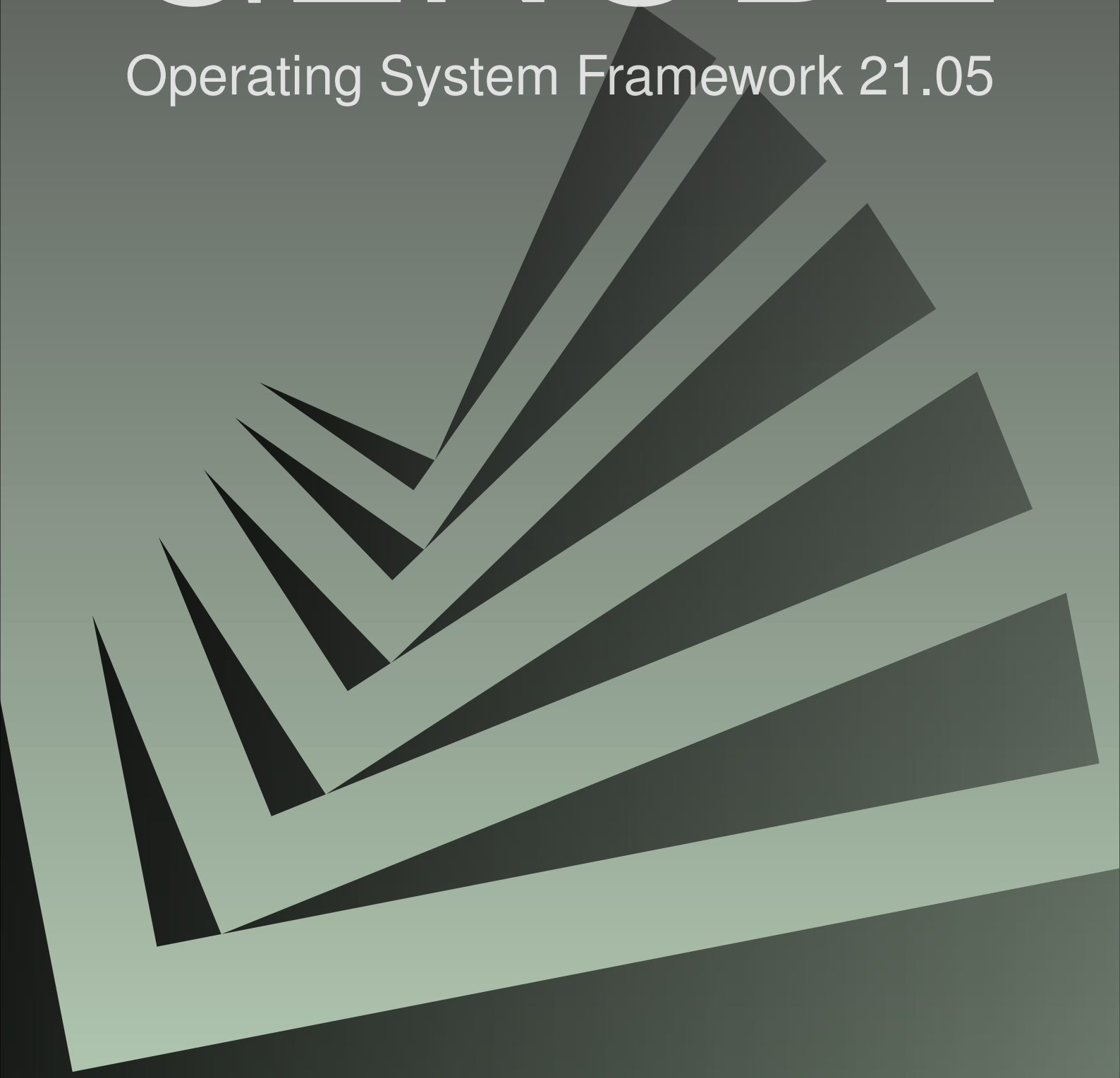
GENODE

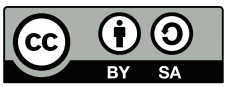
Operating System Framework 21.05



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 https://genodians.org. Note that the document is not set in stone. We plan to continuously extend it with further practical topics as we go.

2 Porting Genode to a new SoC

We get repeatedly asked about the principle steps 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 live.

8. USB host-controller driver

   Genode's USB host-controller driver is based on the Linux USB driver. Adding supplemental support for 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 [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 (GCPL) [2]. 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 or Allwinner 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/* to learn about the various backend modules and options. E.g., look at _tool/run/image/uboot [1] to 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* [2] utility, thereby documenting the steps for reproducing the U-Boot version for your particular board from source.

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

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].

More options can be found at at *tool/run/power_off* [3] and *tool/run/power_on* [4].

For further inspiration, you may also enjoy the article [5].

[1] https://github.com/genodelabs/genode/blob/master/tool/run/log
[2] https://genodians.org/chelmuth/2019-03-13-powerplug
[3] https://github.com/genodelabs/genode/blob/master/tool/run/power_off
[4] https://github.com/genodelabs/genode/blob/master/tool/run/power_on
[5] 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 https://www.pine64.org/ 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.

2.2.1 Getting a first impression

We ordered a Pine64-LTS board [1], 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. Pine64 wiki [4] provides the perfect staring point.

