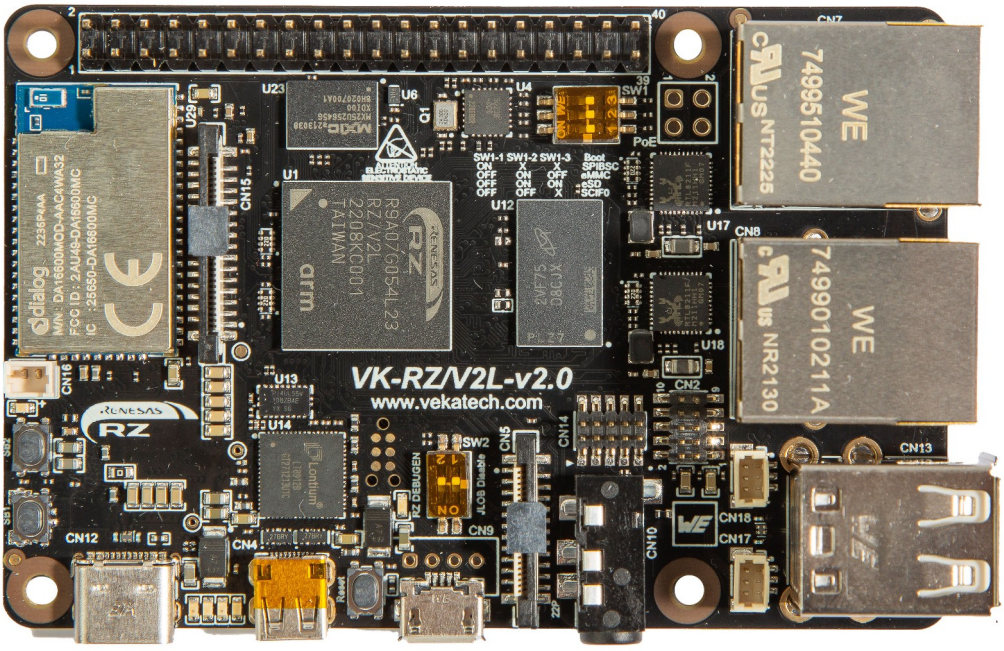
VK-RZ/V2L How To



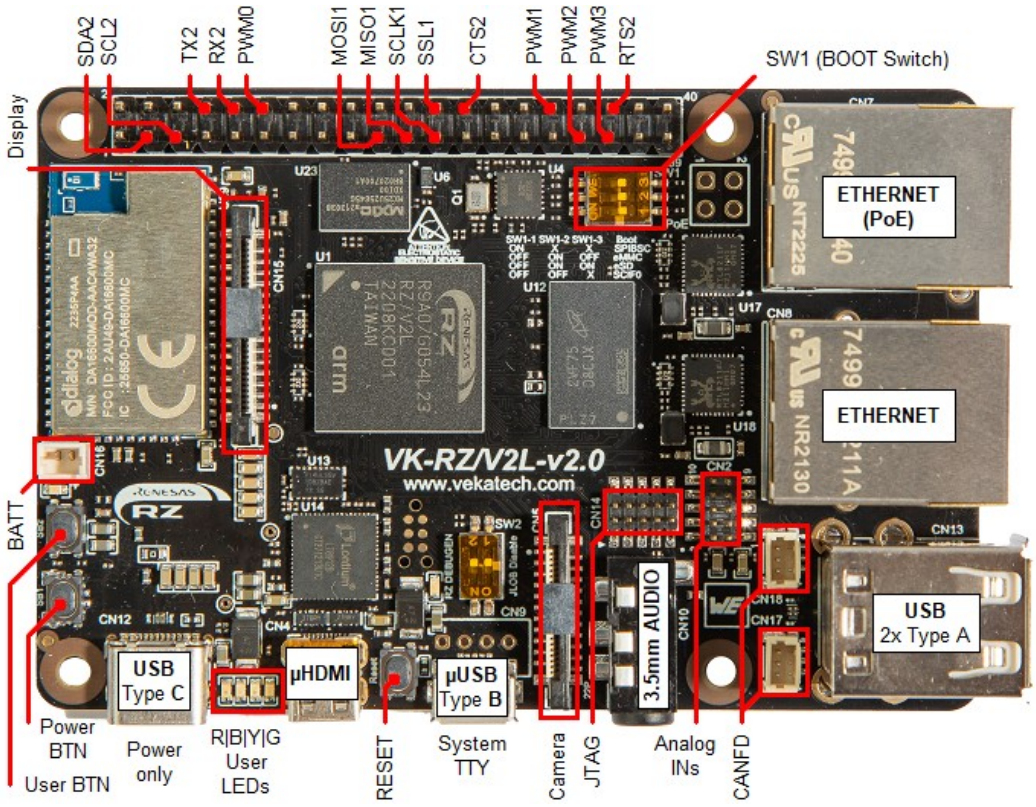
VK-RZ/V2L v2.0 Board

Content:

1. Introduction

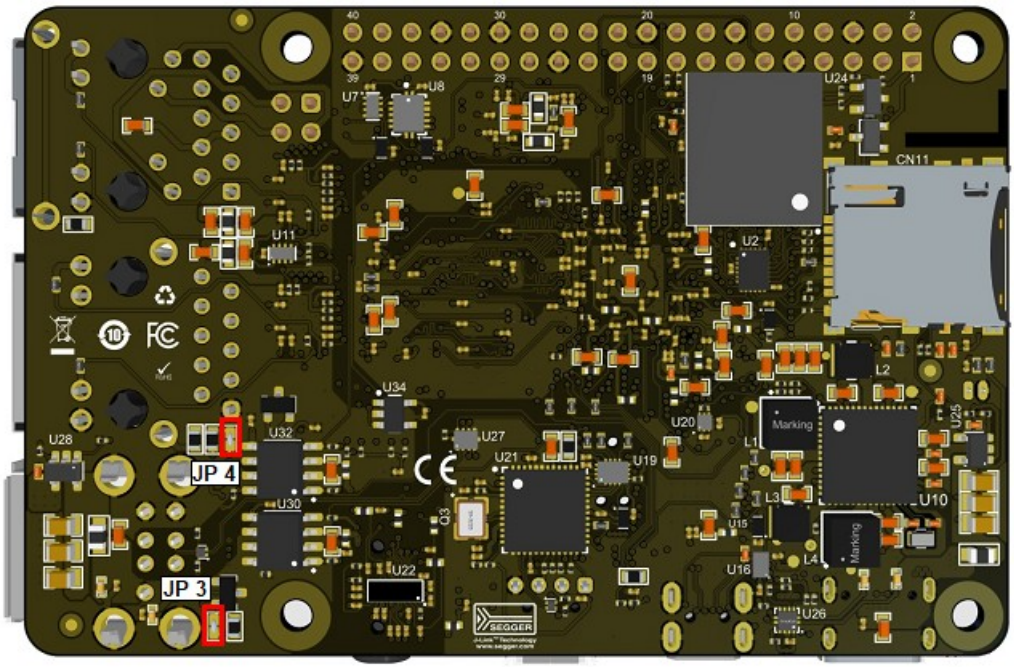
VK-RZ/V2L is industrial oriented board, compatible with Raspberry Pi 4 shields. It is based on Renesas **R9A07G054L23GBG**, **Dual ARM Cortex-A55 + Cortex-M33 MCU**. The main purpose of this manual is to show how to get started with the board.

1.1 Connectors



Connectors & Signals

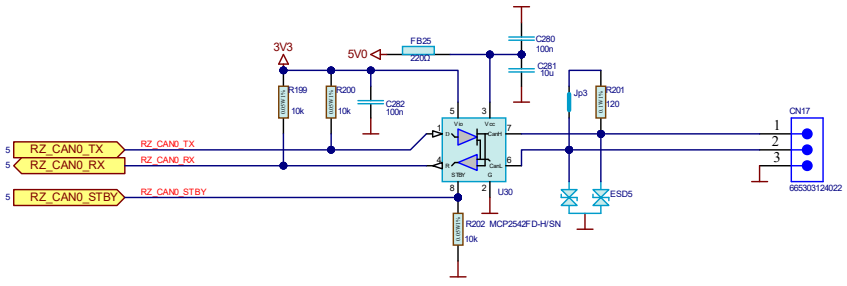
1.2 Signals



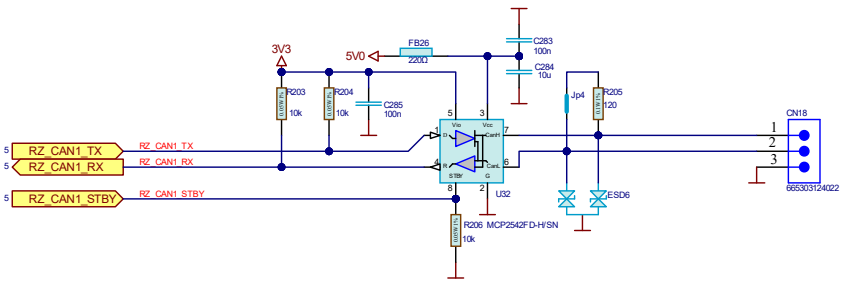
Joint Solderpads

Some of the pads below can be changed to suit your needs:

- **JP 3** (open) → Connects **120Ω termination** resistor between CAN0's **H** & **L** lines.
- **JP 4** (open) → Connects **120Ω termination** resistor between CAN1's **H** & **L** lines.



JP3 & JP6



1.3 3D view

There is available step model of the board. It is convenient if you want to make a case or embed it into a bigger device, so mechanical data can be exported in great detail.

1.4 Dimensions

Board size & Mounting Holes

2. Power Up

The board can be powered with **5 V** from 4 sources:

- USB Type A (Top)
- USB Type B micro
- USB Type C (Switchable: Power only / Power only + USB v2.0)
- 40pin Extension header

In idle state (without plugged periphery) the board consumes ~ **450 mA**. The consumption however, can jump up to ~ **1,2 A** (if Camera, Display, Ethernet, USB Flash, USB Keyboard, Audio jack & SD card are plugged in).

3. Install U-boot (v2021.10)

You can get the U-boot firmware from our site or compile it yourself from our repository, if you prefer manually building it from scratch. There is guidance how you can do it, but let's stick to the precompiled binaries for now:

- Connect **VK-RZ/V2L** to the PC (through **USB Type B micro**) & see what COM port is assigned by the OS in the Device Manager.
- Unzip **vkrzv2l-program-utility.zip**.
- Edit **cfgcom_vkrzv2l.ini** file, altering the **COM** number to match with **VK-RZ/V2L**.
- Set the **SW1** on **VK-RZ/V2L** (**1:OFF | 2:OFF |3:OFF**) so it can boot from **SCIF0**.

