touchscreen driver.

Each of both drivers assumes the responsibility of managing the clocks, reset lines, pins, and power domains related to the driven devices. As those low-level hardware resources are controlled via system-global hardware-configuration registers, each driver tries to manipulate those central registers. In the concrete scenario, we can observe the following legitimate interplays.

- Each driver tries to enable an output at the power-management IC (PMIC) that happens to power both the LCD display and the touchscreen controller. The PMIC is accessed via a so-called reduced serial bus (RSB) two-wire bus. Therefore, both components concurrently try to drive the same RSB bus controller.

- The touchscreen driver modifies the SoC's pin configuration for the four pins connected to the Goodix touchscreen controller, in particular defining one pin as input signal for interrupt delivery, one pin as output signal for reset control, and selecting I2C as pin function for the two I2C wires.

  The framebuffer driver modifies the pin configuration to assign PWM as pin function for the brightness control, and defines two pins as outputs for controlling the backlight and LCD reset.

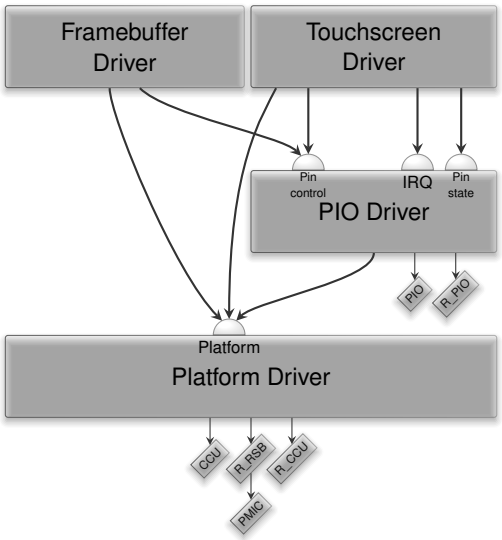
  In both cases, the pins are configured via the system-global PIO device.

- Both drivers interact with the clock and reset control unit (CCU). The touchscreen driver de-asserts the reset signal of the SoC's I2C controller and enables the corresponding bus clock, whereas the framebuffer driver controls the clocks and reset domains of the display engine, MIPI-DSI, DPHY, and the two TCON channels.

There are two principle approaches for the reconciliation of both drivers. One could be tempted to co-locate both drivers into a single component. But this is bad for two reasons. First, it would effectively turn this highly complex component into the central authority over system-management controls, literally yielding power over the whole system, including low-complexity security-sensitive Genode components unrelated to the drivers. Second, with each driver added, this component would grow bigger. Down

the line, we would ultimately end up with an all-powerful monolithic driver subsystem that stands in the way of a clean separation of concerns. E.g., in contrast to an individual framebuffer driver that can be started, removed, and restarted on demand, a monolithic driver component that includes drivers for persistent storage couldn't be restarted without risking data loss.

The right way to go is the consequent removal of low-level system-control access from the drivers. In Genode, the natural place for clock and power management functionality is the platform driver we introduced in Section 2.9, whereas the pin-MUX configuration and GPIO access are covered by the dedicated PIO driver component. The following illustration shows the aspired architecture.



The framebuffer and touchscreen drivers no longer access the low-level system-control registers directly. Instead, the platform driver controls the reset, clock, and power states depending on the presence of its clients. Analogously, the driver's direct GPIO access of the direct pin-MUX manipulation is replaced by the use of the services provided by the PIO driver component.

2.15.1 SoC-aware platform driver

The picture above calls for the enhancement of the platform driver with SoC-specific driver code for controlling clocks, power, and reset lines. Instead of laying those controls into the hands of the driver, the platform driver implicitly drives them based on the mere presence of a related platform client. For example, the following policy assigns the "tcon0" device to the client labeled as the framebuffer driver.

```
<policy label="fb_drv -> " info="yes">
    ...
    <device name="tcon0"/>
    ...
</policy>
```