Booting an officially supported GNU/Linux image The wiki lists numerous ready-to-use Linux distributions. I went for https://www.armbian.com. Just a few minutes later, after downloading the disk image [5], 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://dl.armbian.com/pine64so/Buster_current

```
[    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.

Pine-A64-LTS board schematics

https://files.pine64.org/doc/SOPINE-A64/PINE%20A64-TLS-20180130.pdf

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 [1] describing the UART on Pine64. 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 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://linux-sunxi.org/Pine64

```
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 dedicated 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] https://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 variable CONFIG_ENV_FAT_DEVICE_AND_PART. 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 [2] very soon, now would be a good time to install it. 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

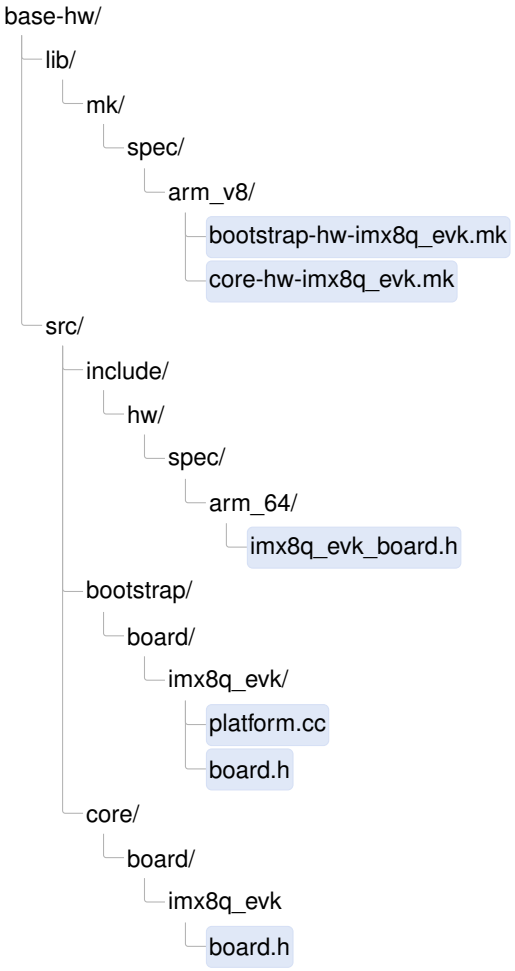
The starting point of our line of work will be the existing board support for the i.MX8Q EVK. To get an idea of the amount of work ahead of us, let's examine the base-hw source tree within Genode for occurrences of the board's name. The search pattern "imx" is a good start.

```
$ 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
```

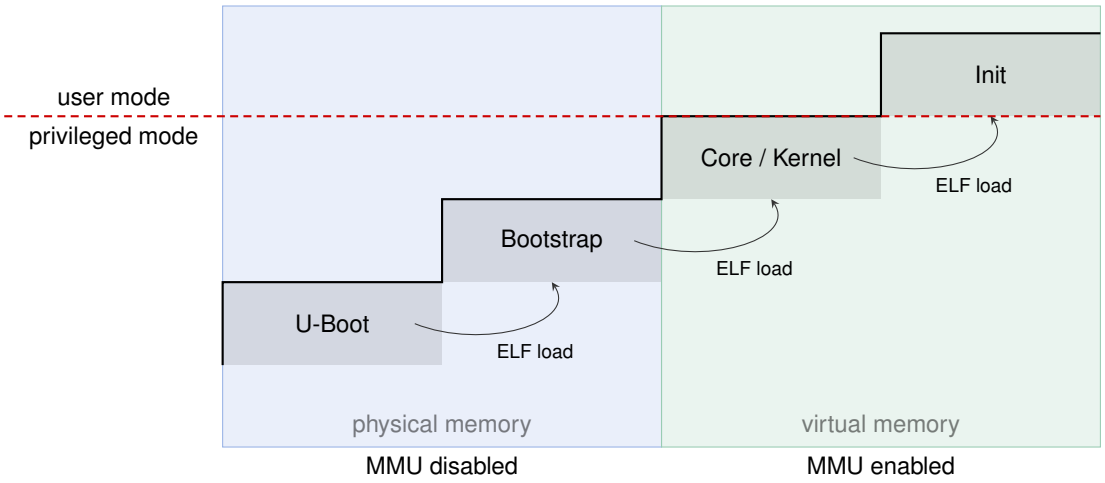
On the one hand, it is nice to see such a small number of files to be concerned about. On the other hand, those files appear quite scattered throughout the source tree with a deep hierarchy, which is a bit confusing. To lift the clouds, let's have a look at the source-tree structure.



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 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 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.

