Run mainline Linux on $5 dollar hardware

Originalartikel

Backup

<html> <p>Today we&#8217;ll look at a fairly minimal set up that should run the latest mainline Linux. I&#8217;ve previously written about running Linux on hardware <a href=„https://popovicu.com/posts/789-kb-linux-without-mmu-riscv/“>without MMU</a>, but that exercise was done entirely virtually and more or less only verified theoretical ideas. This time we will take very cheap physical components and put them together into a reasonably powerful Linux set up (which doesn&#8217;t exclude MMU this time!). We&#8217;ll use the latest mainline kernel and deploy it on a real device.</p><h2 id=„table-of-contents“>Table of contents</h2><details><summary>Open Table of contents</summary> </details><h2 id=„background-and-motivation“>Background and motivation</h2><p>George Hilliard wrote about an amazing project that <a href=„https://www.thirtythreeforty.net/posts/2019/12/my-business-card-runs-linux/“>runs Linux on a business card</a>. I highly recommend reading that article, as it was my inspiration to dig deep through Linux on custom mini devices. There will be a lot of hardware overlap here, especially when it comes to the SoC of choice, so please take a look.</p><p>Per the article above, we&#8217;ll focus on the Allwinner F1C100S SoC. It&#8217;s very cheap and reasonably powerful, though it&#8217;s built on a slightly old ARM architecture. This SoC will bundle together the CPU core + plenty of RAM, along with the peripherals, where USB and UART are of interest here. In addition to the SoC, we only need some onboard flash storage, and we&#8217;ll focus on SPI NOR flash. Something like 8-16MB should be enough for this exercise.</p><p>Looking at the <a href=„https://jlcpcb.com/“>JLCPCB</a> website, those 2 chips can cost you less than $5, hence the title of this article. The few other components plus connectors (header pins and USB) you would need around these 2 chips plus the PCB fabrication shouldn&#8217;t add much to the price, and especially if you order in large quantities, the price can rapidly go down. The SoC can drop down all the way to around $2.</p><p>There are 2 ways to proceed with this exercise:</p><ol><li>Follow George Hilliard&#8217;s article and create a simple PCB that exposes the SoC&#8217;s UART and USB and connects it to the SPI NOR flash. This is really everything we need. You can then produce the PCB with JLCPCB and have it delivered to you within a week or two.</li><li>Alternatively, obtain a <a href=„https://linux-sunxi.org/LicheePi_Nano“>LicheePi Nano</a> board. It&#8217;s based on the same SoC, has onboard SPI NOR flash, which is all we need, but it also has a few other goodies, such as LCD connector, and comes with an old Linux set up on it. It will set you back a few more dollars, though. I bought mine for around $12 dollars.</li></ol><p><strong>Note: This won&#8217;t necessarily be a step-by-step walkthrough, and will mostly focus on concepts to bring up the software for the aforementioned hardware system. George Hilliard&#8217;s articles provides enough hardware details and I want to expand on that knowledge by providing more of software pointers/details, but only as concepts.</strong></p><h2 id=„high-level-overview“>High-level overview</h2><p><strong>Note: Before we proceed, I will assume you have a solid command of Linux fundamentals such as building your own kernel and customizing the build options. If you&#8217;d like to refresh your knowledge on this, please check out <a href=„https://popovicu.com/posts/making-a-micro-linux-distro“>this older article</a>. It&#8217;s written for RISC-V architecture, but the concepts are identical and only thing that should really be significantly different is which compiler you use to build Linux. We&#8217;re building for ARM architecture here.</strong></p><p>First, your primary website for guiding you around Linux with Allwinner SoC is likely <a href=„https://linux-sunxi.org/Main_Page“>linux-sunxi.org</a>, if you are primarily looking for documentation in English. My suspicion is that Allwinner may have fairly good

documentation in Chinese, but linux-sunxi.org was enough for me. Please do note, as the page calls out, that Allwinner does not actively support that website/community. It&#8217;s based on a lot of volunteer work from what I can see, and there are some really good efforts there, such as enabling the mainline Linux kernel to run on these SoCs (really really cool in my books), which Allwinner may not be directly involved with.</p><h3 id=„booting“>Booting</h3><p>The Allwinner SoC we&#8217;re using here is capable of firing up the bootloader in a few different ways. It could get the booloader from a memory card, or it could fall back to the SPI NOR flash onboard, as a secondary choice. In this exercise, we&#8217;ll build the latest mainline U-boot and use that as the bootloader for our Linux build.</p><p>Additionally, we&#8217;ll focus on the more modern U-boot FIT images. The devicetree will be added to the FIT image and made accessible to U-boot. More on that below.</p><h3 id=„userspace-software“>Userspace software</h3><p>To keep things simple, we&#8217;ll just run directly off from an

```
initramfs
```