The “tcon0” device is declared as follows. Note the declaration of the <clock>, <power-domain>, and <reset-domain> sub nodes. The names of listed resets, clocks, and power outputs are defined by the implementation of the SoC-aware platform driver.

```
<device name="tcon0" type="allwinner,sun8i-a83t-tcon-lcd">
    <io_mem address="0x01c0c000" size="0x1000"/>
    <irq number="118"/>
    <reset-domain name="tcon0"/>
    <reset-domain name="lvds"/>
    <power-domain name="pmic-gpio0"/>
    <clock name="bus-tcon0" driver_name="ahb"/>
    <clock name="tcon0"     driver_name="tcon-ch0"/>
    <clock name="dummy"     driver_name="tcon-pixel-clock"/>
</device>
```

Given this information, the platform driver knows that the framebuffer driver depends on the reset lines “tcon0” and “lvds” being de-asserted. It also knows that the driver requires the powering of the “pmic-gpio0” output of the PMIC chip. It also knows that the clocks “ahb”, “tcon-ch0”, and “tcon-pixel-clock” must be set up. Once the framebuffer driver connects to the platform driver, the platform driver can establish all those requirements implicitly while establishing the connection. This not only relieves the actual driver from those low-level peculiarities. It also enforces the proper reset of these settings whenever a driver disappears - with no active participation of the driver needed. For reference, the implementation of the A64-specific platform driver can be found at the following link.

SoC-aware platform driver for the Allwinner A64 SoC

https://github.com/genodelabs/genode-allwinner/tree/master/src/drivers/platform/a64.

To simplify the implementation of the clock, reset, and power drivers within the SoC-specific platform driver, the generic platform driver[1] offers a few handy utilities in the

[1] https://github.com/genodelabs/genode/tree/master/repos/os/src/drivers/platform

form of the clock.h[1], reset.h[2], and power.h[3] headers. In practice, most system-control operations come down so toggling a single bit for (de-)asserting a reset line, or for (un-)gating a clock.

2.15.2 Curbing the Linux driver code

The SoC-specific platform driver looks fairly simple. However, at the client side - in the Linux driver component - two hairy questions arise.

1. How to remove the direct access of the low-level system-control registers while keeping the Linux code happy? Some the related Linux subsystems, in particular those related to pin-MUX configuration, are quite central to the healthy operation of the Linux kernel. They are mandatory and cannot be ripped out.
2. Which clocks, resets, power domains are actually expected by the driver? Candidates are plenty. The answer seems rather vague and is scattered over many kernel modules.

2.15.3 Mimicking Linux subsystems

We encounter the first problem when trying to remove the *drivers/pinctrl/sunxi/** driver code, which interacts with the PIO device. The Linux code briefly complains and just stops.

```
mv64xxx_i2c 1c2ac00.i2c: can't get pinctrl, bus recovery not supported
```

To lift the clouds a bit, it helps to enable the debug messages in Linux *drivers/base/dd.c* and *drivers/base/core.c* by placing the following line at the top of those files.

```
#define DEBUG
```

This results in a very plausible message.

```
i2c 0-005d: probe deferral - supplier 1c20800.pinctrl not ready
i2c 0-005d: Added to deferred list
```

[1] https://github.com/genodelabs/genode/blob/master/repos/os/src/drivers/platform/clock.h
[2] https://github.com/genodelabs/genode/blob/master/repos/os/src/drivers/platform/reset.h
[3] https://github.com/genodelabs/genode/blob/master/repos/os/src/drivers/platform/power.h

The I2C subsystem depends on the pinctrl driver, which we just removed. To satisfy this dependency without using the actual pinctrl driver, we have to create a custom kernel module that looks like the original pinctrl but is just an almost empty hull. To create such a stub driver, the easiest way is to start looking at the original driver code and mirroring its basic structure. The original driver code can be found by inspecting the device tree, which contains the following node.

```
 pio: pinctrl@1c20800 {
  compatible = "allwinner,sun50i-a64-pinctrl";
```

By searching for the compatible string inside the *drivers/pinctrl/sunxi* directory, we find the right spot.

```
src/linux$ grep "allwinner,sun50i-a64-pinctrl" drivers/pinctrl/sunxi/*
drivers/pinctrl/sunxi/pinctrl-sun50i-a64.c:  { .compatible = "allwinner,sun50i-a64-pinctrl",
```

To mirror the driver's structure, it is good to start looking at the "allwinner,sun50i-a64-pinctrl" string and follow its tracks. It appears inside a table of `of_device_id` entries.

```
static const struct of_device_id a64_pinctrl_match[] = {
  { .compatible = "allwinner,sun50i-a64-pinctrl", },
  {}
};
```