In 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/base-hw/src/include/hw/spec/arm_64/imx8q_evk_board.h

```
#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/base-hw/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)

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/base-hw/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/base-hw/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/base-hw/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/base-hw/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_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.

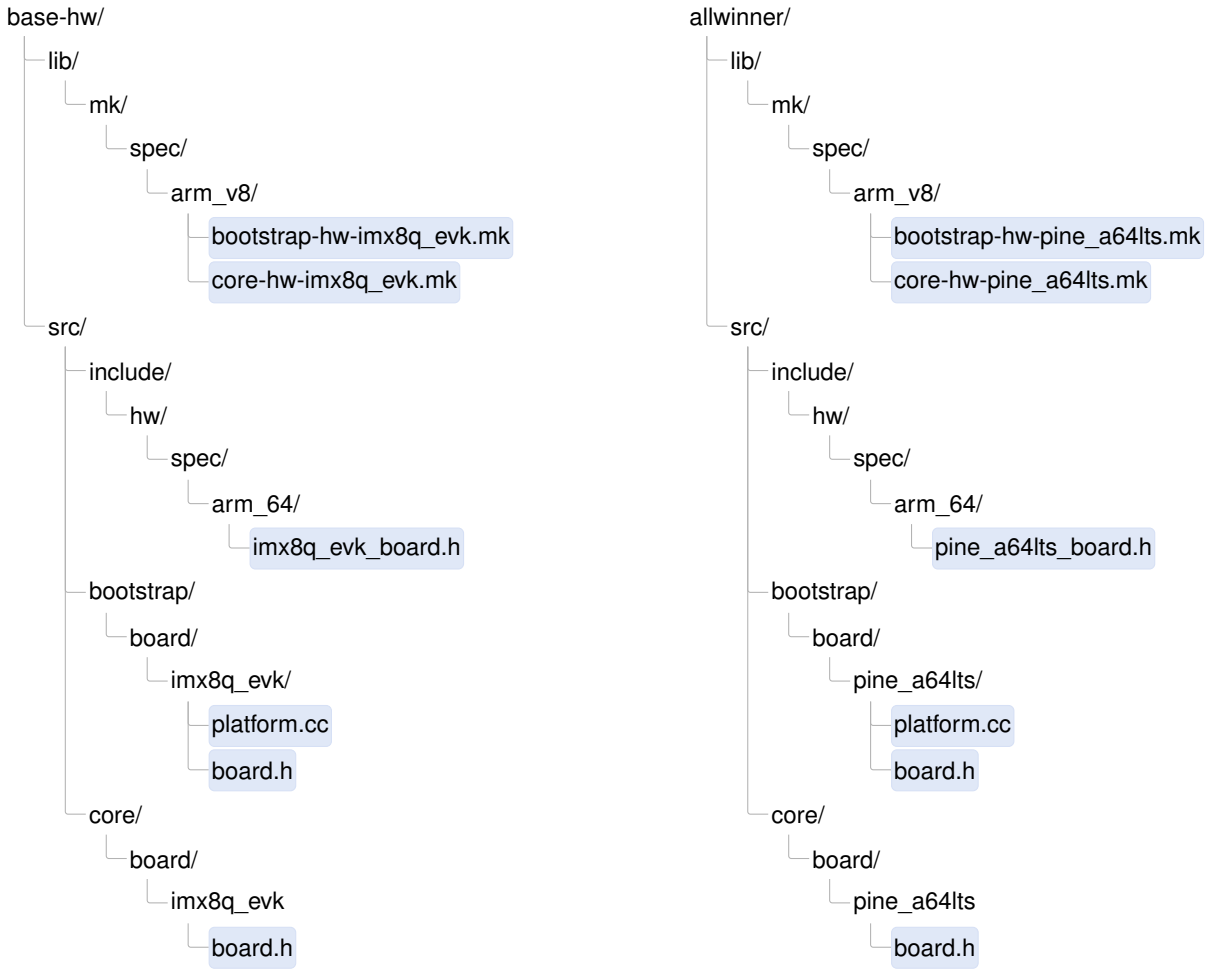
In the following, we will put our code into a new repository named *allwinner*.

```
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
```

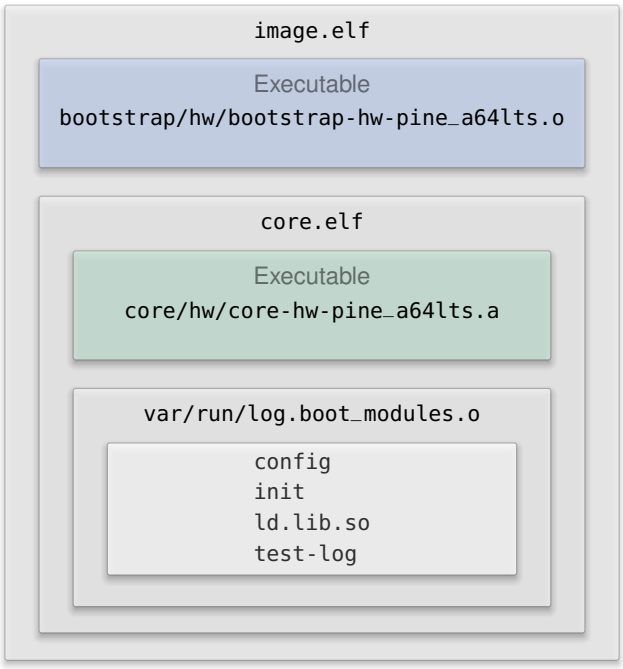
Similar to the build of bootstrap, we can find the result at the corresponding subdirectory, here *core/hw/*. We find a single archive file named *core-hw-pine_a64lts.a* along with a stripped version of this file.

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 dedicated 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*.

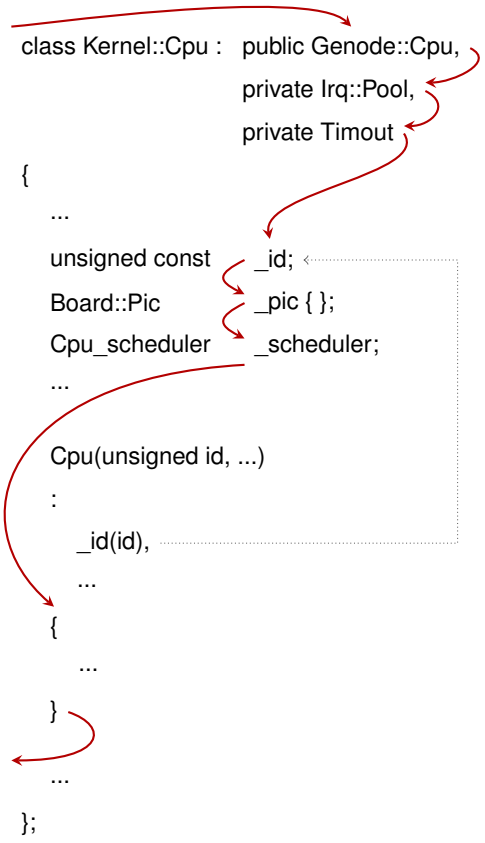
```
bool Cpu_pool::initialize()
{
    unsigned id = Cpu::executing_id();
    _cpus[id].construct(id, _global_work_list);
    return --_initialized == 0;
}
```

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();
}
```