image and won&#8217;t be using block devices, though I will include some notes on that below. It should be a relatively easy exercise to upgrade to using block devices, maybe even on top of onboard flash storage.</p><h2 id=„implementation“>Implementation</h2><p>To quickly summarize the above, the hardware consists of the following:</p><ol><li>Allwinner SoC connected to&#8230;</li><li>Onboard SPI NOR flash storage</li></ol><p>UART and USB of the SoC are accessible.</p><p>Software consists of:</p><ol><li>U-boot</li><li>Kernel</li><li>

```
initramfs
```

</li><li>Devicetree</li></ol><p>All software needs to be dropped into the onboard flash, as that&#8217;s the only storage we&#8217;re working with here, and that leads us into the first problem to solve.</p><h3 id=„step-1-filling-up-the-onboard-flash-using-allwinner-fel-mode“>Step 1: Filling up the onboard flash using Allwinner FEL mode</h3><p>We&#8217;re not using a memory card and we&#8217;re not exposing the SPI NOR&#8217;s pins in any way &#8212; but we have an alternative way to fill that storage. This is where Allwinner FEL mode matters.</p><p>The SoC we&#8217;re using will first try to find the bootloader on the memory card. Failing to do that, it will look for the bootloader on the SPI NOR storage. There are 2 outcomes there that you may want:</p><ol><li>Go ahead with the boot.</li><li>Don&#8217;t boot that: either you don&#8217;t want the pre-loaded Linux that comes with something like LicheePi, or you want to overwrite your own previous installation, or you want to blast the contents for whatever reason.</li></ol><p>If the SoC doesn&#8217;t find the bootloader within the onboard flash either, it will fall through to FEL mode, which is the code that lives inside the SoC&#8217;s internal ROM (baked directly into the chip, you can&#8217;t alter it). This is a very important part of our project, but let&#8217;s first talk about how we can force the chip to enter this mode.</p><p><a href=„https://linux-sunxi.org/FEL“>linux-sunxi website</a> offers an explanation on how to force the SoC into the FEL mode. I initially did this exercise on LicheePi Nano, and that one doesn&#8217;t have any buttons to force the FEL mode. Additionally, I didn&#8217;t want to use a memory card, so that wasn&#8217;t an option either. Finally, LicheePi Nano came preloaded with Linux on its onboard flash. The UART trick mentioned on that page seemed like the only option, but it wouldn&#8217;t work for me. The

```
go
```

command in the pre-deployed U-boot just wouldn&#8217;t do anything for me. The right trick came from user

```
squeuei
```

on Github, link <a href=„https://gist.github.com/squeuei/280339368e85b9faf4d756aad8171379?permalink_comment_id=3788950#gistcomment-3788950“>here</a>. The hack consists of bringing one of the SPI signals that goes out of the SoC into the onboard flash to

```
GND
```

, which effectively disables the onboard flash and leaves the SoC with no bootloader. Given that LicheePi has only one

```
GND
```

pin and I needed it for UART, I used breadboard as a helper to break out this

```
GND
```

signal into multiple pins. So my UART cable

```
GND
```

connects into the breadboard, and I also literally hold a wire poking out of the breadboard pressed onto the pin from the Github diagram as the SoC boots. I&#8217;m not kidding, I just use my thumb to press the wire &#8212; it&#8217;s not ideal, but it got the job done and grounded the SPI signal.</p><p>So quick summary: you will enter the FEL mode by leaving the SoC with no option to find any working bootloader. If the bootloader already lives on the onboard flash, as it is the case with LicheePi, take it out of the game with something like physically disabling the onboard flash for a few seconds while the SoC boots, and let it come back to life right away afterwards. If you&#8217;re designing your own PCB, maybe include a more sane way to force FEL, such as adding a FEL button.</p><p>Now, why should you care about FEL? <strong>FEL mode means that the SoC is now serving FEL protocol on the USB.</strong> You can connect your computer to the SoC (and this is why you need USB onboard, connected to the SoC) and send it commands through the FEL protocol. To keep this exercise simple, we&#8217;ll just focus on one functionality of the FEL mode &#8212; writing to the onboard SPI NOR flash. Therefore, what you get by booting into the FEL mode here is the ability to pipe arbitrary data from your computer, through USB, into the SoC and finally into the onboard flash. Those bytes can be anything.</p><p>With this in mind, we know everything we need to know in order to fill up our PCB/device with software. The only thing missing is the tool on our computer side to speak the FEL protocol with our little device. I recommend that you build <a href=„https://github.com/linux-sunxi/sunxi-tools“>sunxi-tools</a> from source, and use that. Just run the

```
sunxi-tools
```

command to get the help menu and you will find the appropriate command for asking the SoC to write to the onboard SPI flash.</p><p><strong>Note: You may encounter USB sometimes timing out. If it happens consistently, simply try to checkout a slightly older release from

```
sunxi-tools
```