The table `a64_pinctrl_match` is referenced by a `struct` called `a64_pinctrl_driver`.

```
static struct platform_driver a64_pinctrl_driver = {
  .probe  = a64_pinctrl_probe,
  .driver = {
    .name   = "sun50i-a64-pinctrl",
    .of_match_table = a64_pinctrl_match,
  },
};
builtin_platform_driver(a64_pinctrl_driver);
```

The struct refers to a probe function `a64_pinctrl_probe`.
The implementation is merely a wrapper around `sunxi_pinctrl_init`.

```
static int a64_pinctrl_probe(struct platform_device *pdev)
{
  return sunxi_pinctrl_init(pdev,
          &a64_pinctrl_data);
}
```

Piece by piece, we assemble a custom puzzle of (almost) empty functions and structures. By using the exact same symbol names as the original driver, our stub driver nicely overtakes its position, in particular our initcall is called at the appropriate time.

Fast forward, the complete stub driver for replacing the pinctrl driver can be found at src/lib/lx_emul/a64/pio.c[1]. It is arguably not trivial, which is due to the fact that the touchscreen driver uses one GPIO pin as interrupt source. Hence, our stub needs to mimick an interrupt controller as well. The interaction with Genode's PIO driver is done via calls to the lx_emul_pin_* API[2].

The clue for sneaking our stub driver into Linux as GPIO driver is the function `gpiochip_add_data`, which takes a `gpio_chip` struct as argument. This struct defines a number callbacks that our stub driver provides for interacting with the pins. The following two callbacks are especially interesting.

`of_xlate` is called with the coordinates of a GPIO pin as arguments and returns a pin number. The organization of the namespace for such pin numbers is up to us. As the PIO pins of the A64 SoC are organized in a number of banks with 32 pins per bank, a suitable naming scheme is

```
number = bank*PINS_PER_BANK + pin_within_bank
```

`set` sets an output pin of a given number (according the result of `of_xlate`) to the specified level. This function triggers the physical effect.

Compared to the removal of the pinctrl subsystem, replacing the reset (*drivers/reset/**) and clock (*drivers/clk/**) controls is relatively simple. For stubbing the clock control, there already exists a reusable stub driver within the *repos/dde_linux* repository at lx_emul/shadow/drivers/clk/[3].

[1] https://github.com/genodelabs/genode-allwinner/blob/master/src/lib/lx_emul/a64/pio.c

[2] https://github.com/genodelabs/genode/blob/master/repos/dde_linux/src/include/lx_emul/pin.h

[3] https://github.com/genodelabs/genode/tree/master/repos/dde_linux/src/lib/lx_emul/shadow/drivers/clk

2.15.4 Gathering the required clock, reset, and power controls

The second tricky question is to find out the few needles in the haystack of clock, reset, and power controls that are required by the individual driver. There may be more than a dozen of such prerequisites. When missing merely one, the driver won't work.

Of course, the device tree is always a nice reference to start with. Specifically the `clocks` and `reset` properties of the device tree provide useful clues. For example, the device node for the `tcon1` contains the following declarations.

```
tcon1: lcd-controller@1c0d000 {
  ...
  clocks = <&ccu 48>, <&ccu 101>;
  clock-names = "ahb", "tcon-ch1";
  resets = <&ccu 25>;
  reset-names = "lcd";
  ...
```

The numbers can be correlated in definitions found at *include/dt-bindings/clock/sun50i-a64-ccu.h* in the Linux source tree.

```
...
#define CLK_BUS_TCON1 48
...
```

Those definitions, in turn, show up in the driver's source tree - in this particular case *drivers/clk/sunxi-ng/ccu-sun50i-a64.c* - which draws the connection to the physical coordinates of the clock.

```
...
static SUNXI_CCU_GATE(bus_tcon1_clk,  "bus-tcon1",  "ahb1",
                      0x064, BIT(4), 0);
...
static struct clk_hw_onecell_data sun50i_a64_hw_clks = {
  .hws  = {
    ...
    [CLK_BUS_TCON1]   = &bus_tcon1_clk.common.hw,
    ...
```

Now, a look into the CCU documentation for the 4th bit of the register 0x64 should close the circle, prompting us to add an appropriately named clock definition Genode's platform driver.