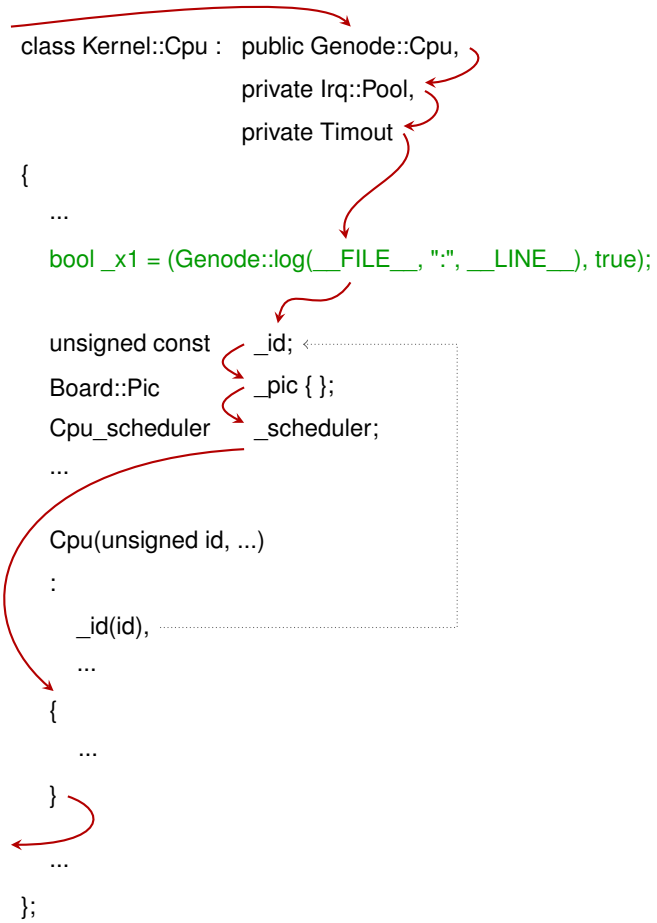
Upon the next run, we see no such message. So we can conclude that we get stuck in the middle of the construction of one of the base classes or aggregated members. As illustrated by the following picture, the body of the constructor is called pretty late in the process of constructing an object.



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 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 as found in the PINE64 Wiki [1] 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.

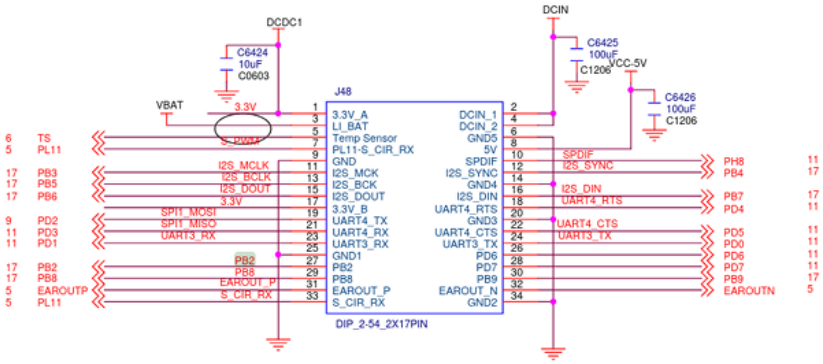


Figure 3

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 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

[1] https://wiki.pine64.org/wiki/PINE_A64-LTS/SOPine

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 010: UART2_RTS 100: JTAG_DO0 110: PB_EINT2	001: Output 011: Reserved 101: SIM_VPPEN 111: IO Disable
7	/	/	/	

Figure 4

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, ...).

2.7.1 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.