Git repo and try that one. Also, if at any point you hit

```
Ctrl-C
```

and abort some FEL operation, I think it could throw the SoC off and make it freeze in FEL mode. If that happens, I recommend powering off the device and repeating the FEL operation.</strong></p><h3 id=„step-2-building-u-boot“>Step 2: Building U-boot</h3><p>This SoC seems to be decently supported by the mainline U-boot. <a href=„https://linux-sunxi.org/U-Boot“>linux-sunxi.org</a> has some information on U-boot, but I personally find it a bit outdated. I really think there&#8217;s nothing terribly specific to these SoCs that needs to be done in order to build U-boot properly. Personally, I would go ahead and just follow the <a href=„https://docs.u-boot.org/en/latest/build/gcc.html“>standard guide with GCC</a>. Find the correct

```
sunxi
```

defconfig and just build for that. We&#8217;ll soon talk about which output file we want to deploy and how.</p><h3 id=„step-3-building-kernel-and-devicetree“>Step 3: Building kernel (and devicetree)</h3><p>We&#8217;re building the mainline kernel here, and it&#8217;s pretty much the same deal as building U-boot. Check out the

```
sunxi
```

defconfig, and tweak it to whatever you need. As you build the kernel, the relevant devicetree file (DTB file) should be built as well. If you&#8217;re building your own PCB, I leave it to you to build your own devicetree, but please pay attention to the following.</p><p>

```
CONFIG_MACH_SUNIV
```

enabled me to build the devicetree file for LicheePi Nano, but I <strong>think</strong> that disabled the support for ARM EABI in the kernel build config. There was no problem simply enabling it, but I thought it may be worthwhile to explicitly call out. If you build userspace code with relatively new GCC, as I&#8217;m sure you are, you may get some weird problems if EABI is not supported. Check out this X/Twitter thread for examples of this happiness:</p><p>Alright, at this point you should have your devicetree + kernel ready to go.</p><h3 id=„step-4-building-the-initramfs-image“>Step 4: Building the

```
initramfs
```

image</h3><p>You can get as creative as you want here. Either you can bake your own

```
init
```

, or you can use Buildroot to create your image, or you can use something more exotic like U-root, as described in <a href=„https://popovicu.com/posts/making-a-micro-linux-distro“>older post</a> (I promise it&#8217;s super cool and worth your time). At this point, you should have your

```
cpio
```

archive. You can also compress it if you enable support for such compressed archives in your kernel config.</p><p>Be mindful of the resources though! We&#8217;re building for a tiny system, and so space can be an issue. Consider tradeoffs such as using

```
uClibc
```

as your C library, and so on. It could help a lot.</p><h3 id=„step-5-stitching-the-software-image-together-for-your-board“>Step 5: Stitching the software image together for your board</h3><p>This is really where the magic happens &#8212; we&#8217;re making the full software image for our device, and it will be a single binary blob that should land into the onboard flash. The SoC will get the bootloader from here and the bootloader will also extract the kernel +

```
initramfs
```

from there.</p><p>First things first, the bootloader. What you need from your U-boot build artifacts is the

```
.bin
```

file that includes SPL. The filename should obviously indicate which one that is. SPL here means U-boot is working in stages. First, the SoC will load a &#8220;light&#8221; version of U-boot, the SPL, which will then have the capability to &#8220;upgrade&#8221; to the full blown U-boot. Therefore, the stages we have here are the following:</p><ol><li>In-SoC code will locate and fetch the initial part of the bootloader, in our case, within the onboard flash. This is SPL.</li><li>SPL then leads to the full, regular U-boot.</li><li>U-boot gets the FIT image and builds from there.</li></ol><p>A nice Stack Overflow question about this is <a href=„https://stackoverflow.com/questions/31244862/what-is-the-use-of-spl-secondary-program-loader“>here</a>.</p><p>And now about the FIT image. Think of the FIT image as an archive that packs together the kernel,

```
initramfs
```

and whatever other bits and blobs you may want to pack inside. The FIT image is built with

```
mkimage
```