I/O register tracing That said, unfortunately it is all too easy to miss one piece of the puzzle when merely looking from *above* (from the device tree). In this case, a look from *below* may help to complete the picture: To find the right bits - and also to quickly rule out the wrong ones - the low-level tracing of register accesses is sometimes inevitable.

Usually, Linux subsystems come with their own pieces of infrastructure, which provide us with a convenient hook for instrumentation. For example, all parts of the driver for the Allwinner clock and reset unit (CCU) happen to include the file *linux/drivers/clk/sunxi-ng/ccu_common.h*. Hence, changes of this file affect only the driver code we are interested in. So we can add the following hillbilly I/O tracing facility that captures all `writel` and `readl` operations.

```
static inline void my_writel(u32 value, volatile void __iomem *addr)
{
  printk("::: writel 0x%x addr=0x%p\n", value, addr);
  writel(value, addr);
}
#undef writel
#define writel my_writel

static inline u32 my_readl(volatile void __iomem *addr)
{
  u32 result = readl(addr);
  printk("::: readl addr=0x%p -> 0x%x\n", addr, result);
  return result;
}
#undef readl
#define readl my_readl
```

The ":::" prefix is just a band aid to easily distinguish the trace output from regular log output. Note that the instrumentation print virtual addresses though. To correlate those virtual addresses with physical addresses, we can add an instrumentation to the `lx_emul_io_mem_map` function in lx_emul/io_mem.cc[1].

```
log("mapped memory-mapped I/O resource ", Hex(phys_addr),
    " (size=", Hex(size), ") to ", ret);
```

This I/O tracing approach is extremely simple, yet surprisingly powerful. Consider the following ideas.

[1] https://github.com/genodelabs/genode/blob/master/repos/dde_linux/src/lib/lx_emul/io_mem.cc

- One can programmatically filter out superfluous noise by making the `printk` statements conditional. For example, skipping the output for certain uninteresting accesses (like polling a certain bit), or keeping a counter and starting the output not before the counter has reached a certain value.
- Even more interesting is the conditional skipping of write operations to confirm that certain register accesses are *really* needed. One can even change the bits written to the hardware registers to see, e. g., the effect of different clock settings.
- As a sledgehammer approach, one can replay a once gathered register trace at the startup of Genode's platform driver and skip as many `writel` operations in the driver as possible.

By iteratively tweaking the filtering, thousands of register accesses during the initialization of the touchscreen driver could be condensed to the following few *interesting* accesses. With the focus drawn to such a few registers, the manual review of the bits suddenly becomes practical.

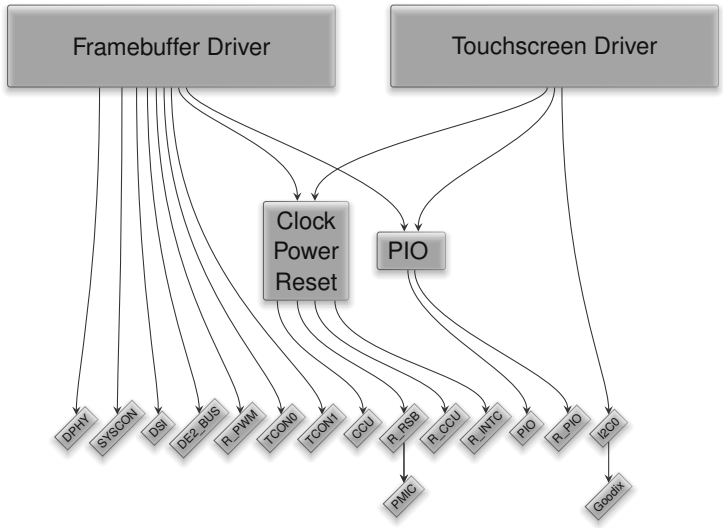
```
writel 0x5514     addr=0x2008   PLL_AUDIO control register
writel 0x515      addr=0x2040   PLL_MIPI Control Register
writel 0x90001031 addr=0x2000   PLL_CPUX Control Register
writel 0x90002d00 addr=0x204c   PLL_DDR1 Control Register
writel 0x90041811 addr=0x2028   PLL_PERIPH0 Control Register
writel 0x81000002 addr=0x215c   MBUS Clock Register
writel 0x10001    addr=0x206c   Bus Clock Gating Register 3
```

That said, keep in mind that we can never be sure to capture all I/O accesses this way. Drivers may operate in a way that bypass the `readl` and `writel` functions. Also, the place of the instrumentation is important. For example, if a driver accesses registers indirectly using the *drivers/base/regmap/* utilities, one needs to the place the instrumentation inside the regmap implementation.