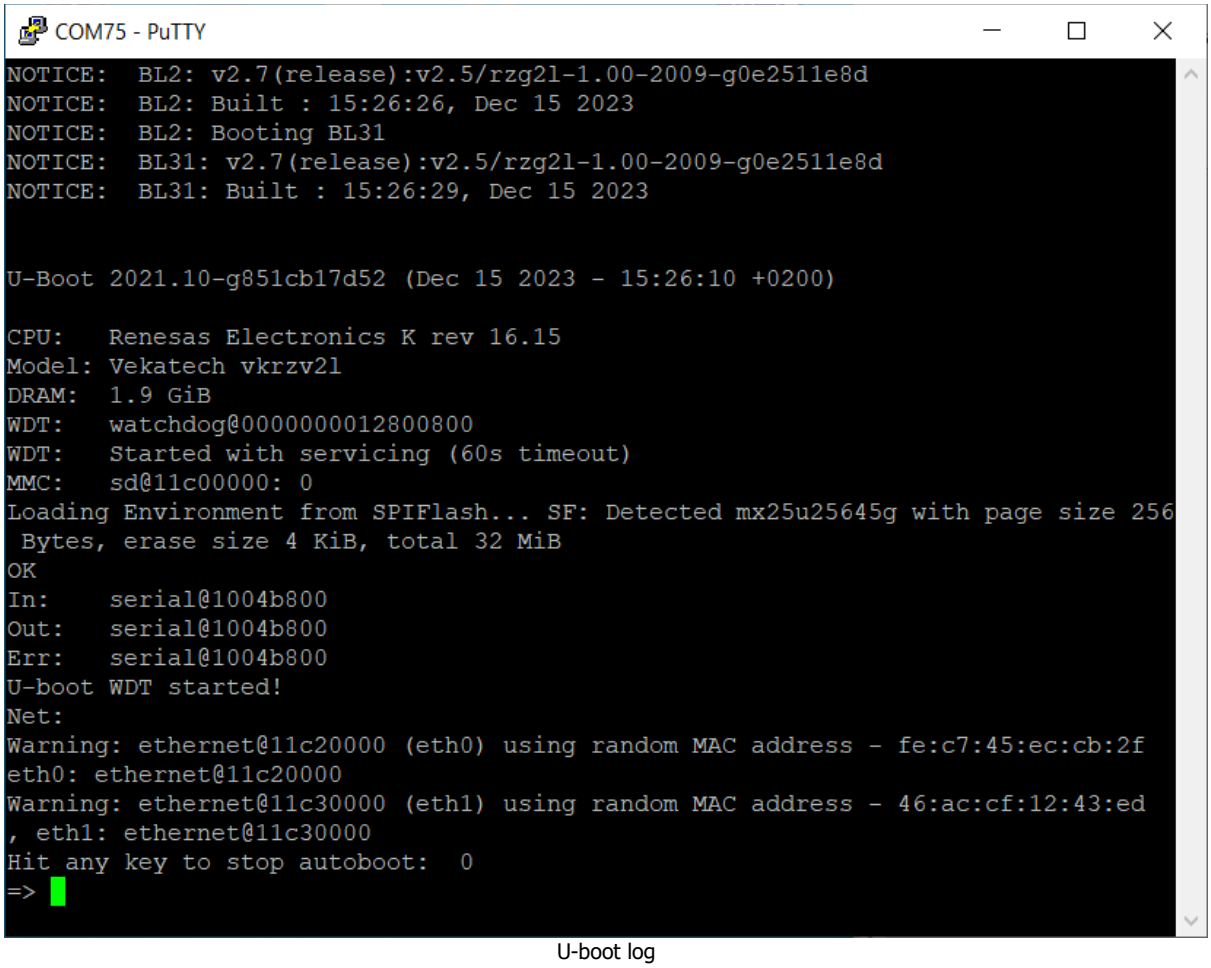
3.1 into SPI Flash

- Execute **vkrzv2l_bootloader-sf.bat**.
- Press **reset** button on the **VK-RZ/V2L**

3.2 into eMMC SSD

- Execute **vkrzv2l_bootloader-emmc.bat**.
- Press **reset** button on the **VK-RZ/V2L**

After download is ready, set **SW1** back to boot from **SPIBSC** (1:ON | 2:X | 3:X) in case you executed 3.1 or **eMMC** (1:OFF | 2:ON | 3:OFF) in case of 3.2. Open VK-RZ/V2L's COM port on (115200|8|N|1) and press **reset** => you should now be able to see the boot log of the U-boot.



U-boot log

4. Boot logic

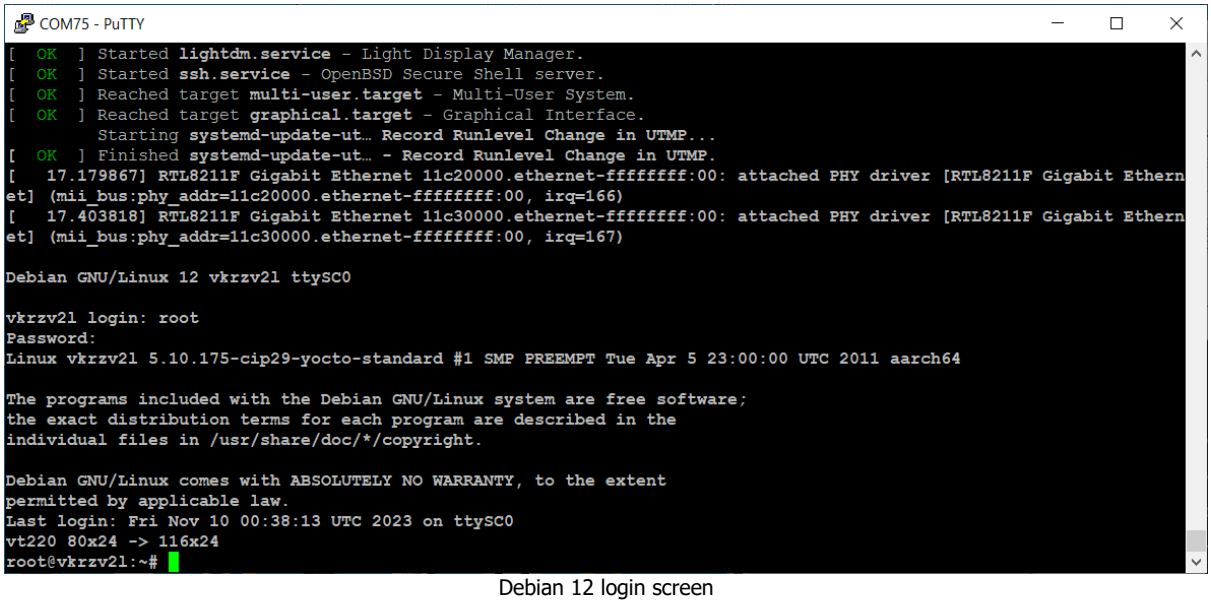
Understanding boot logic is crucial. As you see there are **2** copies of **U-boot** (1 in eMMC & 1 in SPI). Environment parameters of the ones are slightly different from the others. They are configured in such way, that when **SW1** is set to boot from eMMC (**1:OFF | 2:ON |3:OFF**) it's U-boot searches Linux image in the same that **eMMC**, but when SW1 is set to boot from SPI (**1:ON | 2:X | 3:ON**) it's U-boot searches Linux image in **µSD** card. That's why default U-boot environment parameters differs on purpose and you don't need to edit them every time you want to boot from one or other media (you just set SW1). That actually explains why booting from SPI, can leads to booting from µSD (if SW1:3 is ON), & booting from eMMC (if SW1:3 is OFF). Image on the eMMC can be booted from both (SPI & eMMC), but image on the µSD card - only from SPI.

5. Install Linux (Kernel v5.10.175)

Now that you have 2 workable U-boot instances, up & running, it is time to load something bigger. You can get Debian Linux image from our site or compile it yourself from our repository, (if you prefer building it from scratch). The guidance how you can build the image is in **readme** of the repository, but let's stick to the precompiled binaries for now:

5.1 into SD card → Debian v12.4 (Bookworm)

- Get a **µSD** card with **min 4 G** capacity and plug it into the PC.
- Download tool, named Rufus from here.
- Unzip **debian-bookworm-vkrzv2l.img.xz**.
- Launch Rufus, for **Device** select µSD card drive, for **boot section** select desired Linux image file (in this case **debian-bookworm-vkrzv2l.img**).
- Hit **START** & wait completion (Status **READY**), eject µSD card & insert it again.
- Open the **boot** partition and edit **uEnv.txt** file, so **sdhi** overlay to be added i.e. make sure `fdt_extra_overlays=vkrz-sdhi0-sdhi.dtbo` … is typed out (applyed).
- Plug µSD card into VK-RZ/V2L's holder, set **SW1** to (**1:ON | 2:X | 3:ON**) & press reset.
- You should now be able to see login screen, use user: `root` & password: `vkrzv2l`.



Debian 12 login screen

- You can check for available package updates: `sudo apt-get update`.