tool. When you build your U-boot, there will be a

```
mkimage
```

binary as well inside the

```
tools
```

directory. That&#8217;s the one you need. The concept is that you define your FIT image with a single text file and simply point

```
mkimage
```

to that &#8212; your output witll be a FIT image file. The format for this text file is interestingly a devicetree. U-boot documentation provides some rationale on why they opted for that format and I leave it to the reader to form an opinion about that. My

```
image.its
```

file is below, as an example:</p><pre>/dts-v1/;/ { description = „LicheePi Nano FIT Image“; #address-cells = &lt;1&gt;; images { script { description = „Boot script“; data = /incbin/(„script.txt“); type = „script“; compression = „none“; }; kernel { description = „Kernel“; data = /incbin/(„/home/uros/path/to/linux/kernel/linux-6.8.2/arch/arm/boot/zImage“); type = „kernel“; arch = „arm“; os = „linux“; compression = „none“; load = &lt;0x80008000&gt;; entry = &lt;0x80008000&gt;; hash { algo = „sha1“; }; }; fdt { description = „DTB“; data = /incbin/(„/home/uros/path/to/linux/kernel/linux-6.8.2/arch/arm/boot/dts/allwinner/suniv-f1c100s-licheepi-nano.dtb“); type = „flat_dt“; arch = „arm“; compression = „none“; load = &lt;0x80C00000&gt;; entry = &lt;0x80C00000&gt;; hash { algo = „sha1“; }; }; initrd { description = „Initrd“; data = /incbin/(„/home/uros/path/to/lichee/initramfs.cpio“); type = „ramdisk“; arch = „arm“; os = „linux“; compression = „none“; hash { algo = „sha1“; }; }; }; configurations { default = „standard“; standard { description = „Standard Boot“; kernel = „kernel“; fdt = „fdt“; ramdisk = „initrd“; hash { algo = „sha1“; }; }; };};</pre><p>I recommend <a href=„https://www.thegoodpenguin.co.uk/blog/u-boot-fit-image-overview/“>this article</a> to learn about the FIT images. After reading that article, you should understand the snippet above without any issues.</p><p><em>Bonus info: you can run U-boot scripts from FIT images trivially too, and this could be your first U-boot experiment with your device. I will call that out in the section below where we run the whole software stack.</em></p><p>At this point, we have a bootloader (built with SPL), and a FIT image that wraps the kernel,

```
initramfs
```

and the devicetree. How do these 2 play together, though? You can simply

```
cat
```

them together into one image and that&#8217;s what goes into your onboard flash. The U-boot with SPL image goes first, as the bootloader should be at the very beginning of the flash storage. This is where the SoC will peek to find the bootloader to use.</p><p>If we refer back to the beginning of the implementation section, we see that we have all the binaries we need to run the system, so let&#8217;s do that!</p><h2 id=„running-the-system“>Running the system</h2><p>Let&#8217;s

```
cat
```

the bootloader and the FIT image into one blob. I assume you have built your FIT image as an

```
itb
```

file, using

```
mkimage
```

.</p><pre>cat /path/to/u-boot-sunxi-with-spl.bin image.itb &gt; lichee.img</pre><p>Next, I&#8217;ll demonstrate the stack on LicheePi Nano &#8212; I&#8217;ll use the wire to ground the SPI pin and disable the onboard flash for a bit. Additionally, I will both power the device and fill the onboard flash via my computer&#8217;s USB. Finally, I have a USB-to-TTL cable that connects my SoC&#8217;s UART to my computer.</p><p>Now, since I&#8217;m forcing the SoC into the FEL mode, there should be no output whatsoever to UART. FEL mode won&#8217;t send anything there. A quick way to verify that the SoC is listening in FEL mode, though, would be to simply run

```
lsusb
```

tool or something similar that will scan what USB devices are visible. I see the following:</p><pre>Bus 001 Device 009: ID 1f3a:efe8 Allwinner Technology sunxi SoC OTG connector in FEL/flashing mode</pre><p>Great, so my SoC is in the FEL mode, and the onboard flash can be written.</p><pre>./sunxi-fel -v -p spiflash-write 0 /home/uros/path/to/lichee/lichee.img</pre><p>The

```
-p
```

flag will give you a nice little progress bar that also shows the writing speed. I get around 110 kB/s.</p><p>Cool, everything is written. I can now either power down and power up the board again, or I can issue a FEL command for a reset. I&#8217;ll do it the FEL way.</p><pre>./sunxi-fel wdreset</pre><p>I forgot to mention that you may need to run

```
sunxi-fel
```