2.15.5 The drivers reconciled

The result of the described development step is best illustrated by the configuration[1] of the drivers subsystem for Genode's interactive system scenarios on the PinePhone. It clearly documents the *enforced* relationship between the drivers and the related hardware at an almost intuitive level of abstraction. For example, it becomes perfectly clear, which driver has the authority over which GPIO pin. The access to low-level system-management registers is exclusively guarded by the platform driver. Isn't it beautiful?

[1] https://github.com/genodelabs/genode-allwinner/blob/master/recipes/raw/drivers_interactive-pinephone/drivers.config


Framebuffer Driver
Touchscreen Driver
Clock
Power
Reset
PIO
DPHY
SYSCON
DSI
DE2_BUS
R_PWM
TCON0
TCON1
CCU
R_RSB
R_CCU
R_INTC
PIO
R_PIO
I2C0
PMIC
Goodix

2.16 Telephony

This section is based on Sebastian Sumpf's Telephony article[1] at https://genodians.org.

It describes the steps for enabling basic telephony support on the PinePhone.

LTE Modem The PinePhone ships with a stock Quectel EG-25-G[2] LTE capable modem designed for machine to machine communication. Inherently, the modem[3] is a system on a chip (SoC) in itself with its own CPU core (Qualcomm MDM9207), RAM (256MB), and flash memory (256MB). It features next to LTE/UMTS/GSM, audio codecs, audio playback, GPS, and Bluetooth support.

Firmware The firmware is essentially a custom Linux system that PinePhone's Allwinner A64 SoC[4] can interact with through UART, USB, and PCM interfaces. A speciality of the Quectel modem is the support to flash custom firmware into the modem. With this feature, it is possible to execute well-known trusted firmware instead of opaque binary blobs on the modem. An example of such a firmware is the *PinePhone Modem SDK* as provided by Biktorgj[5]. The firmware offers better power-management support because it can scale the CPU frequency down to 100MHz (stock firmware is 400MHz) during sleep. Additionally it offers some features like non-persistent storage, time synchronization from carrier to user space, as well as zero binary blobs unlike the stock firmware. The non-persistent storage is especially interesting because the stock firmware may corrupt the flash memory - and thereby the whole SoC - if not shutdown correctly. This could happen on a crash or in case of battery failure. A downside of the custom firmware is that it is not feature complete, might be unstable, and in the worst case can even brick the modem. For these reasons, we decided to stay with the stock firmware for our telephony experiments.

UART and USB interfaces Traditionally, modems are accessed through serial UART interfaces using the **AT** command set[6] This feature is still provided by all modern modems. It can be used for placing and receiving calls, sending and receiving SMS, managing the SIM card, phone book, selecting carriers, managing audio, or GPS. For a complete list of the AT commands supported by the Quectel EG25-G, please refer to the official manual[7].

[1] https://genodians.org/ssumpf/2022-05-09-telephony
[2] https://wiki.pine64.org/images/8/82/Quectel_EG25-G_LTE_Standard_Specification_V1.3.pdf
[3] https://wiki.pine64.org/wiki/PineModems
[4] https://linux-sunxi.org/images/b/b4/Allwinner_A64_User_Manual_V1.1.pdf
[5] https://github.com/Biktorgj/pinephone_modem_sdk
[6] https://en.wikipedia.org/wiki/Hayes_command_seti
[7] https://wiki.pine64.org/images/1/1b/Quectel_EC2x&EG9x&EG2x-G&EM05_Series_AT_Commands_Manual_V2.0.pdf

The UART of the Quectel modem is connected to UART3 on the Allwinner SoC, which is a standard 16550 UART[1] as used by normal PCs since 1987. A driver for this UART was already present in Genode but only for the output (TX) case, since it was solely used for logging to the serial port. Because for each AT command sent via the UART to the modem a response is generated, we had to extend Genode's UART driver with receive (RX) support. This way, any terminal emulator (e. g., minicom, picocom, screen) can connect to the UART, send AT commands, and receive responses. Unfortunately this will only work if the modem is powered on.