If there are available upgrades such as in this case:

```
34 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

- You can install them: `sudo apt-get upgrade`.
- If you want to extend the root partition to use the full capacity of the µSD:

`sudo growpart /dev/mmcblk0 2 && sudo resize2fs /dev/mmcblk0p2`.

- If you want to extend the root partition to a fixed size and form a new separate partition you should first install partition editor `sudo apt-get install parted`.

Extend the root partition to **N** GB size: `sudo parted /dev/mmcblk0 resizepart 2 NG`.

Fix filesystem block size: `sudo resize2fs /dev/mmcblk0p2`.

Make new partition: `sudo parted /dev/mmcblk0 "mkpart primary fat32 NG -1"`.

Format the new partition: `sudo mkfs.vfat -F 32 /dev/mmcblk0p3`.

To verify how the µSD card look like, type that: `sudo parted /dev/mmcblk0 print`.

5.2 into eMMC SSD → Yocto v3.1.26 (Dunfell)

First you have to prepare a **boot** server to boot from. For this purpose, let's setup such a server, from which we can boot Debian. Once booted, it will help us to place Yocto image into the **eMMC**. In other words, 2 services are needed to work on that server: **TFTP** and **NFS**.

- Make sure the server has all the software needed for network boot:

`sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat cpio python python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit mesa-common-dev p7zip-full tftp tftpd-hpa nfs-common nfs-kernel-server`.

- Create directory for the **TFTP** server: `sudo mkdir -p /tftpboot/vkrzv2l`.
- Change the TFTP server configuration by editing the `/etc/default/tftpd-hpa` file:

make sure `TFTP_DIRECTORY="/tftpboot"` path is specified instead TFTP's default one.

- Activate the TFTP server: `sudo systemctl enable tftpd-hpa`.
- Start the TFTP server: `sudo systemctl restart tftp-hpa`.
- Test the TFTP server, by putting a text file in it & checking if the file can be downloaded:

`sudo chmod 777 /tftpboot` && `sudo echo "Hello World" > /tftpboot/hello.txt`.

Enter in `tftp>` mode with: `tftp localhost` and get the file: `get hello.txt`.

The command should return no errors.

- Create directory for the **NFS** server: `sudo mkdir -p /nfs/vkrzv2l`.
- Start the NFS server: `sudo /etc/init.d/nfs-kernel-server start`.
- Change the NFS server configuration by editing the `/etc/exports` file, add the following line at the end: `/nfs/vkrzv2l *(rw,no_subtree_check,sync,no_root_squash)`.
- Refresh the NFS server: `sudo exportfs -a`.
- Test the NFS server: `showmount -e localhost`.
  You should get: `/nfs/vkrzv2l *`.

- Make sure the machine that runs those 2 services, gets a static IP. The easiest way is to configure the DHCP of the LAN (in which the boot server participates) to do that for you.

- Get the Debian image on the boot server machine (it is the same image from 5.1).
- Unzip it: `xz -d debian-bookworm-vkrzv2l.img.xz`.
- Get a loop dev**N**: `sudo losetup -f -P --show debian-bookworm-vkrzv2l.img`.
- Mount the boot partition: `sudo mount /dev/loopNp1 /mnt`.
- Copy it: `sudo cp -r /mnt/* /tftpboot/vkrzv2l`.
- Edit **uEnv.txt** file in `/tftpboot/vkrzv2l/uEnv.txt`, so **emmc** overlay to be added i.e. make sure `fdt_extra_overlays=vkrz-sdhi0-emmc.dtbo`... is typed out & save.
- Mount the root partition: `sudo umount /mnt` && `sudo mount /dev/loopNp2 /mnt`.
- Copy it: `sudo cp -r /mnt/* /nfs/vkrzv2l`.
- Edit **fstab** file in `/nfs/vkrzv2l/etc/fstab`, so the right setting for the root partition to be applied i.e. make sure to delete the second row and edit the first row to look like this: `/dev/root/<TAB>/<TAB>nfs<TAB>rw,tcp,nolock<TAB>0<TAB>1` & save it.
- Release loop dev**N**: `sudo umount /mnt` && `sudo losetup -d /dev/loopNp2`.
- Download, unzip the Linux image (in this case Yocto) & place it in `nfs/vkrzv2l/home`.
  `sudo bzip2 -d /nfs/vkrzv2l/home/core-image-bsp-vkrzv2l.wic.bz2`.

- Boot from the boot server, setting **SW1** to **(1:ON | 2:X | 3:OFF)** & press reset.
- Login to Debian with user: **root** & password: **vkrzv2l**.
- Go to `/home`, and write it into the **eMMC**:
  `dd of=/dev/mmcblk0 if=core-image-bsp-vkrzv2l.wic bs=1M iflag=fullblock oflag=direct conv=fsync status=progress`.
- If you want to extend the root partition to use the full capacity of the eMMC:
`growpart /dev/mmcblk0 2 && resize2fs /dev/mmcblk0p2`.
- If you want to extend the root partition to a fixed size N (where N is number in GB)

`parted /dev/mmcblk0 resizepart 2 NG`.

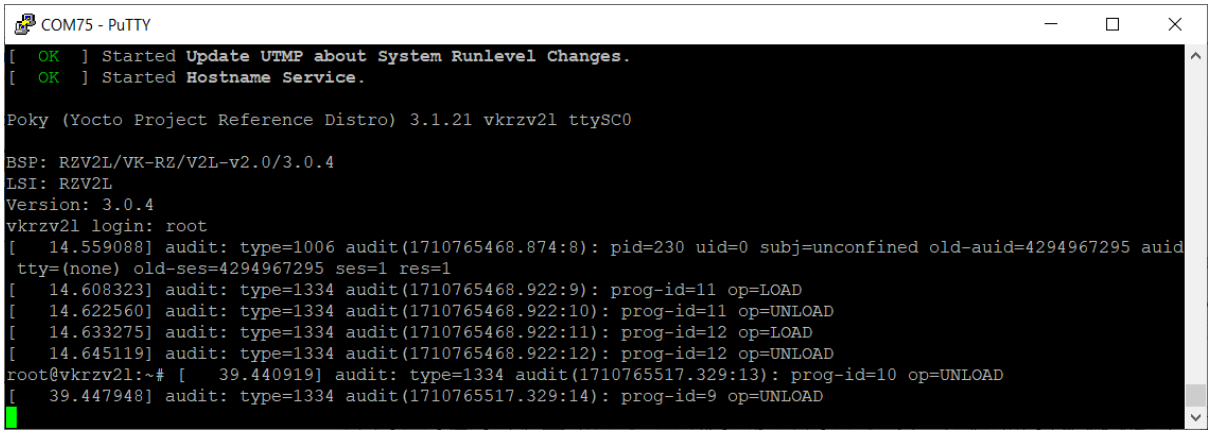
Fix filesystem block size: `resize2fs /dev/mmcblk0p2`.

You can make additional partitions & format them if you like:

`parted /dev/mmcblk0 "mkpart primary ext4 NG MG"`.

`mkfs.ext4 /dev/mmcblk0p3`.

- Make sure **SW1** is set to boot from **eMMC (1:OFF | 2:ON | 3:OFF)** & press reset.
- You should now be able to see login screen of the Yocto, use `root` to login.



Yocto Login screen

6. Launch the installed Linux images

Thanks to the convenient boot logic, launching is easy, it depends on correct setting of **SW1**

- To boot from the SPI flash: set the switch to **SPIBSC (1:ON | 2:X | 3:X)**.

µSD and eMMC shares the same **sdhi** hardware channel (**ch 0**) so:

→ If you want U-boot to have access to the µSD after boot: set **(1:ON | 2:X | 3:OFF)**.

→ If you want U-boot to have access to the eMMC after boot: set **(1:ON | 2:X | 3:ON)**.

That's why when Linux is started by U-boot from µSD or eMMC media, make sure the correct overlay is selected at **that** media location: `vkrz-sdhi0-(sdhi/emmc).dtbo`. (in uEnv.txt).

- To boot from eMMC: set the switch to **eMMC (1:OFF | 2:ON | 3:OFF)**.
- To boot from µSD, (using the Renesas bootloader) set to **eSD (1:OFF | 2:ON | 3:ON)**.
- To boot from Ethernet: set the switch to **SPIBSC** and make sure µSD slot is Empty.
  When U-boot can't find SD card, it will try to boot from its **serverip**, **tftpdir** & **netrootfs** environment parameters. (serverip is the address, where U-boot will look for NFS & TFTP servers, tftpdir shows where the boot directory is on the server and netrootfs → where root directory is on the server).

7. Apply Overlays

Overlays are way to tell Linux to use or not a given periphery. They can even point a specific hardware for a same periphery and form configurations for different version of a board. For example in version 1, the board can use **Realtek** audio driver, in other **someone's else**, in third no audio at all. Management of the overlays is done through **uEnv.txt**, located in **/boot** directory of the particular linux image. This file is analyzed during boot of the Linux image, so every change in that file takes effect after boot. To apply overlay, you need to line it up in the following row:

```
fdt_extra_overlays=vkrz-audio.dtbo vkrz-dsi-vklcd-ee0700.dtbo
```

All available overlays that can be applied are listed in **/boot/overlays**, but simultaneously only these can be activated at the same time.

- vkrz-cm33.dtbo
- vkrz-audio.dtbo (not available at the moment, but will be, very soon)
- vkrz-csi-imx219.dtbo
- vkrz-dsi-av_disp2.dtbo / vkrz-dsi-vklcd07.dtbo / vkrz-dsi-vklcd-ee0700.dtbo / vkrz-dsi-hdmi.dtbo
- vkrz-exp-riic2.dtbo
- vkrz-exp-rspi1.dtbo
- vkrz-exp-pwm0.dtbo / vkrz-exp-scfi1_rts_cts.dtbo
- vkrz-exp-pwm1.dtbo
- vkrz-exp-pwm2.dtbo
- vkrz-exp-scfi2.dtbo / vkrz-exp-scfi2_rts_cts.dtbo
- vkrz-sdhi0-sdhi.dtbo / vkrz-sdhi0-emmc.dtbo

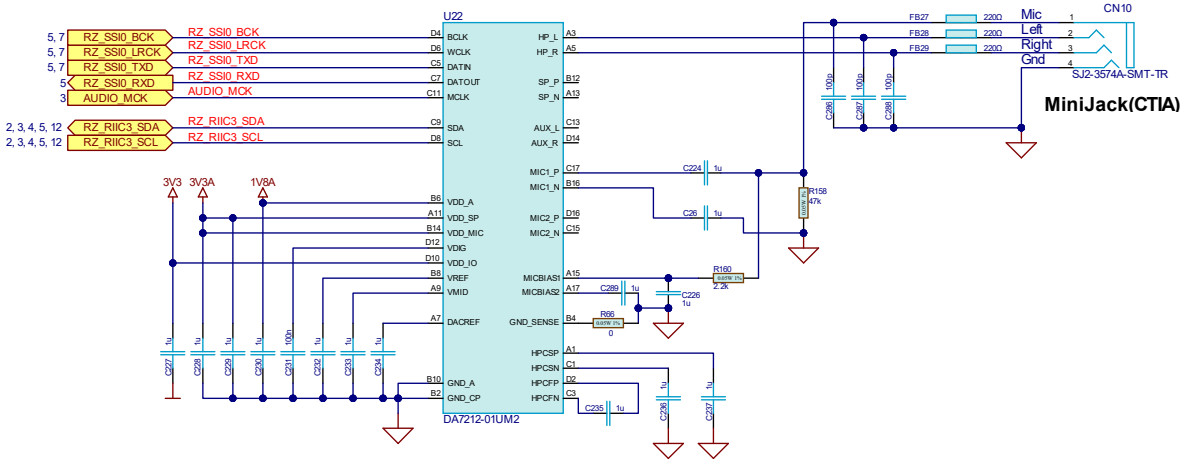
8. Use various periphery in Linux

To demonstrate using different periphery, we are going to use the Debian image, for other images (Yocto for example) the commands could be not available or could be slightly different, so don't worry if some of them do not work, just google how to do it for your particular distribution.

8.1 Audio

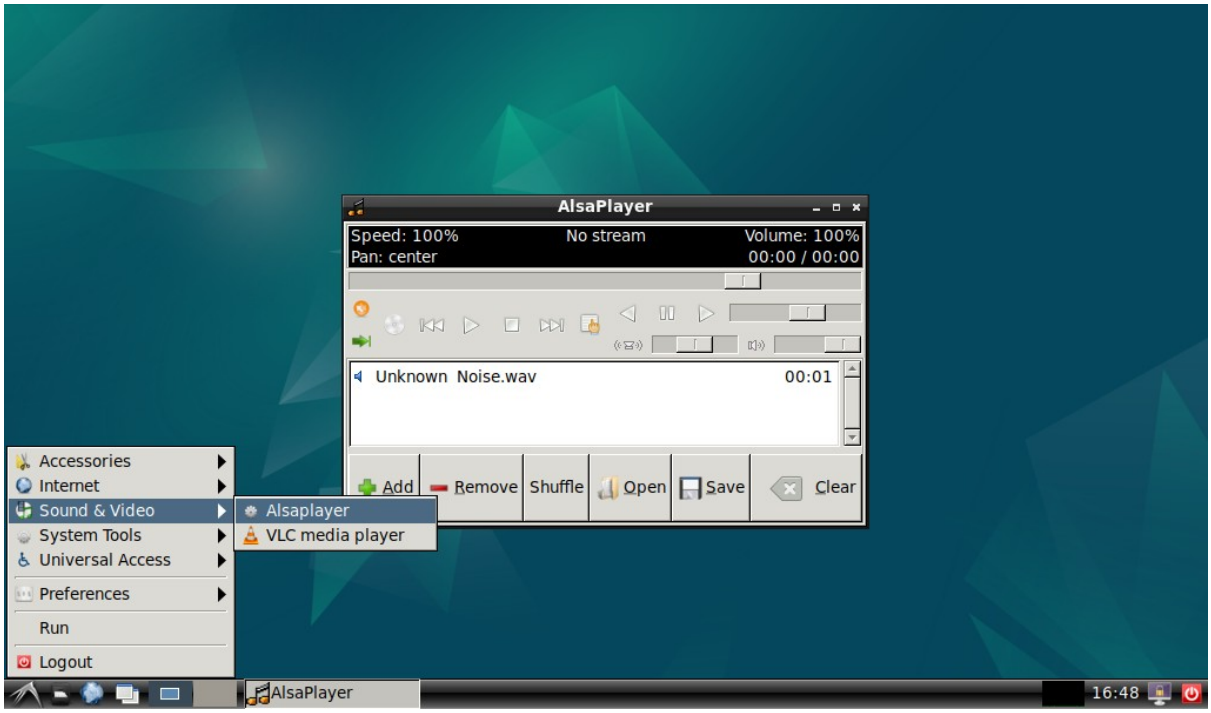
Make sure you are applied **vkrz-audio.dtbo** overlay in **/boot/uEnv.txt**, i.e. you should see it is included in the following row:

```
fdt_extra_overlays=vkrz-audio.dtbo vkrz-dsi-vklcd-ee0700.dtbo
```



3,5mm CTIA Mini Jack

- Plug headphones in the jack & type: `aplay /usr/share/sounds/alsa/Noise.wav`.
- You can also play it through the GUI: just go to **Start/Sound & Video /Alsaplayer**. press Add, browse to /usr/share/sounds/alsa/Noise.wav, press Play.
- You should now be able to hear white noise sample. If for some reason you can't here anything, open alsamixer and check if **Card** (in the upper left corner) is **audio-da7212**. If it is different, press **F6** (Select sound card) and choose **audio-da7212**, also find **Mixout Left DAC Left** & **Mixout Right DAC Right** and make sure they are not **Off**.

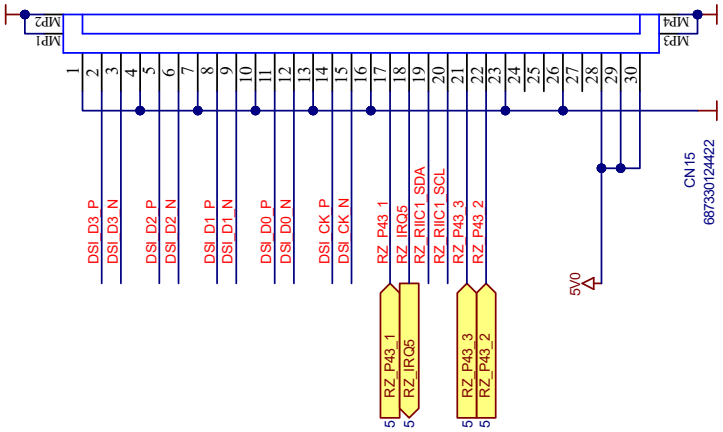


AlsaPlayer application

8.2 MIPI DSI Display

Make sure you are applied one of these overlays in **/boot/uEnv.txt**.

vkrz-dsi-av_disp2.dtbo / **vkrz-dsi-vklcd07.dtbo** / **vkrz-dsi-vklcd-ee0700.dtbo** / **vkrz-dsi-hdmi.dtbo**.

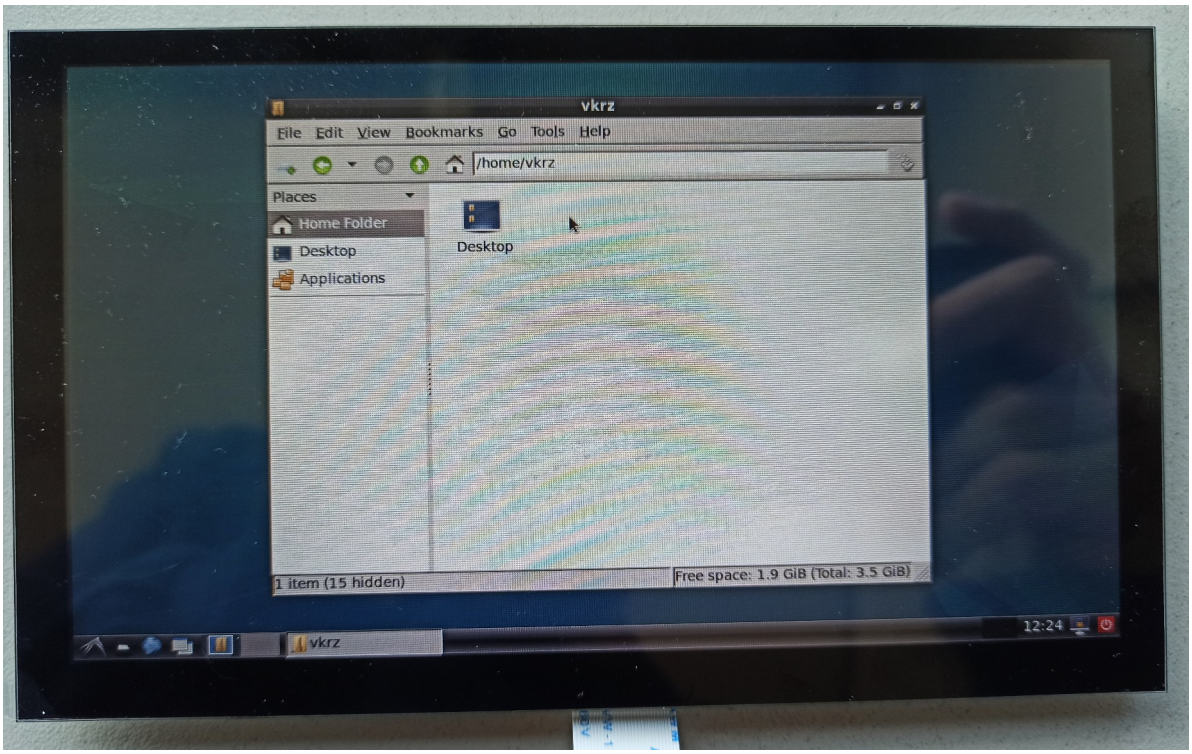


30 pin FPC MIPI DSI Connector

Let's try each of them:

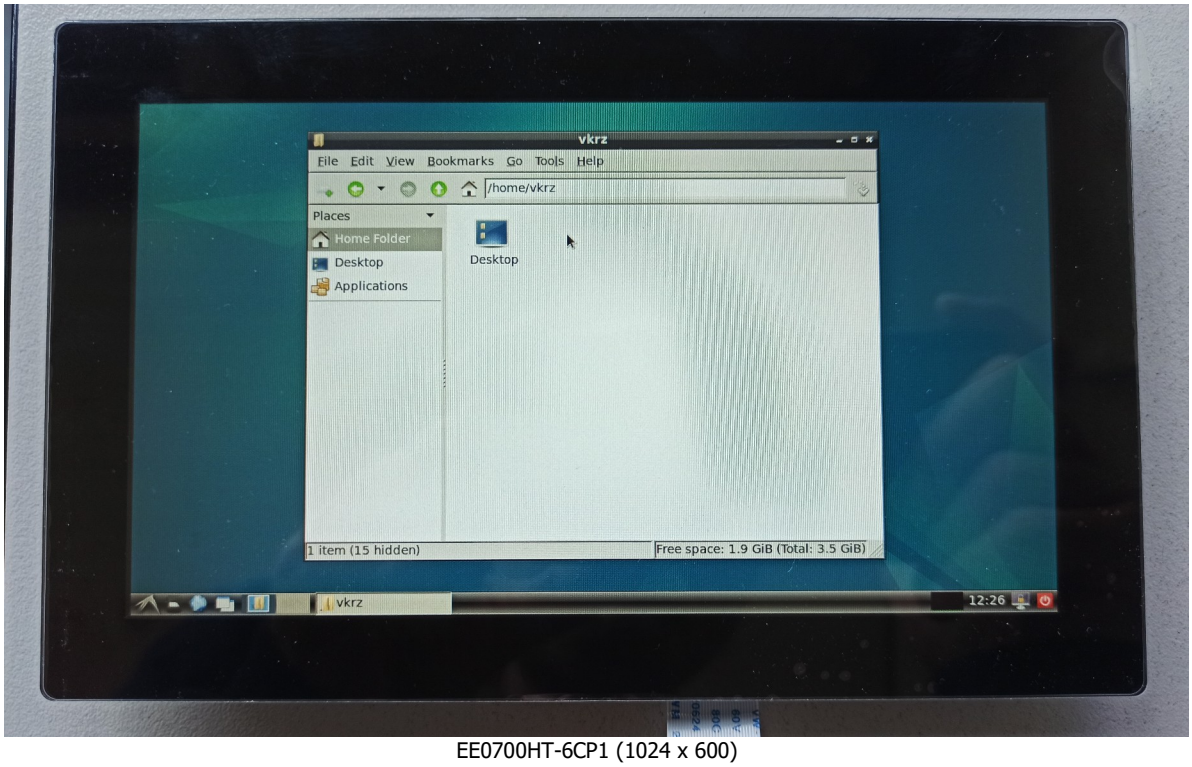
vkrz-dsi-vklcd-ee0700 display FPC cable orientation

- **fdt_extra_overlays=vkrz-dsi-vklcd07.dtbo.**



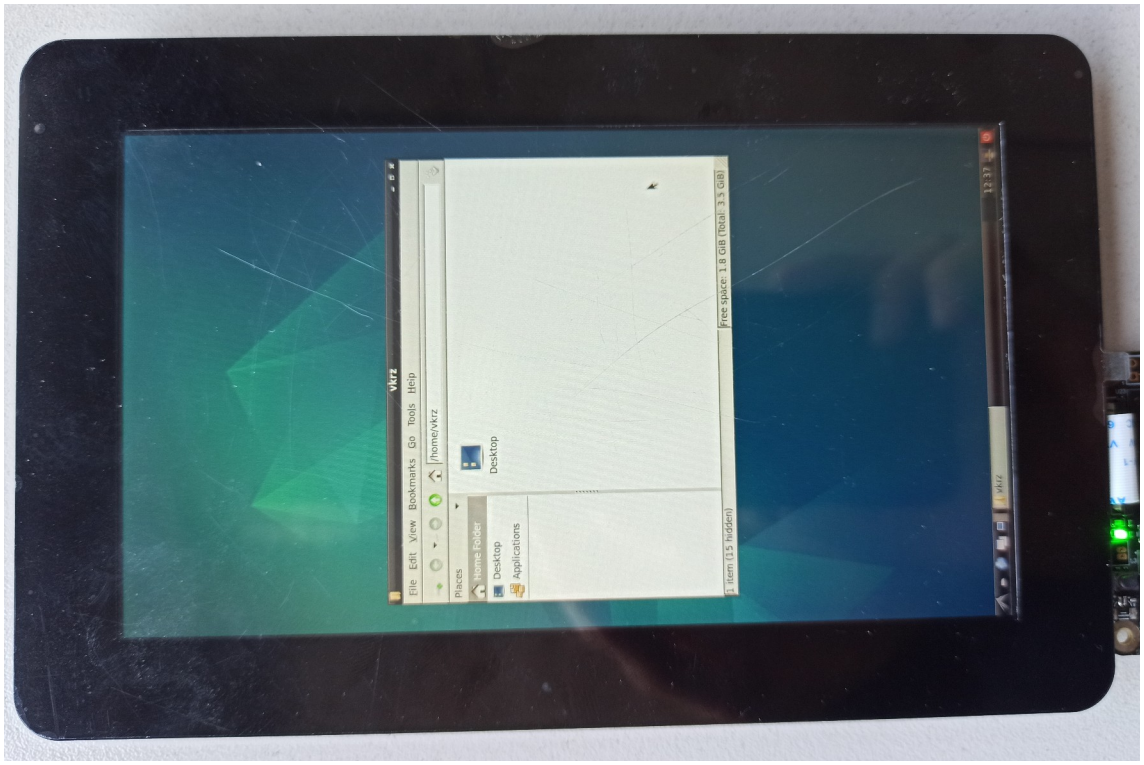
FORMIKE KWH070KQ40-C08 (1024 x 600)

- `fdt_extra_overlays=vkrz-dsi-vklcd-ee0700.dtbo.`



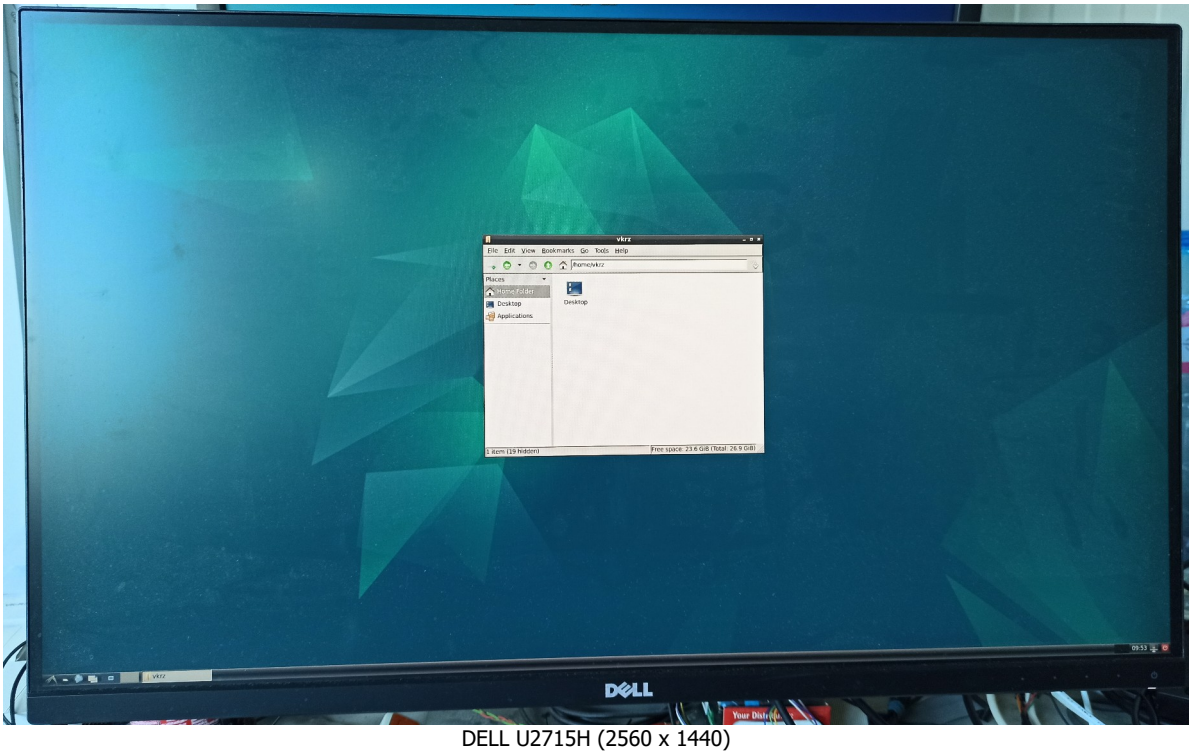
EE0700HT-6CP1 (1024 x 600)

- `fdt_extra_overlays=vkrz-dsi-av_disp2.dtbo.`



AES-ACC-MAAX-DISP2 (720 x 1280)

- `fdt_extra_overlays=vkrz-dsi-hdmi.dtbo`.

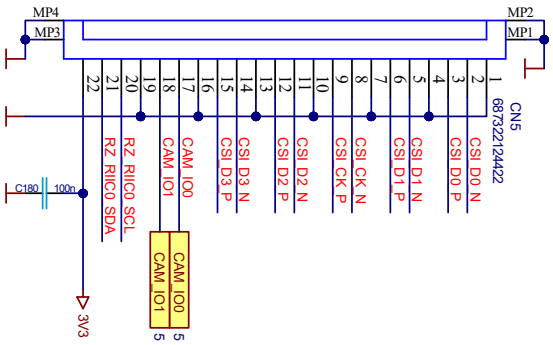


DELL U2715H (2560 x 1440)

8.3 MIPI CSI Camera

Make sure you are applied **vkrz-csi-imx219.dtbo** overlay in **/boot/uEnv.txt**.