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

```
#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 dedicated article about

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
```

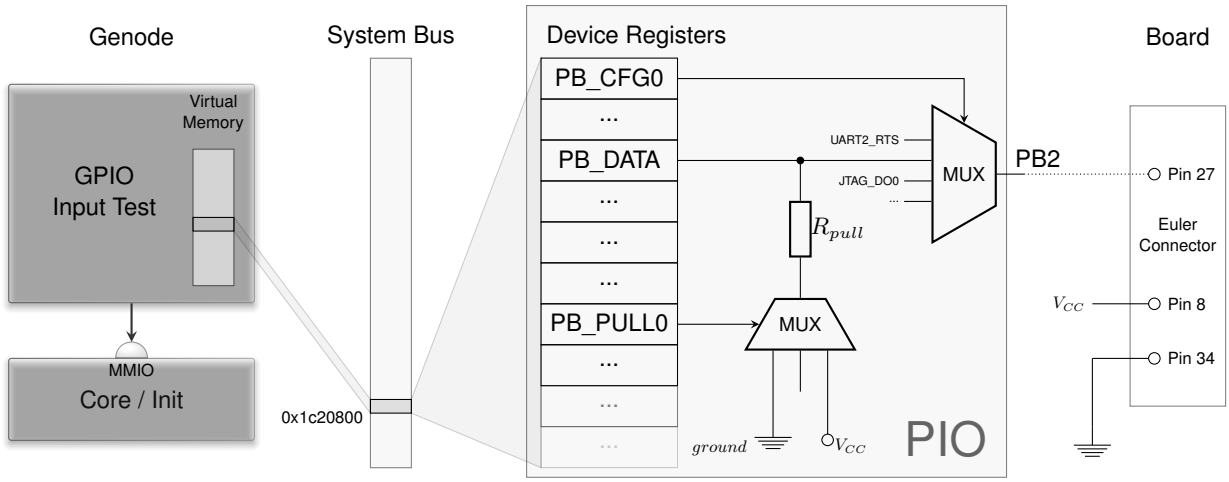
The pattern looks interesting, like if the PB2 pin is not quite sure about its state. For the experiment, let's try to connect the PB2 pin to ground. That is shorting the pins 27 (PB2) with 34 (GND). As a matter of courtesy, it is good to avoid connecting the pins directly but instead placing a resistor 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.

2.7.2 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.

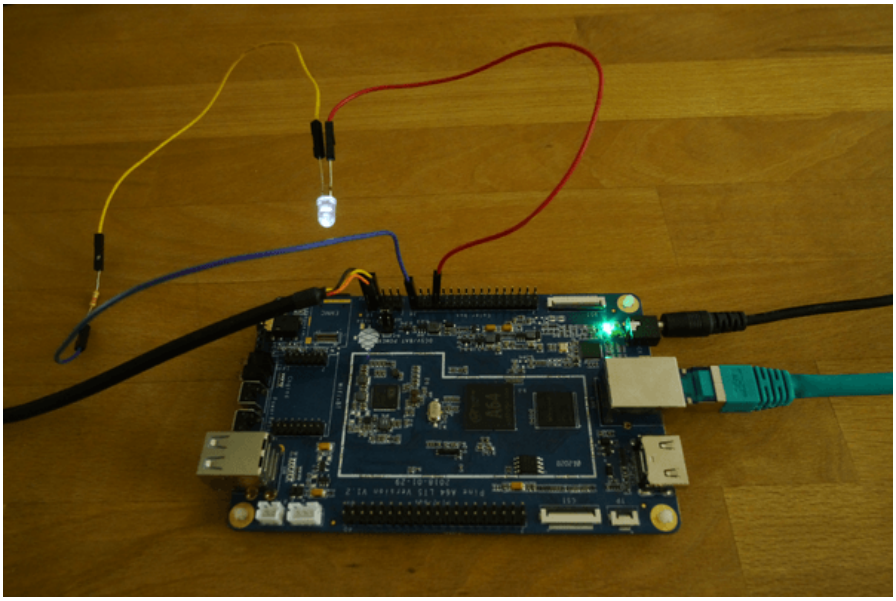


Figure 5

2.7.3 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.

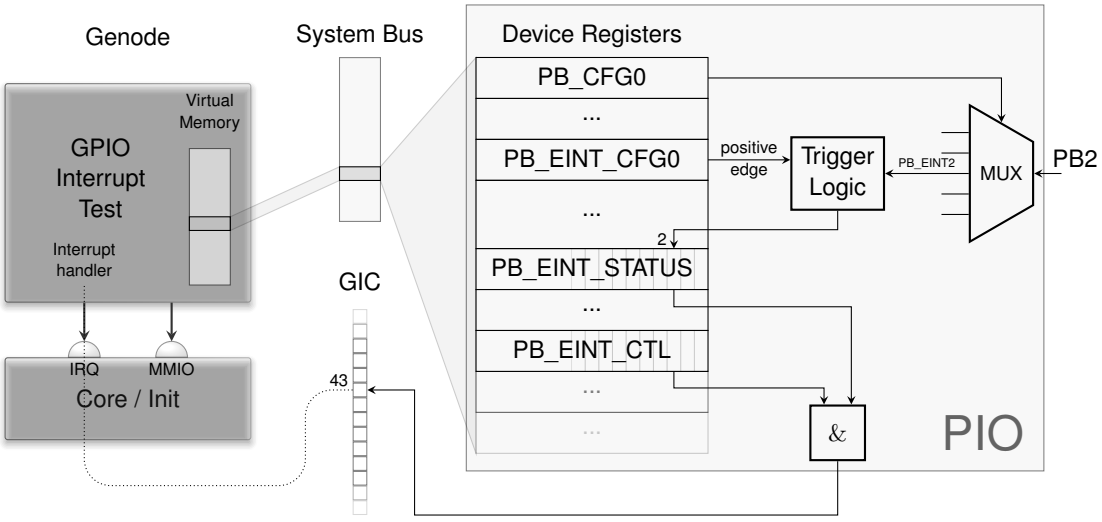
- The GIC interrupt number is passed as argument to the IRQ connection to core.
- Interrupts are delivered as signals. The `_irq_handler` is a signal handler that is registered at the IRQ session via the `_irq.sigh` method. Each time the signal occurs, the `Main::_handle_irq` method is executed.
- The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