The USB interfaces are used for data connections (2G/3G/4G) because UARTs are not fast enough (i. e., 115200 baud) to handle the data rates - up to 300 MBit/s - supported by modern standards. In this case, the modem exposes a USB network interface in order to send and receive data plus an additional control channel to configure the connection. For a basic description on the workings of data connections, please refer to our LTE modem support for Genode article[2]. Because we were primarily interested in telephony at this point, we did not enable the USB part nor the Allwinner SoC connections of the modem - which in turn conserves power - and handled telephony through the UART interface using AT commands.

2.16.1 Modem startup

As with most SoCs, the modem is not powered during PinePhone's boot process. This requires manual power-on/off handling by Genode. Important to know is that the modem is powered by the battery directly and cannot be started with just the USB charger plugged in. The reason for this is that the power consumption of the modem can be anywhere from a 2 mA (sleep) up to 700 mA (LTE data[3]) and may vary abruptly, which does not go well with power supplies but can be handled by a battery.

The modem is connected through various input/output pins (GPIO) to the Allwinner SoC through the Port controller[4]. Without going into too much detail, there is one pin that enables the power supply from the battery to the modem. Once the power supply is ready, the modem can be powered on through a PWRKEY pin. This pin must be pulled down for at least 500 ms, which will initiate the modem's boot process that can take up to 30 seconds. In order to determine if the modem is powered on, another pin (status) can be observed. When the pin goes low, the modem has finished booting. For Genode, we have implemented this behavior in a small component called modem manager[5].

[1] https://en.wikipedia.org/wiki/16550_UART
[2] https://genodians.org/ssumpf/2020-12-04-mbim
[3] https://wiki.pine64.org/images/2/20/Quectel_EG25-G_Hardware_Design_V1.4.pdf
[4] https://linux-sunxi.org/images/b/b4/Allwinner_A64_User_Manual_V1.1.pdf
[5] https://github.com/genodelabs/genode-allwinner/blob/master/src/drivers/modem/pinephone/main.cc

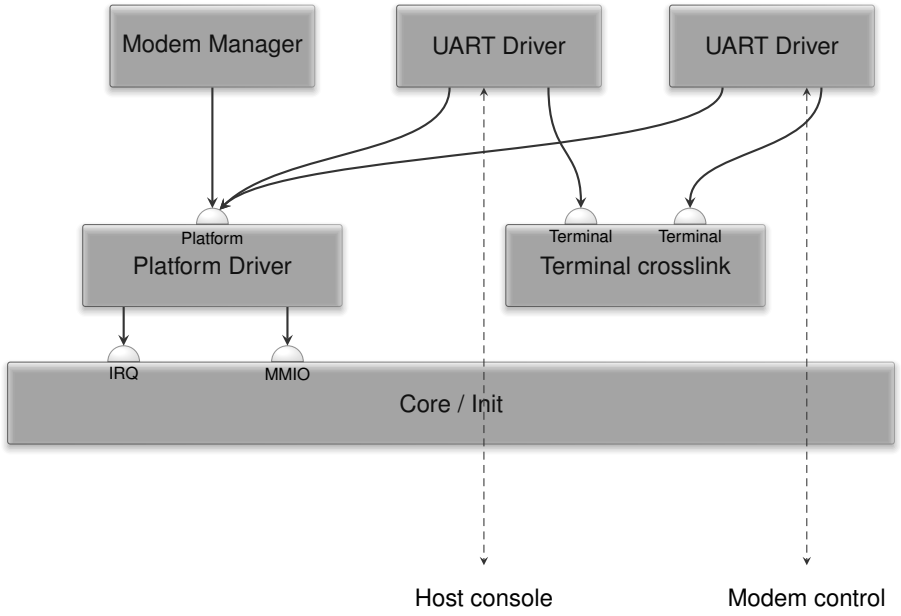


Figure 14: Experimental modem setup on Genode's PinePhone.