```
fdt_extra_overlays=vkrz-csi-imx219.dtbo vkrz-dsi-vklcd-ee0700.dtbo
```



22 pin FPC MIPI CSI Connector

- Install tool to catch the video stream: `sudo apt-get install vlc`.
- Download the CSI v4l2-init script and put it in the home directory (`~`).

If the hardware is available, the script will set default settings through these rows:

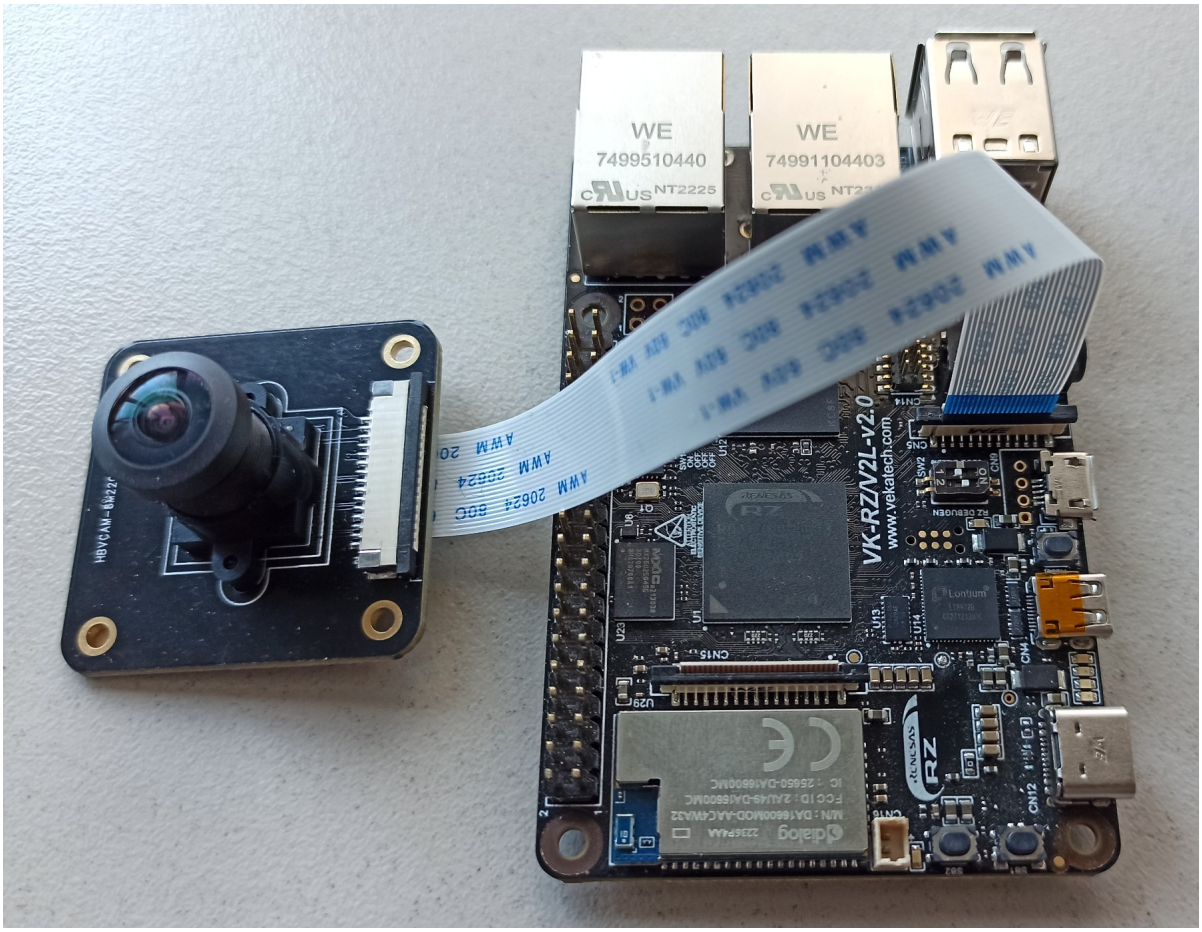
media-ctl -d /dev/media0 -l "'rzg2l_csi2 10830400.csi2':1 -> 'CRU output':0 [1]"

media-ctl -d /dev/media0 -V "'rzg2l_csi2 10830400.csi2':1 [fmt:UYVY8_2X8/$imx219_res field:none]"

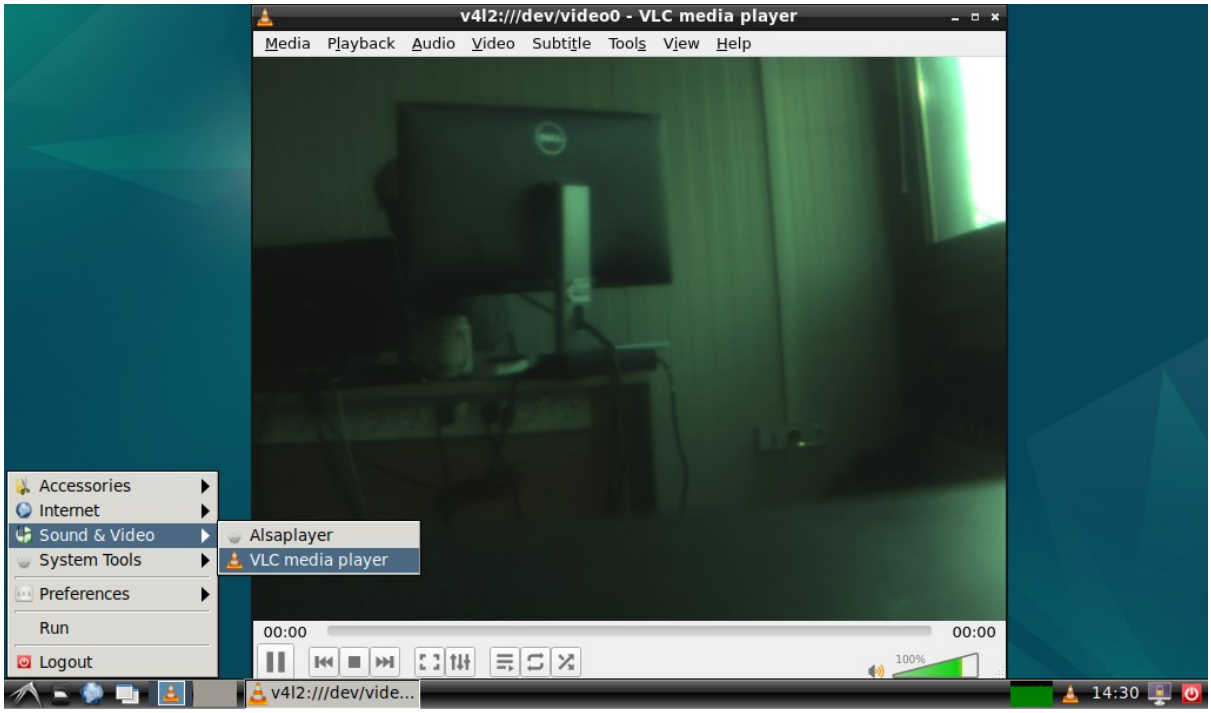
media-ctl -d /dev/media0 -V "'imx219 0-0010':0 [fmt:UYVY8_2X8/$imx219_res field:none]"

If default format (UYVY8_2X8) does not suit your needs, you can always change it.

- When you are happy with settings in the script, execute it: `~/v412-init.sh`.
- Open VLC through the GUI: go to **Start/Sound & Video/VLC media player**.
- Go to **Media** → **Open Capture Device …** → **Capture Device** Tab
  for **Video device name** select **/dev/video0** and press **Play**.
- You should now be able to see the video input stream.



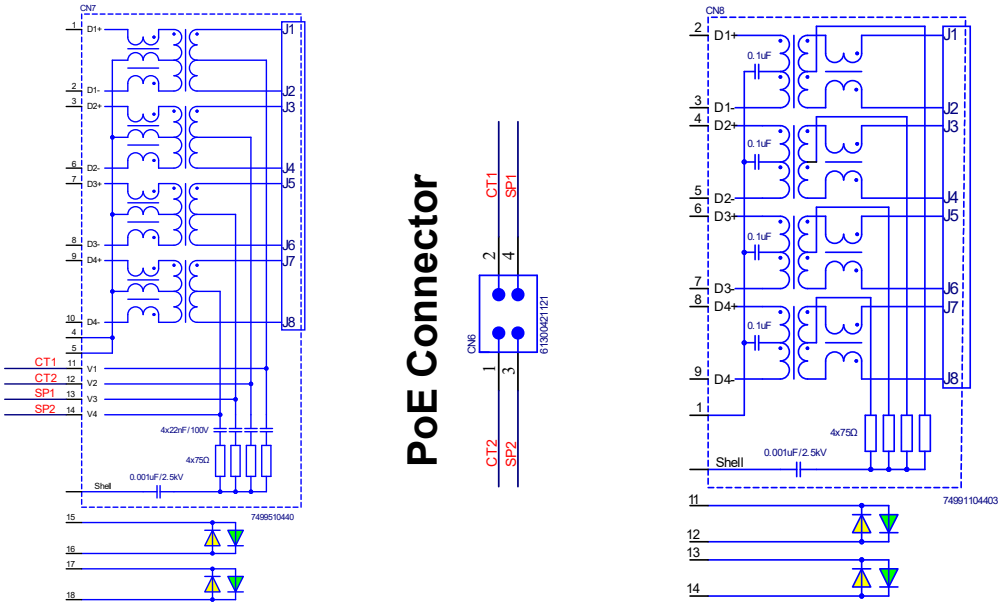
vkrz-csi-imx219 camera FPC cable orientation



imx219, set to (640 x 480) to be viewable on the screen

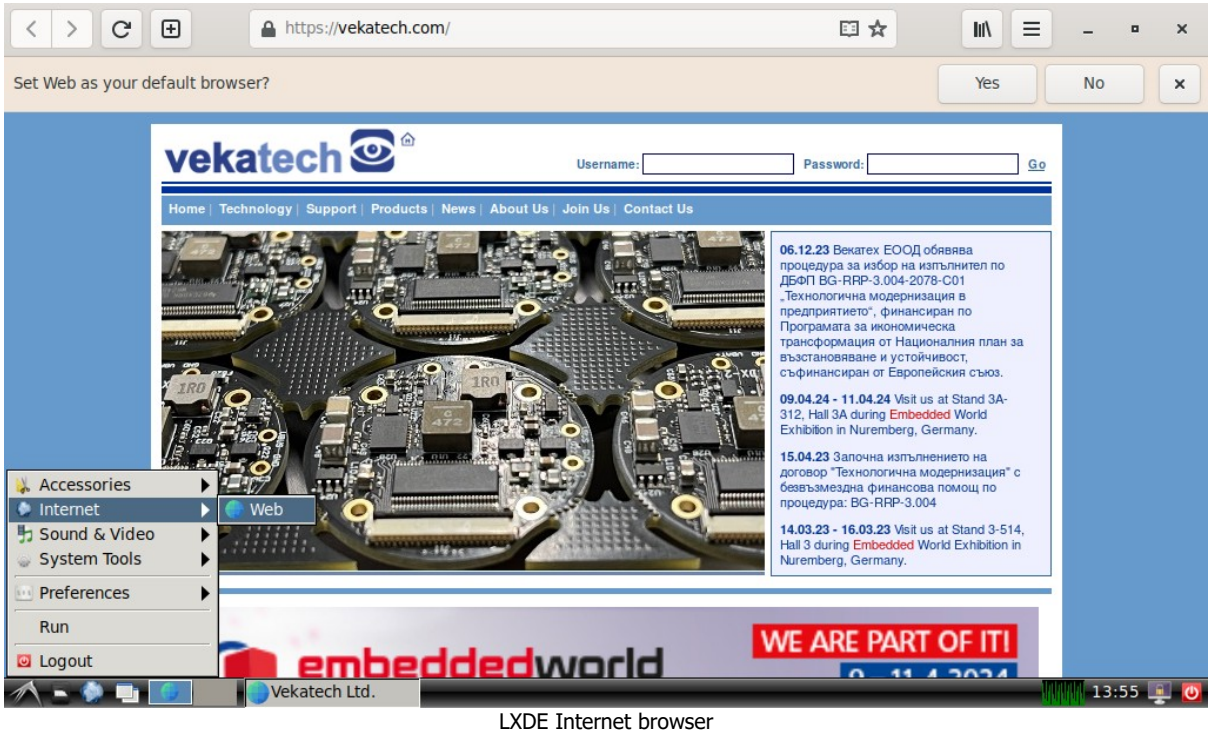
8.4 Ethernet

Ethernet is enabled by default and does not need overlay. Plug the **cat** cable into **RJ45** connector and in a couple of seconds the board should get an **IP** from the local **DHCP** server.



RJ45 connectors

- You can check what IP is assigned: `sudo ifconfig`.
- You can probe if Host is reachable: `ping vekatech.com`.
- You can browse an internet site: go to **Start/Internet/Web**.



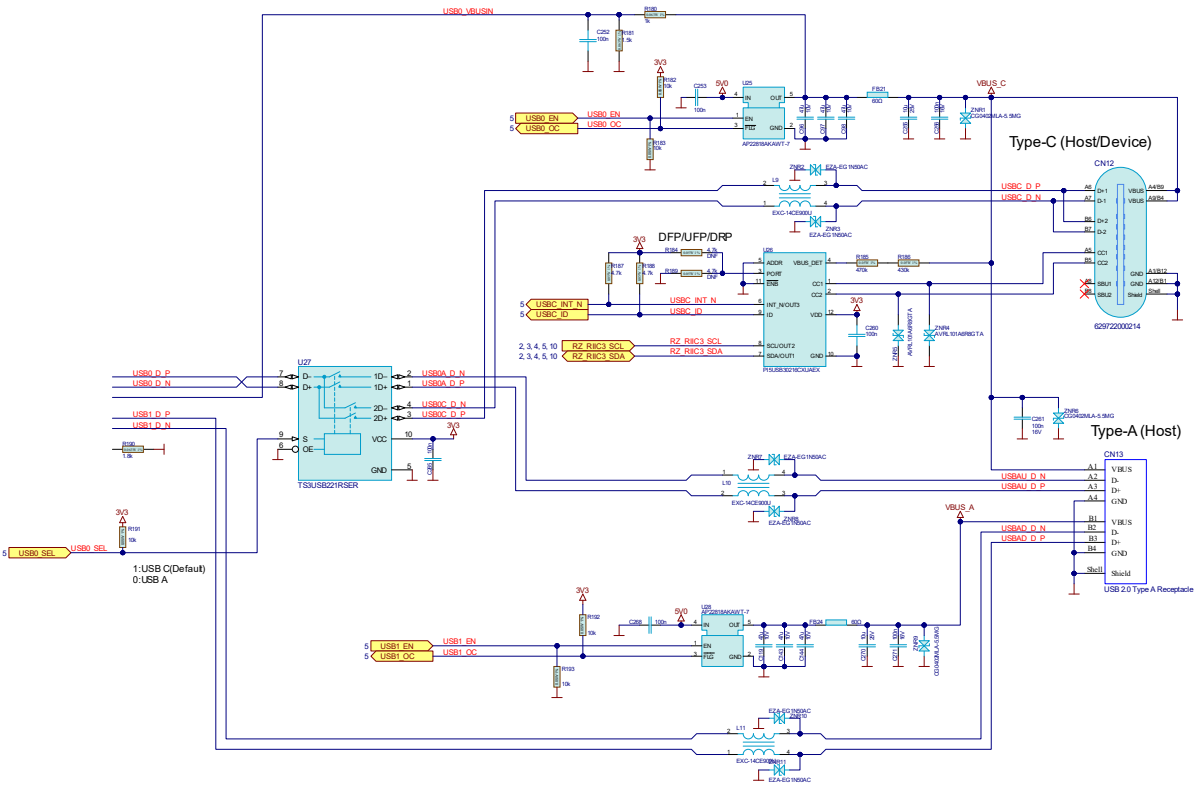
LXDE Internet browser

8.5 USB

USB is also enabled by default and does not need overlay. To test it, plug a USB device such as: Mouse, Keyboard, USB Flash, USB Camera & whatever other USB device you can think of.

- Mouse: It is Plug & Play device , you don't need to do anything.
- Keyboard: Also Plug & Play device, you don't need to do anything.
- Flash Drive: most of the latest Linux distributions mounts the drive automatically, (Debian 12 is not an exception), so it is again another Plug & Play device.

One of the USB ports (PORT 0) can be redirected to connectors: Type C or Type A (upper). The selection is made by a GPIO pin **P5_2**. By default it is wired to Type C, but Once U-boot takes control over the board it is switched to Type A and Type C becomes Power Only.



USB Type A & Type C connectors

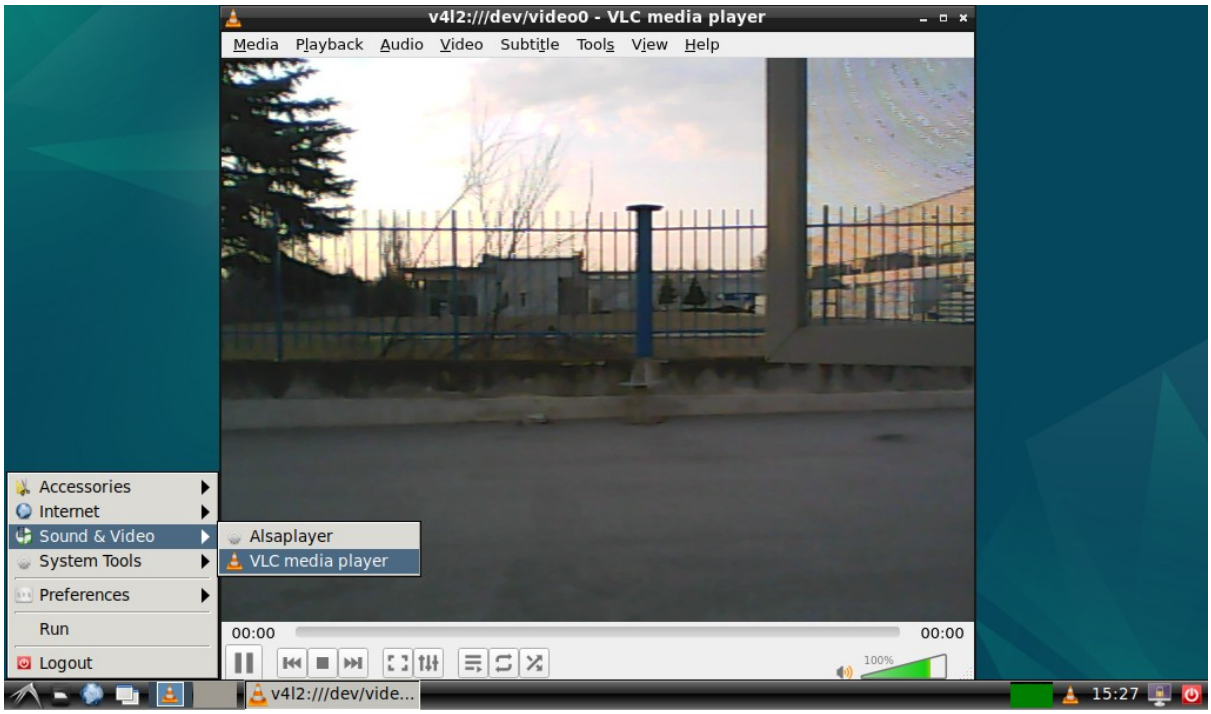
- Camera: usually you need software to see the stream, the example below is with VLC, but ffmpeg or others are also a great choice.

  → Open VLC through the GUI: go to **Start/Sound & Video/VLC media player**.

  → Go to **Media** → **Open Capture Device …** → **Capture Devie** Tab

  for **Video device name** select **/dev/video0** and press **Play**.

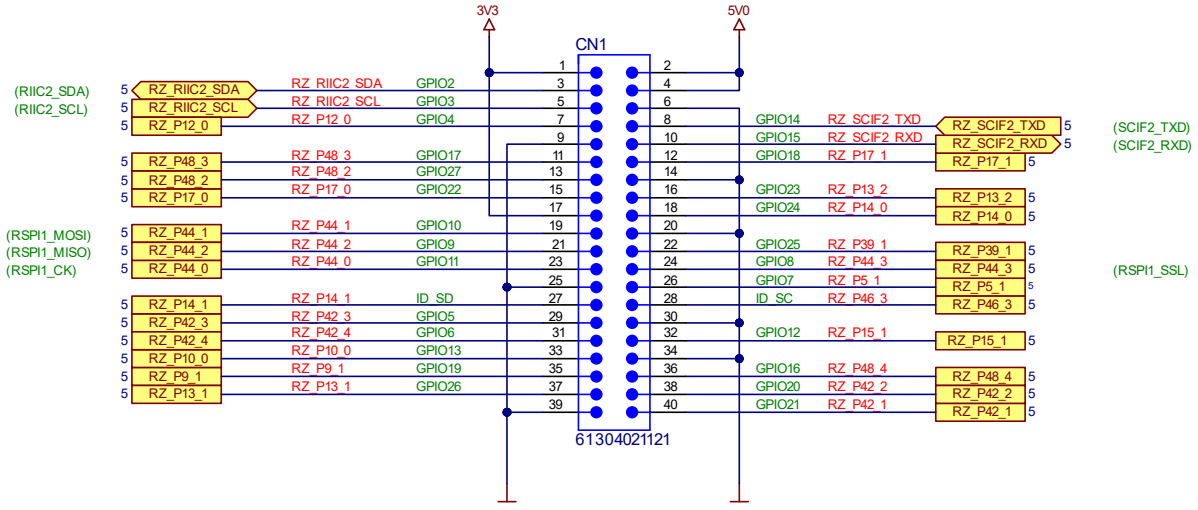
  → You should now be able to see the USB video stream.



Logitech C170

8.6 GPIO

GPIOs are enabled by default and do not need overlay.



40 pin Extension Header

- You have to tell the system which pin you want to manipulate. Let's say the pin is "**Pport_pin**". You have to calculate the internal pin number: (**port** x **8**) + **pin** + **120**, if **P48_4** is desired to be controlled, the internal number is: (**48** x 8) + **4** + 120 = **508**.
  The way to tell the system is with these commands: `sudo su`
  `echo 508 > /sys/class/gpio/export`.
- If you want to get the value of **P48_4** pin, you can type this:
  `echo in > /sys/class/gpio/P48_4/direction`.
  `cat /sys/class/gpio/P48_4/value`.
- If you want to set **P48_4** pin to **HI**, you can type this:
  `echo out > /sys/class/gpio/P48_4/direction`.
  `echo 1 > /sys/class/gpio/P48_4/value`.
- If you don't want to use the pin anymore you can release the resource:
  `echo 508 > /sys/class/gpio/unexport`.

8.7 PWM ???

Make sure you are applied **vkrz-exp-(pwm0/pwm1/pwm2/pwm3).dtbo** overlay.