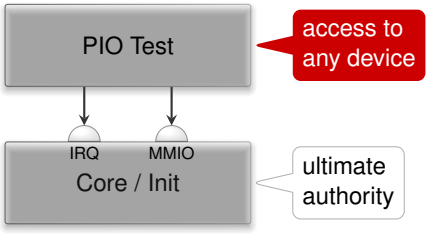
Pointers to the corresponding code The test programs described above can be found at the Genode-Allwinner Git repository [1]. The C++ code described above is located at *src/test/pin_state/*, *src/test/pin_control/*, and *src/test/pin_interrupt/*. These test programs are accompanied with matching run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

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

2.8 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.

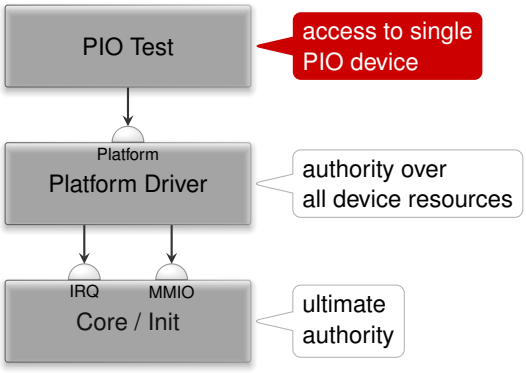


Our user-level test program created connections to core's services for accessing memory-mapped I/O registers and receiving notifications for device interrupts. The choice of physical register addresses and the GIC interrupt number was up to the test program. So in principle, our program could access 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.

2.8.1 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/nfeske/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>
    <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 about the known device resources. 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 2.7.

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
```

To exercise the interplay between the designated PIO driver with the platform driver, it is a good idea to transplant the *test/pin_state* 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

  ```
  #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.

2.8.2 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.

In principle, there may exist legitimate use cases for controlling the direction on an I/O pin by a client, e. g., for implementing a bit-banging driver for a single-wire-bus. However, in practice, we haven't observed such use cases with modern SoC's. Should such a tristate use case pop up, we may address it by dedicated session type.

At the time of writing, the "Pin_state" and "Pin_control" session interfaces still reside local to the genode-allwinner repository. After a period of time-testing, they will eventually become part of the Genode API, replacing the traditional "Gpio" session interface.