Basic telephony scenario With the modem powered, we can connect to the UART of the modem using a terminal emulator and should see:

```
> RDY
```

This means we can now enter AT commands, the simplest would be:

```
AT
> OK
```

Enter the PIN for the SIM card:

```
AT+CPIN=1234
> OK
```

Check the connection status:

```
AT+CREG?
> CGREG: 0,1
> OK
```

This means the modem is connected to the home network. Issue a call:

```
ATD03513231421424;
> OK
```

Hang up:

```
ATH
> OK
```

If one sees a

```
> RING
```

one can accept the call by

```
> ATA
```

The recommended way to shutdown the modem before the PinePhone is powered off is

```
AT+QPOWD
```

With these simple commands, we achieved basic telephony support at an extremely low complexity on the Genode side. The only noteworthy drawback is that no one can hear you and you can hear no one because audio has not been setup yet. We will look into this next.

2.16.2 Audio codec

The Allwinner SoC audio codec is split into two parts. First, an analog part for microphones, speakers, and headsets. And second, a digital part where the SoC itself, the modem, and the Bluetooth interfaces are connected to. As expected, analog-to-digital converters (ADC) and digital-to-analog converters (DAC) bridge both the analog and digital worlds for recording and playback.

Fortunately, we are not the first ones interested in the topology of all the components involved. For example, https://xnux.eu gives valuable insights in the dedicated article Audio on PinePhone[1], in particular an annotated version[2] of the audio diagram depicted in the Allwinner A64 manual. At the top, three different AIFs (audio interface) are displayed. AIF1 leads to the APB bus, which in turn connects to the Allwinner SoC.

[1] https://xnux.eu/devices/feature/audio-pp.html
[2] https://xnux.eu/devices/feature/audio-controls.svg

AIF2 connects to the BB (base band), which is the Quectel modem whereas AIF3 connects to Bluetooth that is not of interest for this line of work. The green boxes ADCL/R and DACL/R are the connection to the analog part of the codec, where the recording and playback devices are connected to.

The whole scenario depicted in the diagram leads to the assumption that the microphone of the PinePhone and the earpiece/speaker can be routed through simple configuration to and from the modem without the actual involvement of the SoC (AIF1). AIF1 exists for playback sounds/music, record audio, and handle data connections like video conferencing where data packets need to be sent to the speaker while microphone inputs need to be transferred to the modem. AIF1 would also require an actual sound driver within Genode. Note that the modem implements this driver locally for itself because it has its own operating system.

Following this idea, Figure 15 depicts the setup we want to achieve for telephony for the left channel. The red arrow marks the path from the microphone to the modem whereas the green arrow shows the path from the modem to the earpiece/speaker.