```
fdt_extra_overlays=vkrz-exp-pwm0.dtbo vkrz-exp-pwm1.dtbo vkrz-exp-pwm2.dtbo
```

? PWM[0] pins go to Pin12 (**xTIOCnm**) of 40pin extension connector.
--PWM[1] pin go to Pin33 (**xTIOCnm**) of 40pin extension connector.
PWM[2] pin go to Pin32 (**xTIOCnm**) of 40pin extension connector.
PWM[3] pin go to Pin32 (**xTIOCnm**) of 40pin extension connector.

- You have to tell the system which pwm interface you want to manipulate. For that purpose you have to examine /sys/class/pwm folder. Depending on what overlays you are included in uEnv.txt, you can see up to 5 pwmchip folders: **pwmchip0**, **pwmchip1**, **pwmchip2**, **pwmchip3**, **pwmchip4**, **pwmchip5**. Usually this is the case when some of the displays & all pwm overlays are used (the current example is exactly that case), so the mapping is as follows:
  Display **brightness** →pwmchip**0** | PWM[**0**] → pwmchip**1**, | PWM[**1**] → pwmchip**2**.
  Les's say **PWM[1]** needs to be set to 10%/1kHz, so pwmchip2 folder should be used.
  The way to tell that to the system is with these commands: `sudo su`
  `echo 0 > /sys/class/pwm/pwmchip2/export`.
- Now PWM[1] interface **xTIOCnm** should be available as **pwm0**, and can be configured:

period in µs `echo 1000000 > /sys/class/pwm/pwmchip2/pwm0/period`.

duty_cycle in µs `echo 100000 > /sys/class/pwm/pwmchip2/pwm0/duty_cycle`.

+ polarity `echo normal > /sys/class/pwm/pwmchip2/pwm0/polarity`.

- polarity `echo inversed > /sys/class/pwm/pwmchip2/pwm0/polarity`.

- If you want to turn **PWM[1] ON**, you can type this:
  `echo 1 > /sys/class/pwm/pwmchip2/pwm0/enable`.
- If you want to turn **PWM[1] OFF**, you can type this:
  `echo 0 > /sys/class/pwm/pwmchip2/pwm0/enable`.
- If you don't want to use **PWM[1]** any more, release **pwm0** resource:
  `echo 0 > /sys/class/pwm/pwmchip2/unexport`.

8.8 SPI

Make sure you are applied **vkrz-exp-rspi1.dtbo** overlay in **/boot/uEnv.txt**.