2.8.3 PIO device driver

The A64 PIO driver implements the three session interfaces outlined above. It resides at *src/drivers/pin/a64* 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.

```
<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>.

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* [1]. 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* [2], 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* [3] file where all the strings are coming together.

[1] https://github.com/nfeske/genode-allwinner/blob/master/src/drivers/pin/a64/pio.h
[2] https://github.com/nfeske/genode-allwinner/blob/master/src/drivers/pin/a64/types.h
[3] https://github.com/nfeske/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.

2.8.4 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.

2.8.5 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.

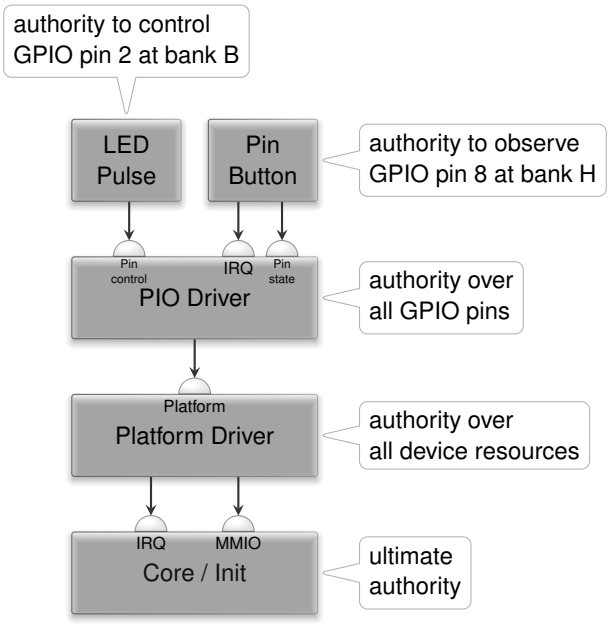
```
<config>
  ...
  <policy label_prefix="pin_event ->" pin="button"/>
  <policy label_prefix="pin_pulse ->" pin="led"/>
</config>
```

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.

2.8.6 Integrated test scenario

The final version of the *a64_pio_drv.run* script [1] 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, the button happens to be also wired to pin 5 of the smaller 10-pin Euler header. I figured that I

[1] https://github.com/nfeske/genode-allwinner/blob/master/run/a64_pio_drv.run

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.

2.9 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* [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 [**?**].

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/
[2] https://graphviz.org/

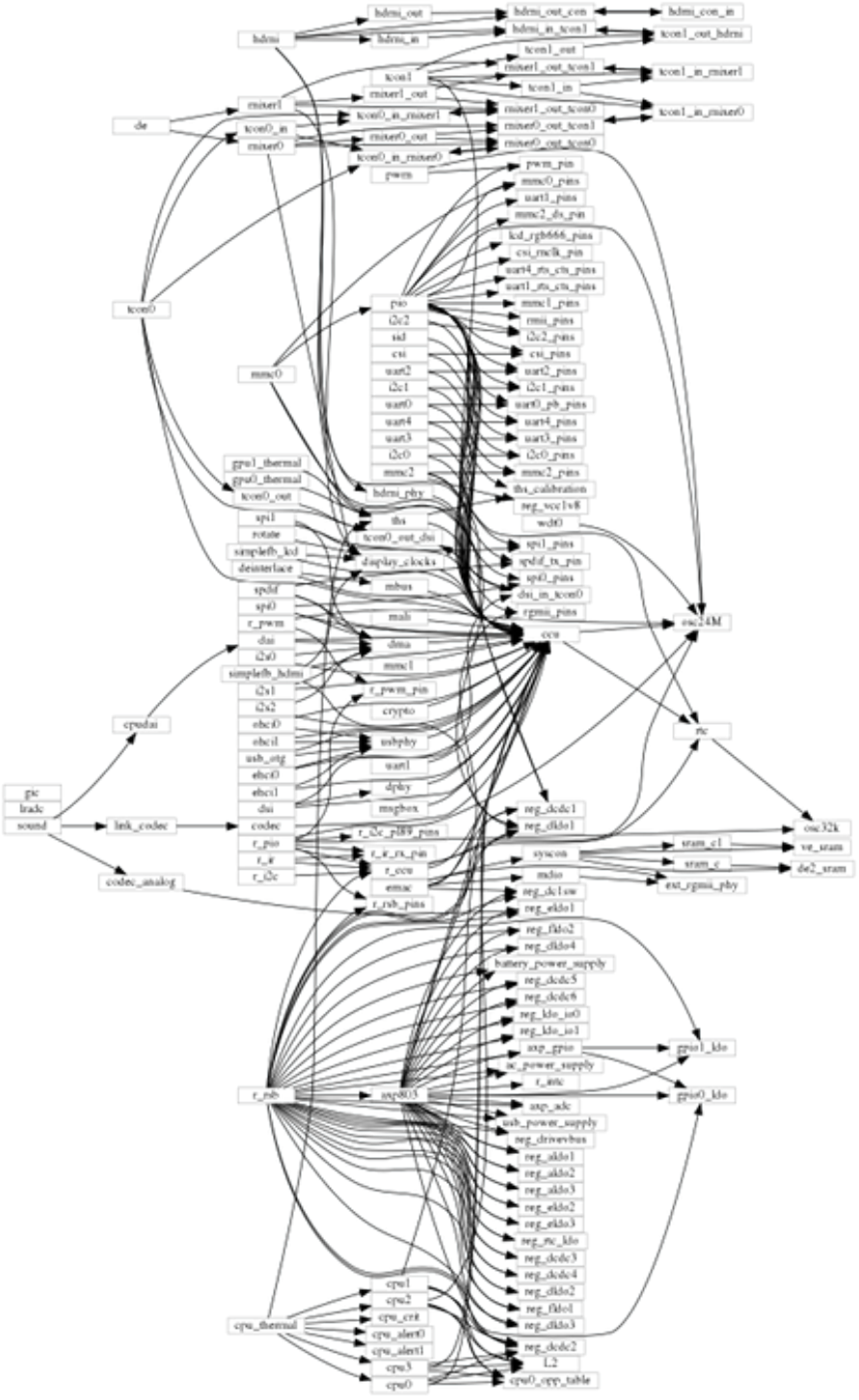


Figure 6: Does this count as generative art?

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.

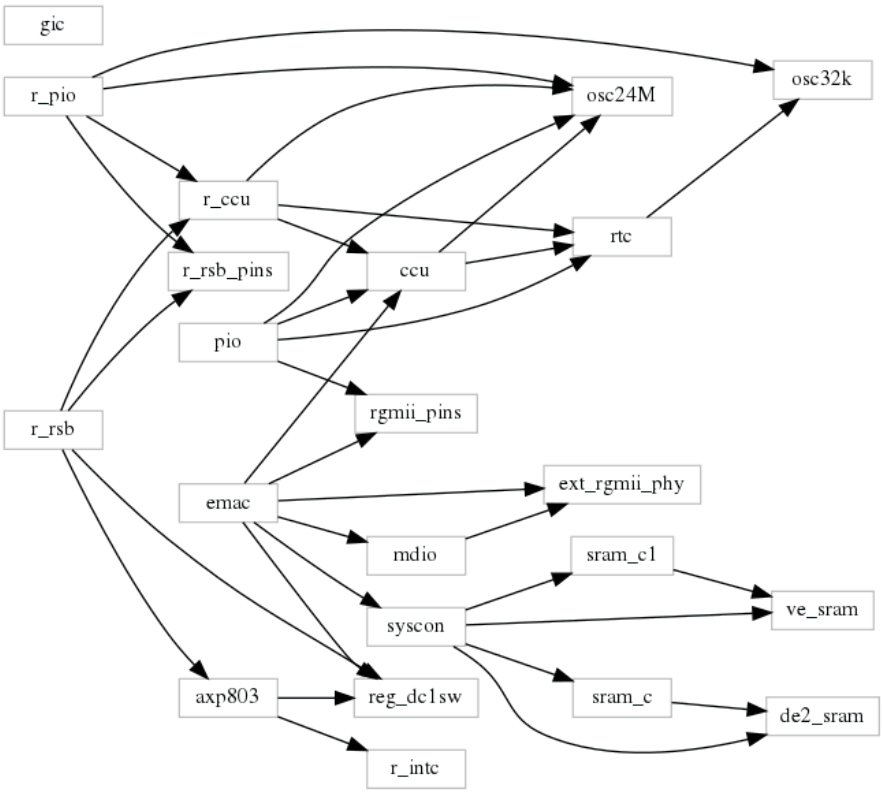


Figure 7: A sudden moment of clarity.

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.

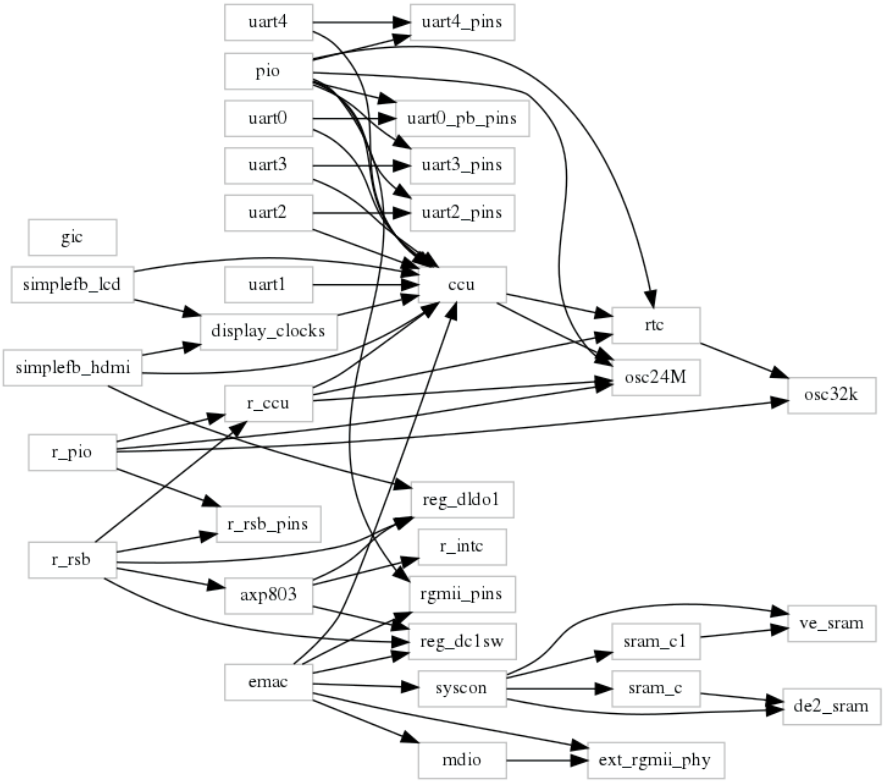


Figure 8: This fine-tuned device tree suffices for using Ethernet with Linux.

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.

2.10 Linux device-driver environment (DDE)