with

```
sudo
```

because you&#8217;re poking with the hardware on the USB port.</p><p>Now I&#8217;ll pay attention to UART. I won&#8217;t be forcing the device into FEL mode now, so I should see the proper boot up on UART. And that&#8217;s exactly what happens, I can see U-boot doing its magic.</p><pre>U-Boot SPL 2024.01 (Mar 29 2024 - 11:06:03 -0700)DRAM: 32 MiBTrying to boot from sunxi SPIU-Boot 2024.01 (Mar 29 2024 - 11:06:03 -0700) Allwinner TechnologyCPU: Allwinner F Series (SUNIV)Model: Lichee Pi NanoDRAM: 32 MiBCore: 30 devices, 18 uclasses, devicetree: separateWDT: Not starting watchdog@1c20ca0MMC: mmc@1c0f000: 0Loading Environment from FAT... Card did not respond to voltage select! : -110 **Bad device specification mmc 0** In: serial@1c25000Out: serial@1c25000Err: serial@1c25000Net: No ethernet found.Hit any key to stop autoboot: 0Card did not respond to voltage select! : -110No ethernet found.missing environment variable: pxeuuidRetrieving file: pxelinux.cfg/00000000No ethernet found.Retrieving file: pxelinux.cfg/0000000No ethernet found.Retrieving file: pxelinux.cfg/000000No ethernet found.Retrieving file: pxelinux.cfg/00000No ethernet found.Retrieving file: pxelinux.cfg/0000No ethernet found.Retrieving file: pxelinux.cfg/000No ethernet found.Retrieving file: pxelinux.cfg/00No ethernet found.Retrieving file: pxelinux.cfg/0No ethernet found.Retrieving file: pxelinux.cfg/default-arm-sunxi-sunxiNo ethernet found.Retrieving file: pxelinux.cfg/default-arm-sunxiNo ethernet found.Retrieving file: pxelinux.cfg/default-armNo ethernet found.Retrieving file: pxelinux.cfg/defaultNo ethernet found.Config file not foundNo ethernet found.=&gt;</pre><p>Now, U-boot doesn&#8217;t know by default where to boot from and instead, it interactively waits in its shell for commands. It did something by default, like try to load from a memory card, or fall back to Ethernet, but none of that worked. That&#8217;s why it&#8217;s waiting for explicit commands.</p><p>First, let&#8217;s see how it sees the board by running

```
bdinfo
```

.</p><pre>=&gt; bdinfoboot_params = 0x80000100DRAM bank = 0x00000000-&gt; start = 0x80000000-&gt; size = 0x02000000flashstart = 0x00000000flashsize = 0x00000000flashoffset = 0x00000000baudrate = 115200 bpsrelocaddr = 0x81f99000reloc off = 0x00899000Build = 32-bitcurrent eth = unknowneth-1addr = (not set)IP addr = &lt;NULL&gt;fdt_blob = 0x81e76da0new_fdt = 0x81e76da0fdt_size = 0x00002120lmb_dump_all: memory.cnt = 0x1 / max = 0x10 memory[0] [0x80000000-0x81ffffff], 0x02000000 bytes flags: 0 reserved.cnt = 0x1 / max = 0x10 reserved[0] [0x81e72a20-0x81ffffff], 0x0018d5e0 bytes flags: 0devicetree = separateserial addr = 0x01c25000 width = 0x00000004 shift = 0x00000002 offset = 0x00000000 clock = 0x05f5e100arch_number = 0x00000000TLB addr = 0x81ff0000irq_sp = 0x81e76d90sp start = 0x81e76d80Early malloc usage: 300 / 2000</pre><p>This tells us something about the board and hopefully clears up some mystery on why the loading addresses in the FIT image script were used the way they were. Well, to be perfectly honest, the addresses could be different, obviously, but I lifted the addresses from the factory LicheePi Nano image.</p><p>OK, now let&#8217;s load that FIT image from the onboard flash into the SoC&#8217;s RAM. We&#8217;ll use U-boot&#8217;s

```
sf
```