```
fdt_extra_overlays=vkrz-exp-rspi1.dtbo vkrz-dsi-vklcd-ee0700.dtbo
```

SPI[1] pins go to Pin19 (**MOSI**), Pin21 (**MISO**), Pin23 (**SCLK**), Pin24 (**CS0**) of 40pin ext. connector.

- Install SPI software: `sudo apt-get install spi-tools`.
- See the default config of SPI bus: `sudo spi-config -d /dev/spidev0.0 -q`.
- Set the config you need, use `spi-tools -help` to see what can be configured.
- Short MOSI & MISO together (Pins 19 & 21), so a Loopback test can be run.
- Send some data and see if it is received properly: `echo -n -e "1234567890" | sudo spi-pipe -d /dev/spidev0.0 -s 10000000 | hexdump`.
- If everything is OK you should see the digits: `3231 3433 3635 3837 3039`.
- Remove the short and try again sending the digits: `echo -n -e "1234567890" | sudo spi-pipe -d /dev/spidev0.0 -s 10000000 | hexdump`.
- Now you should see a complete flat line (Vcc): `ffff ffff ffff ffff ffff`.

8.9 I2C

Make sure you are applied **vkrz-exp-riic2.dtbo** overlay in **/boot/uEnv.txt**.

```
fdt_extra_overlays=vkrz-exp-riic2.dtbo vkrz-dsi-vklcd-ee0700.dtbo
```

I²C[2] pins go to Pin3 (**SDA**) & Pin5 (**SCL**) of 40pin extension connector.

- Install I²C software: `sudo apt-get install i2c-tools`.
- Now all I²C devices on the system bus 3 can be seen: `sudo i2cdetect -y -a -r 3`.

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- UU -- -- -- -- -- -- -- UU -- -- 1d -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: UU 51 52 53 54 55 56 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- UU -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

- Read one byte from addr 0x00 of E²PROM: `sudo i2cget -y 3 0x51 0x00 b`.
- Write byte 0x55 to addr 0xF0 of E²PROM: `sudo i2cset -y 3 0x51 0xF0 0x55 b`.

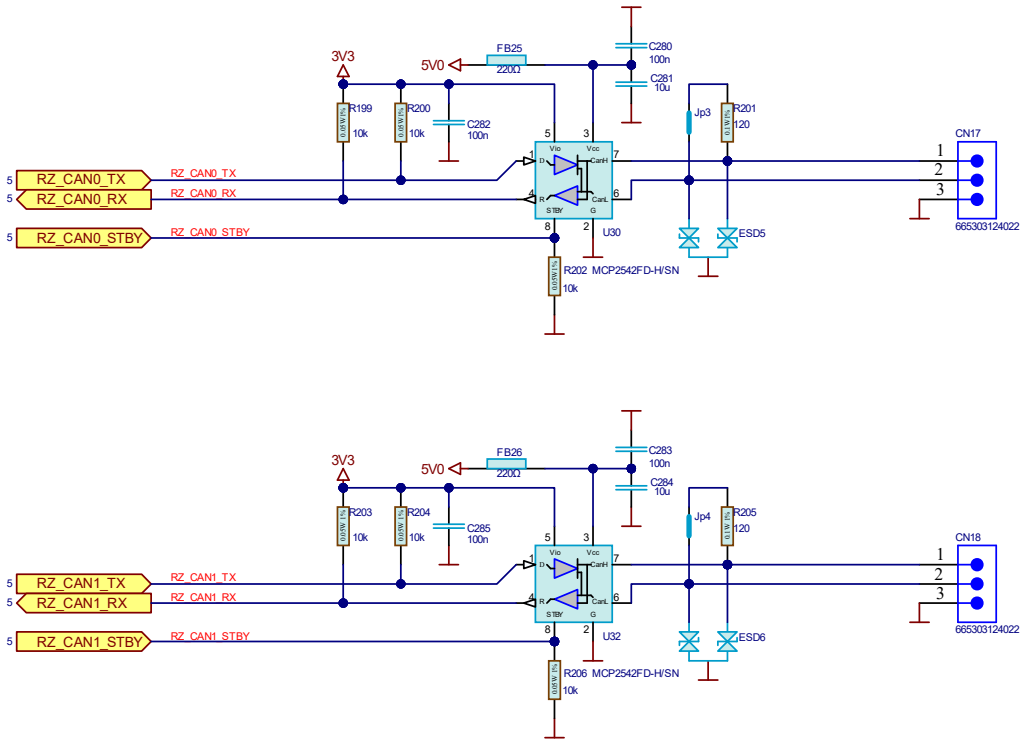
If you prefer Python you can get the same result using the script below, but you first have to install `sudo apt-get install python3-smbus`.

```
import sys
import smbus

bus = 3

def _list_devices(_bus):
    sys.stdout.write('    ')
    for _address in range(16):
        sys.stdout.write(' %2x' % _address)  # Print header

    for _address in range(128):
        if not _address % 16:
            sys.stdout.write('\n%02x:' % _address)  # Print address
        if 2 < _address < 120:  # Skip reserved addresses
            try:
                _bus.read_byte(_address)
                sys.stdout.write(' %02x' % _address)  # Device address
            except:
                sys.stdout.write(' --')  # No device detected
        else:
            sys.stdout.write('    ')  # Reserved

    sys.stdout.write('\n')

_list_devices(smbus.SMBus(bus))
```

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- 1d -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- 51 52 53 54 55 56 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

8.10 CAN

CANs are enabled by default and do not need overlay.



CAN connectors.

Use the CAN modules:

- Install CAN software: `sudo apt-get install can-utils`.
- Turn off the can modules: `ip link set can0 down` && `ip link set can1 down`.
- Set the config you need: `ip link set can0 type can bitrate 2000000 dbitrate 2000000 fd on`.

`ip link set can1 type can bitrate 2000000 dbitrate 2000000 fd on`.

- Turn on the can modules: `ip link set can0 up` && `ip link set can1 up`.
- You can run a CAN test by connecting CN17 with CN18 with a cable and set one to send and other to receive: Let's say CAN0 will receive messages: `candump can0`.

  → Open another bash terminal instance and make CAN1 to send message:

  `cansend can0 123#0102030405060708`.

  → Use `cansend -help` for more info for the format of the can messages (in this case, 11bit address mode is used → [addr: **123**] & full msg len [data: **0102030405060708**])

8.11 UART ???

Make sure you are applied **vkrz-exp-scfi2.dtbo / vkrz-exp-scfi2_rts_cts.dtbo** overlay. Choose whether you need flow control or not and specify one or the other in **/boot/uEnv.txt**.

```
fdt_extra_overlays=vkrz-exp-scfi1.dtbo vkrz-dsi-vklcd-ee0700.dtbo
```

UART[2] pins go to Pin8 (**TX**), Pin10 (**RX**), PinXX (**CTS**) PinYY (**RTS**) of 40pin extension connector.

- Install port software: `sudo apt-get install screen`.
- Short TX & RX together (Pins 8 & 10), so a Loopback test can be run.
- Execute: `screen /dev/ttySC2` or `screen -f /dev/ttySC2` (if flow control overlay)
- Type something and you should now be able to see what you are typing.
- If you remove the short, you won't see what you are typing.
- Press **Ctrl+A** and then **K** to exit, (you may need to apply with **y**)

9. Using .NET in Linux

There is install script prepared from microsoft on their site. (its source is also available here).

- Get the script: `wget https://dot.net/v1/dotnet-install.sh`.
- Make it executable: `chmod +x dotnet-install.sh`.
- If you want to create projects, install the whole SDK: `./dotnet-install.sh --channel 6.0 --architecture arm64 --install-dir ~/.NET`.
- If you want to run projects, install the runtime: `./dotnet-install.sh --runtime dotnet --channel 6.0 --architecture arm64 --instal-dir ~/.NET`.
- Set env. variable: `echo 'export DOTNET_ROOT="$HOME/.NET"' >> ~/.bashrc`.
- Update the path: `echo 'export PATH="$PATH:$HOME/.NET"' >> ~/.bashrc`.
  Apply the changes, so you can use **dotnet** directly in terminal: `source ~/.bashrc`.
- Suppose you don't have the ready project and you want to make one with SDK, type this: `dotnet new console -o I2cDeviceScanner` && `cd I2cDeviceScanner`.

Edit the source file, writing a program which can report all I^2C devices on the bus 0. For this you are going to need **I^2C** lib, so install: `dotnet add package System.Device.Gpio`.
Pasting following content in **Program.cs**, achieves similar behavior like the py script in **8.9**

```
using System;
using System.Device.I2c;
using System.Threading;

class Program
{
  static void Main(string[] args)
  {
    const int i2cBusId = 3; // The I2C bus ID. Adjust this depending on your hardware.