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. Genode's DDE approach combines unmodified driver code taken from the Linux kernel with a driver-specific library that mimics the Linux kernel interface such that the driver code feels right at home when executed on top of Genode. Upcoming revisions of this document will discuss the creation of device driver environments for different classes of device drivers.

At the time of writing, our methodology of crafting DDEs undergoes a fundamental revision. The traditional DDEs as found in the *repos/dde_linux/* part of Genode's source tree used to require significant manual labor per driver. For example, for the porting of a framebuffer driver, we used to estimate an effort of two months. We successfully applied the approach to port drivers for USB host controllers, HID devices, framebuffers, Wifi cards, as well as protocol stacks such as the TCP/IP stack. For reference, the documentation section of https://genode.org provides practical hints for applying the approach [1]. The development costs behind a DDE are arguably cheap compared to the time needed to create such feature-rich drivers from scratch. But each driver still calls for a significant investment.

Based on our experience made during a decade of DDE-Linux work, we recently critically reviewed our methodology and tools and identified ways to dramatically reduce the effort. Stefan Kalkowski documents the approach at https://genodians.org.

Linux device driver ports - Breaking new ground

http://genodians.org/skalk/2021-04-06-dde-linux-experiments

Linux device driver ports - Generate dummy function definitions

http://genodians.org/skalk/2021-04-06-dde-linux-experiments

New articles will follow as we go.

For the development of new drivers, we will apply the new way. Still, the DDEs found in at *repos/dde_linux/* provides valuable cues.

If you start developing DDE-Linux-based drivers for Genode, please get in touch by joining Genode's mailing list. So we can incorporate your feedback into the evolving documentation and tools and provide you with assistance.

Genode Mailing List

https://genode.org/community/mailing-lists

[1] https://genode.org/documentation/developer-resources/porting_device_drivers