subsystem for that. You have to initialize the subsytem by probing the flash devices. This is done by running</p><pre>sf probe</pre><p>and here is my output:</p><pre>=&gt; sf probeSF: Detected xt25f128 with page size 256 Bytes, erase size 4 KiB, total 16 MiB</pre><p>If your U-boot doesn&#8217;t support it, it probably means you didn&#8217;t enable the support for it when building U-boot, so you would want to correct that if this is the case. How do we load the FIT image, though? Well, our onboard flash consists of 2 blobs spliced together: the bootloader and the FIT image. Therefore, the offset where we start reading the FIT image is equal to size of the U-boot image, since it comes first. And the length of the blob is something we can check on our host computer easily as well. In my case, the command to read from the onboard flash into RAM looks like this (all numbers hex):</p><pre>=&gt; sf read 0x81000000 0x60E44 0x809C38device 0 offset 0x60e44, size 0x809c38SF: 8428600 bytes @ 0x60e44 Read: OK</pre><p>This takes a while. I didn&#8217;t put too much work into compressing the kernel, cutting out unnecessary components in the kernel and so on, so I probably have an overweight image that I could otherwise reduce. However, this is an exercise, not a production build, so I can get away with it. The only thing that remains now is to boot the FIT image now that it&#8217;s in memory.</p><pre>=&gt; bootm 0x81000000#standard</pre><p>

```
#standard
```

is the name of my FIT image configuration, and the hex number is the address where FIT was loaded. One of the selling points for FIT images is that you can easily have different configurations and thus have the same image you can deploy on different devices, but loaded differently.</p><p>I now see the following:</p><pre>=&gt; bootm 0x81000000#standard## Loading kernel from FIT Image at 81000000 ... Using 'standard' configuration Trying 'kernel' kernel subimage Description: Kernel Type: Kernel Image Compression: uncompressed Data Start: 0x81000124 Data Size: 3259824 Bytes = 3.1 MiB Architecture: ARM OS: Linux Load Address: 0x80008000 Entry Point: 0x80008000 Hash algo: sha1

```
Hash value: 09cf68b7f88f107a5f36135929f05c9bc8838650 Verifying Hash Integrity ... sha1+ OK##
Loading ramdisk from FIT Image at 81000000 ... Using 'standard' configuration Trying 'initrd' ramdisk
subimage Description: Initrd Type: RAMDisk Image Compression: uncompressed Data Start:
0x8131dae8 Data Size: 5159936 Bytes = 4.9 MiB Architecture: ARM OS: Linux Load Address:
unavailable Entry Point: unavailable Hash algo: sha1 Hash value:
b719e464ef3f368c0c1c19a539e2c34d1b20f8ed Verifying Hash Integrity ... sha1+ OK## Loading fdt
from FIT Image at 81000000 ... Using 'standard' configuration Trying 'fdt' fdt subimage Description:
DTB Type: Flat Device Tree Compression: uncompressed Data Start: 0x8131bfac Data Size: 6767
Bytes = 6.6 KiB Architecture: ARM Load Address: 0x80c00000 Hash algo: sha1 Hash value:
b4e19d4011b7af0cdffd7cebe254d85e272dc6b5 Verifying Hash Integrity ... sha1+ OK Loading fdt
from 0x8131bfac to 0x80c00000 Booting using the fdt blob at 0x80c00000Working FDT set to
80c00000 Loading Kernel Image Loading Ramdisk to 81214000, end 816ffc00 ... OK Loading Device
Tree to 8120f000, end 81213a6e ... OKWorking FDT set to 8120f000Starting kernel ...[ 0.000000]
Booting Linux on physical CPU 0x0[ 0.000000] Linux version 6.8.2 (uros@debian-home) (arm-linux-
gnueabihf-gcc (Debian 12.2.0-14) 12.2.0, GNU ld (GNU Binutils for Debian) 2.40) #5 Wed Apr 3
17:24:53 PDT 2024[ 0.000000] CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=0005317f[
0.000000] CPU: VIVT data cache, VIVT instruction cache[ 0.000000] OF: fdt: Machine model: Lichee Pi
Nano[ 0.000000] Memory policy: Data cache writeback[ 0.000000] cma: Failed to reserve 16 MiB on
node -1[ 0.000000] Zone ranges:[ 0.000000] Normal [mem
0x0000000080000000-0x0000000081ffffff][ 0.000000] HighMem empty[ 0.000000] Movable zone
start for each node[ 0.000000] Early memory node ranges[ 0.000000] node 0: [mem
0x0000000080000000-0x0000000081ffffff][ 0.000000] Initmem setup node 0 [mem
0x0000000080000000-0x0000000081ffffff][ 0.000000] Kernel command line:[ 0.000000] Dentry
cache hash table entries: 4096 (order: 2, 16384 bytes, linear)[ 0.000000] Inode-cache hash table
entries: 2048 (order: 1, 8192 bytes, linear)[ 0.000000] Built 1 zonelists, mobility grouping on. Total
pages: 8128[ 0.000000] mem auto-init: stack:all(zero), heap alloc:off, heap free:off[ 0.000000]
Memory: 18180K/32768K available (5120K kernel code, 732K rwdata, 1288K rodata, 1024K init, 231K
bss, 14588K reserved, 0K cma-reserved, 0K highmem)[ 0.000000] SLUB: HWalign=32, Order=0-3,
MinObjects=0, CPUs=1, Nodes=1[ 0.000000] workqueue: name exceeds WQ_NAME_LEN. Truncating
to: events_freezable_power_efficien[ 0.000000] NR_IRQS: 16, nr_irqs: 16, preallocated irqs: 16[
0.000010] sched_clock: 32 bits at 24MHz, resolution 41ns, wraps every 89478484971ns[ 0.000140]
clocksource: timer: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 79635851949 ns[ 0.000928]
Console: colour dummy device 80×30[ 0.000999] printk: legacy console [tty0] enabled[ 0.002092]
Calibrating delay loop... 203.16 BogoMIPS (lpj=1015808)[ 0.070368] CPU: Testing write buffer
coherency: ok[ 0.070760] pid_max: default: 32768 minimum: 301[ 0.071177] Mount-cache hash table
entries: 1024 (order: 0, 4096 bytes, linear)[ 0.071355] Mountpoint-cache hash table entries: 1024
(order: 0, 4096 bytes, linear)[ 0.077955] Setting up static identity map for 0x80100000 -
0x80100058[ 0.080594] devtmpfs: initialized[ 0.085527] VFP support v0.3: not present[ 0.086332]
clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 19112604462750000 ns[
0.086557] futex hash table entries: 256 (order: -1, 3072 bytes, linear)[ 0.086953] pinctrl core:
initialized pinctrl subsystem[ 0.089600] DMA: preallocated 256 KiB pool for atomic coherent
allocations[ 0.092641] thermal_sys: Registered thermal governor 'step_wise'[ 0.113107] SCSI
subsystem initialized[ 0.115956] usbcore: registered new interface driver usbfs[ 0.116247] usbcore:
registered new interface driver hub[ 0.116511] usbcore: registered new device driver usb[ 0.117351]
pps_core: LinuxPPS API ver. 1 registered[ 0.117477] pps_core: Software ver. 5.3.6 - Copyright
2005-2007 Rodolfo Giometti &lt;giometti@linux.it&gt;[ 0.120526] clocksource: Switched to
clocksource timer[ 0.173431] Unpacking initramfs...[ 0.191146] workingset: timestamp_bits=30
max_order=13 bucket_order=0[ 0.192883] Block layer SCSI generic (bsg) driver version 0.4 loaded
(major 249)[ 0.193079] io scheduler mq-deadline registered[ 0.193163] io scheduler kyber registered[
0.193407] io scheduler bfq registered[ 0.532161] Freeing initrd memory: 5040K[ 0.688982] Serial:
8250/16550 driver, 8 ports, IRQ sharing disabled[ 0.708330] i2c_dev: i2c /dev entries driver[
```