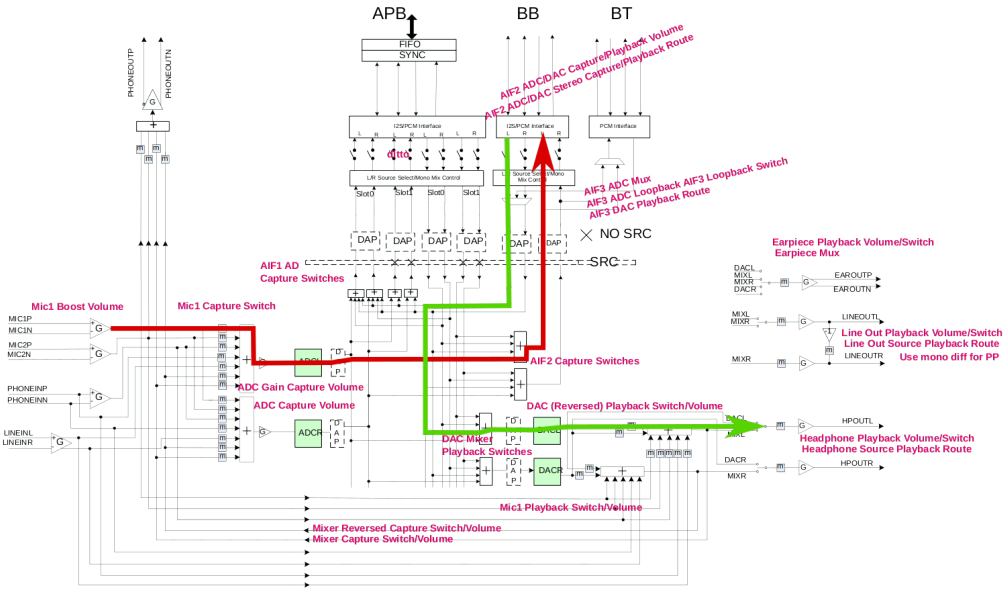


Figure 15: Telephony routes for the left channel

Low-level setup Without going into too much detail, the enablement of any device on ARM platforms is what the BIOS usually does for x86. Meaning - everything is turned off. This step required configuring and activating the Audio PLL, setting up and (un)gateing necessary clocks, bringing the audio codec out of reset state, and configuring the GPIO pins to take and send signals to/from the AIF2 (modem) bus.

Digital audio codec For the digital part, we want to route the ADC where the microphone is connected to the modem (AIF2 mixer), and the playback from the modem to

the DAC where the speaker and the earplugs are connected. For each stage, there exist different mixers that must be configured as well as additional registers to set the data rate, volume, bit width, and audio format. The essential step is to enable ADC (left and right channel) as one source of the AIF2 mixer. There can be more than one source, for example, the AIF2 mixer can also use AIF1 (the Allwinner SoC) as input. For the playback route, we set the source of the DAC mixer to the AIF2 DAC output of the modem. Of course, all parts need to be enabled separately.

Analog audio The analog part of the codec is accessed through a single memory-mapped I/O register called AC_PR (Figure 16). Through this register, 32 8-bit data registers can be addressed through the *Addr* field. In and out data is read/placed by the codec from/into the *Data_in* and *Data_out* fields.

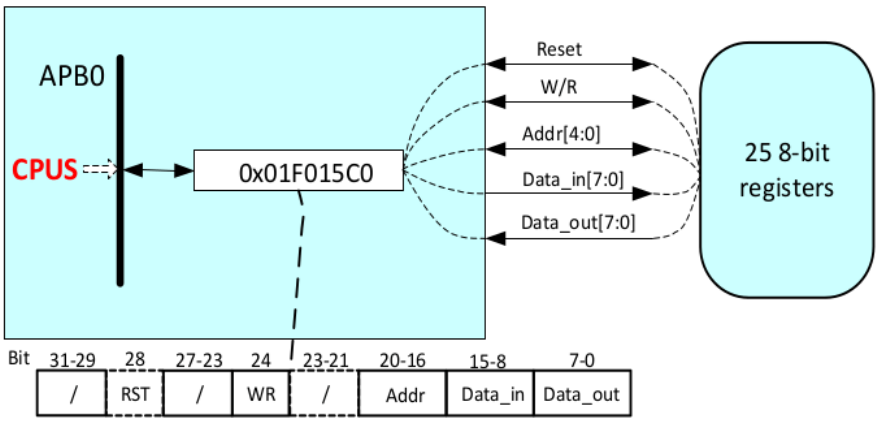


Figure 16: Analog configuration register (Allwinner A64 Manual v1.1, page 362)

With the digital part ready, it was time to activate the actual microphone and the earpiece speaker. In Figure 15 all gray boxes marked with an "m" - for mute - can be configured through the ADC mixer. For the microphone, this means that we have to set the microphone as a source of the ADC mixer. Additionally, every device has a configurable amplifier that needs to be enabled. For the microphone, that is the boost amplifier. Since microphones also require a voltage (bias voltage) this must also be configured. And voila, the microphone was working on a test phone call.

For the earpiece, in turn, we needed to configure the DAC mixer as an input source, enable the amp and unmute the device. This worked out of the box and we were able to perform an actual phone call with the PinePhone running Genode.

As a final test, we enabled the speaker for hand-free communication that is connected to the line out. This works analogously to the earpiece, with the exception that the speaker amplifier is external on the PinePhone and needs to be enabled via a dedicated GPIO pin.

So with everything working it's time for a test call using this run script[1]:
Roger Murdock: We have clearance, Clarence.
Captain Oveur: Roger, Roger. What's our vector, Victor?
Tower voice: Tower's radio clearance, over!
Captain Oveur: That's Clarence Oveur. Over.

[1] https://github.com/genodelabs/genode-allwinner/blob/master/run/modem_pinephone.run