0.712824] sunxi-wdt 1c20ca0.watchdog: Watchdog enabled (timeout=16 sec, nowayout=0)[0.717521] usbcore: registered new interface driver usbhid[0.717688] usbhid: USB HID core driver[0.758847] gpio gpiochip0: Static allocation of GPIO base is deprecated, use dynamic allocation.[0.775519] suniv-f1c100s-pinctrl 1c20800.pinctrl: initialized sunXi PIO driver[0.777459] suniv-f1c100s-pinctrl 1c20800.pinctrl: supply vcc-pe not found, using dummy regulator[0.802857] 1c25000.serial: ttyS0 at MMIO 0x1c25000 (irq = 116, base_baud = 6250000) is a 16550A[0.803172] printk: legacy console [ttyS0] enabled[1.195352] suniv-f1c100s-pinctrl 1c20800.pinctrl: supply vcc-pc not found, using dummy regulator[1.205561] sun6i-spi 1c05000.spi: Failed to request TX DMA channel[1.212148] sun6i-spi 1c05000.spi: Failed to request RX DMA channel[1.221752] spi-nor spi0.0: found spi-nor-generic, expected w25q128[1.257265] usb_phy_generic usb_phy_generic.0.auto: dummy supplies not allowed for exclusive requests[1.266831] usb_phy_generic usb_phy_generic.0.auto: dummy supplies not allowed for exclusive requests[1.277727] musb-hdrc musb-hdrc.1.auto: MUSB HDRC host driver[1.283909] musb-hdrc musb-hdrc.1.auto: new USB bus registered, assigned bus number 1[1.293751] suniv-f1c100s-pinctrl 1c20800.pinctrl: supply vcc-pf not found, using dummy regulator[1.309673] hub 1-0:1.0: USB hub found[1.314855] hub 1-0:1.0: 1 port detected[1.322721] clk: Disabling unused clocks[1.330731] sunxi-mmc 1c0f000.mmc: initialized, max. request size: 16384 KB[1.342892] Freeing unused kernel image (initmem) memory: 1024K[1.349021] Run /init as init processSaving 256 bits of non-creditable seed for next bootStarting syslogd: OKStarting klogd: OKRunning sysctl: OKStarting network: ip: socket: Function not implementedip: socket: Function not implementedFAILWelcome to Buildrootbuildroot login:</pre><p>I&#8217;m in my Linux system! I can move around and do my Linux business as usual. As you can see, I&#8217;m running both my U-boot and Linux at the latest versions.</p><p>One more thing about booting: we had to interactively boot the kernel with all these commands above, and that is not ideal. Keep in mind that when you build U-boot, you can bake your own automatic command to be executed. And that command can be something concise, you could, as hinted above, simply jump into a U-boot script that&#8217;s within the FIT image. That script can be generated as a part of your build flow and it can do whatever you want it to do &#8212; your creativity is the limit here.</p><h2 id=„bonus-section-onboard-flash-as-a-block-device-and-u-boot-devicetree-injections“>Bonus section: onboard flash as a block device and U-boot devicetree injections</h2><p>When I booted my LicheePi Nano just as I received it, I noticed it sees some block devices. Those block devices were formed on top of the MTD system that works with the onboard flash. Clearly, since there was one flash chip, it had to be partitioned in order for me to see the block devices. However, after Googling a bit, I learned it&#8217;s not common for the flash devices to have the partition tables. Then how does the kernel know about the partitions?</p><p>One way is to literally pass the onboard flash partition set up as kernel command line. It&#8217;s rather verbose, but easy to use.</p><p>Another, very interesting way, is to select the option in the kernel build options which enables the kernel to parse the partitions out of the devicetree. I think this is a very interesting concept, but I also think it&#8217;s a bit uncomfortable to use &#8212; do I have to compile a custom device tree just to get some partitions?</p><p>Not really, U-boot has something to do here. If you look at <a href=„https://community.toradex.com/t/how-is-mtdparts-passed-from-u-boot-to-the-kernel/10130“>this interesting thread</a> you can see that U-boot indeed has mechanisms to &#8220;fixup&#8221; the devicetree and inject something into it. One of the things you could inject is the flash partitioning.</p><p>I haven&#8217;t tried this, as I kept it simple and didn&#8217;t deal with block devices in this exercise, but I thought it was a fascinating concept at least.</p><p>Since you probably want to stay very lightweight in your images here, consider reading these articles I have written before to get some ideas about building lightweight

```
initramfs
```

images:</p><ul><li><a href=„https://popovicu.com/posts/building-multiplatform-linux-initramfs-with-one-command-in-bazel“>Building multiplatform Linux initramfs with one command in Bazel</a></li><li><a href=„https://popovicu.com/posts/cross-compiling-c-with-bazel“>Cross compiling C and C++ with Bazel</a></li></ul><p>Definitely not required in order to have some fun with this exercise, but I think it&#8217;s worth your time.</p><h2 id=„conclusion“>Conclusion</h2><p>If you are curious, like I am, about how much can we minimize our devices but still stay able to run a full blown Linux, I think you have your answer. With just 2 very cheap chips, we managed to get the full computing system. Building your own tiny PCB with these concepts should be fairly easy knowing this, especially if you follow George Hilliard&#8217;s articles.</p><p>While I do think Allwinner could do a better job at providing easy-to-digest English documentation on using their chips, and I would hope they do some serious work about mainline Linux support, and supporting the open-source projects that provide the Allwinner tooling &#8212; I think it&#8217;s still absolutely stunningly amazing that they provide Linux-ready SoCs at those low prices. I&#8217;m the first person to complain about cruft and jankiness in tech, but even I will say that the journey of studying this SoC and an existing design of LicheePi Nano wasn&#8217;t that painful, considering the price tags.</p><p>Another amazing thing nowadays is how easy it is to order your own professional PCBs. I recently ordered a practice PCB from <a href=„https://jlcpcb.com/“>JLCPCB</a>, and they got it done + delivered to the US within days. The prices were more than fair, and I will definitely be playing with various custom Linux boards from this point onwards.</p><p>I hope you had some fun with this, and I&#8217;ll see you in the next one.</p><p>Enjoy hacking! Please consider following on <a href=„https://twitter.com/popovicu94“>Twitter/X</a> and <a href=„https://www.linkedin.com/in/upopovic/“>LinkedIn</a> to stay updated.</p> </html>