    Console.WriteLine("Scanning for I2C devices...");

    for (int deviceAddress = 1; deviceAddress < 128; deviceAddress++)
    {
      var settings = new I2cConnectionSettings(i2cBusId, deviceAddress);
      using (var device = I2cDevice.Create(settings))
      {
        try
        {
          // Attempt to write an empty byte array to check for device acknowledgment.
          // This method varies depending on the device; some might require reading or writing
specific registers.
          device.WriteByte(0);
          Console.WriteLine($"Found device at address 0x{deviceAddress:X2}");
        }
```

```
            catch
            {
              // If an error occurs, it's likely no device is present at this address.
            }
          }

          // Delay to prevent spamming the I2C bus too quickly.
          Thread.Sleep(50);
        }

        Console.WriteLine("Scan complete.");
      }
    }
```

- Suppose you have finished project, called I2cDeviceScanner, to build & run with the SDK execute: **`dotnet run`**.
- If you have to deploy .NET **FDD** app: execute: **`dotnet publish -r linux-arm64 -f net6.0 -c Release –no-self-contained`**.
- If the system do not allow you to use the **I²C** resource, add **`vkrz`** user to the **`i2c`** group: **`sudo usermod -aG i2c vkrz`**.
- If you have only the runtime & want to just launch the I2cDeviceScanner app, execute: **`dotnet ~/I2cDeviceScanner/bin/Debug/net6.0/I2cDeviceScanner.dll`**.

You will have to get the following output:

```
Scanning for I2C devices...
Found device at address 0x1A
Found device at address 0x50
Found device at address 0x51
Found device at address 0x52
Found device at address 0x53
Found device at address 0x54
Found device at address 0x55
Found device at address 0x56
Found device at address 0x57
Found device at address 0x68
Scan complete.
```

- If you need to debug I2cDeviceScanner app it's get tricky. Generally for trivial projects it is best to build & debug .NET apps on a regular PC and when they are finished, just run the deployed **dll**s on the target device. However when your project uses a board specific periphery, which is only available on the board (let's say SPI, I^2C, UART, GPIO and whatever), you will have to debug it on the board, remotely and it is kind of slow). The IDEs supporting such debug are: **V**isual **S**tudio & **VS Code**. For the last one, you will also going to need **Remote - SSH** & **C# Dev Kit** extensions.

  → Install .NET SDK on the **dev**elopment **PC** (Win 10 in this case).

  → On the dev PC, choose one of the IDEs (VS Code in this case) and install it.

  → On the dev PC, install ssh connection tool called PuTTY, (or use Win10 built in ssh)

  => If you prefer PuTTY's built in default one: plink (supported only for VS Code debug).

  Go to PuTTY's install folder & launch **puttygen**. Generate some key pairs (EdDSA).

  Select EdDSA radio button & Ed25519(255 bits) dropdown & hit ↵.

  Save private key in `%USERPROFILE%\.ssh\id_ed25519.ppk`.

  Mark the public key & press (`Ctrl+c`) to copy it to the clipboard.

  => If you prefer Win10's built in default one: ssh (fully supported by VS Code).

  Go ot Win10's search bar & launch **cmd**. Generate some key pairs (EdDSA).

  Type: `ssh-keygen -t ed25519` & hit ↵. On every prompt you get apply with ↵.

  View the public key in the cmd: `type %USERPROFILE%\.ssh\id_ed25519.pub`.

  Mark it & press (`Ctrl+c`) to copy it to the clipboard.

  → On the target board (VK-RZ/G2LC in this case) make a folder `mkdir -p ~/.ssh`.

  → On the target board, transfer the public key from dev PC's clipboard:

  `echo <(Ctrl+v) to paste from clipboard> >> ~/.ssh/authorized_keys`.

  → On the target board change the rules for newly created files and folders:

  `chmod 600 ~/.ssh/authorized_keys` && `chmod 700 ~/.ssh`.

  → On the target board, give some steady MAC address to the Ethernet interface, or the board will get different IPs from the DHCP, every time it boots. (yeah U-boot does this magic & generates random MACs on power up, if someone specific is not pointed out). If the IP is random, that could be potential problem, because passwordless ssh connection (i.e. new IP → new Host → new Key pairs generation) will be needed every time target board boots up. You don't want this gymnastic, I mean you want it, but only once, so ...

  Open the uEnv.txt file: `sudo nano /boot/uEnv.txt`.

  Strip the comment from `#ethaddr=xx:xx:xx:xx:xx:xx`: by removing the `#` in front.

  Leave the default MAC, or change it, save the file with `Ctrl+o` & exit nano with `Ctrl+x`.

Press **reset** on the board, so changes in the uEnv.txt to take effect & wait board to boot.

Log in with **vkrz** & **vkrzg2lc**, so you can continue with the setup.

→ On the target board, plug the Ethernet cable, if it's not already & wait DHCP to give IP. You can check if the IP is given by `sudo ifconfig`.

→ On the dev PC, test the connection and accept the fingerprint from the target board:

=> If plink is used:

`plink -ssh vkrz@<bord's IP> -i %USERPROFILE%\.ssh\id_ed25519.ppk`.

Type: `y`, to store the key in Plink's cache, ↵ & then `exit`, to terminate ssh connection.

=> If ssh is used:

`ssh vkrz@<bord's IP> -i %USERPROFILE%\.ssh\id_ed25519`.

Type: `yes`, to store the key in known_hosts, and then `exit`, to terminate ssh connection.

=> If keyless option is required, for some reason (not recommended), there is solution:

`plink -ssh vkrz@<bord's IP> -pw <vkrz's password i.e. vkrzv2l>`.

You see, that's why it's not recommended, password is visible in the command line. For local debug sessions it's convenient & very tempting → save you the headache with the keys. If you prefer this way, you can just skip all nonsense from installing PuTTY, to here.

→ Note neither plink nor ssh needs user's target board password, and that is the goal, Congrats, you successfully accomplished passwordless ssh connection & VS Code needs it.

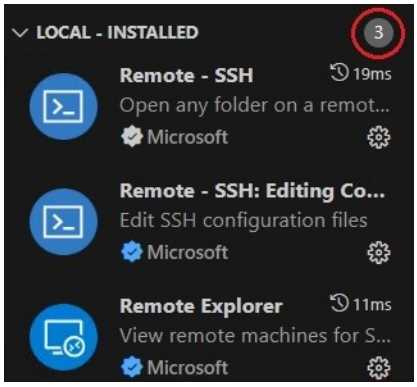
→ On the target board, install Visual Studio Remote Debugger:

Get the script: `curl -sSL https://aka.ms/getvsdbgsh -o vsdbg-install.sh`.

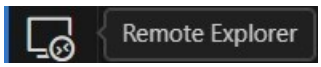
Make it executable: `chmod +x vsdbg-install.sh`.

Install it: `./vsdbg-install.sh -v latest -l ~/.NET/vsdbg`.

→ On the dev PC, launch VS Code & install the extensions mentioned earlier. Go to: **Extensions** tab `(Ctrl+Shift+X)` and type **Remote-SSH** in the Search Extensions bar, select it and install it. Extensions **Remote-SSH:Editing Configuration files** and **Remote Explorer** comes as companions to it, so you end up with all 3 installed **locally**.



Note that a new tab appears, on the left.

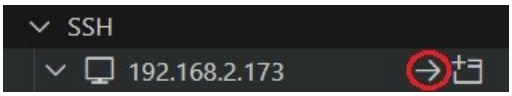


→ On the VS Code, go to that new tab (**Remote Explorer**) hit New Remote (the **+** sign)

For *Enter SSH Connection command*, type this: `ssh vkrz@<bord's IP>`. Keep in mind that only ssh works here. For now it's only it, fully supported by the Remote Explorer.

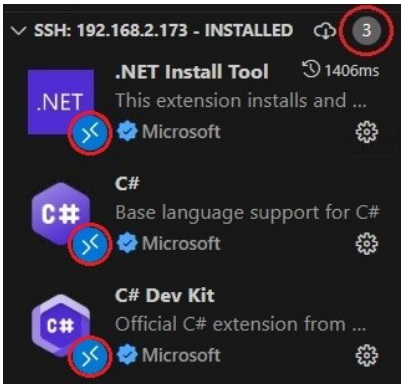
For *Select SSH configuration file to update* select the first, among the offered: (**config**), located in …**.ssh**\config. Close VS Code & reopen it, so the SSH clients tree is updated.

→ On the VS Code, go to **Remote Explorer** tab & hit Connect in Current Window... (→)

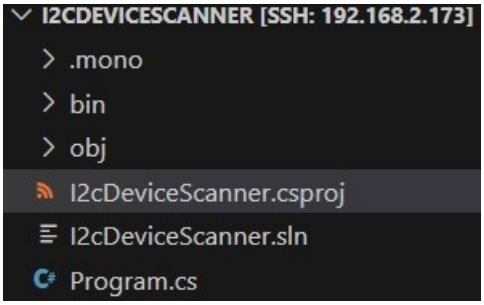


For first connect, you may be asked for *platform* select **Linux** & you should see green IP.

→ On the VS Code, go to **Extensions** tab again `(Ctrl+Shift+X)` & type **C# Dev Kit** in the Search Extensions bar, select it & install it. Extensions **C#** and **.NET Install Tool** comes as companions to it, so you end up with all 3 installed **remotely** (on the board).



→ On the VS Code, go to **Explorer** tab `(Ctrl+Shift+E)`, hit **Open Folder**, select **I2cDeviceScanner** and then **OK**. You should now see the project file, the source file and all folders part of the project. Editing files on the target board is now way more convenient, than through the command line, no doubt about that.



That is useful when developing project entirely on the target board, which is slow and

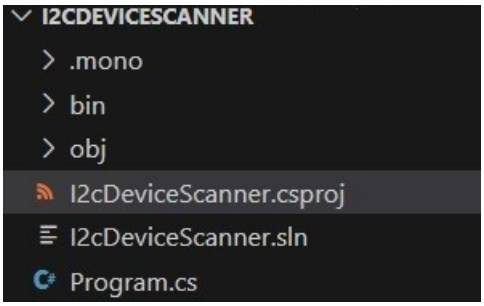
excruciating process. The point is only debug to be on the target board, so open the cmd.

→ On the dev PC, copy the I2cDeviceScanner app from the target board to the dev PC:

`scp -r vkrz@<bord's IP>:~/I2cDeviceScanner %USERPROFILE%\Documents`.

→ On the VS Code, go to **File** menu and select **Close Remote Connection**.

→ On the VS Code, go to **Explorer** tab **(Ctrl+Shift+E)**, hit **Open Folder**, browse to where you placed the project (**Documents** in this case) and select it (**I2cDeviceScanner**) then **Select Folder**. You should now see normal project tree:



→ On the VS Code, go to **Run** menu and select **Add Configuration...**

For *Select debugger*, select **C#** and you will be encouraged to fill **launch.json** file.

Press **Add Configuration...** (the blue button)

Select **{} .NET: Remote debugging - Launch Executable file (Console)**. and edit generated template to correspond the project settings. Make sure it looks like this:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Remote Launch - Framework-dependent [SSH-key]",
            "type": "coreclr",
            "request": "launch",
            "program": "~/.NET/dotnet",
            "args": ["bin/Debug/net6.0/I2cDeviceScanner.dll"],
            "cwd": "~/I2cDeviceScanner",
            "justMyCode": false,
            "stopAtEntry": false,
            "console": "internalConsole",
            "pipeTransport": {
                "pipeCwd": "${workspaceFolder}",
                "pipeProgram": "ssh",
                "pipeArgs": [
                    "vkrz@<board's IP>"
                ],
                "debuggerPath": "~/.NET/vsdbg/vsdbg"
            }
        }
    ]
}
```

→ If you prefer the **plink** ssh tool, make sure to change the following part:

=> with keys:

```
"pipeTransport": {
    "pipeCwd": "${workspaceFolder}",
    "pipeProgram": "plink",
    "pipeArgs": [
        "-ssh",
        "vkrz@<board's IP>",
        "-i",
        "${env:USERPROFILE}/.ssh/id_ed25519.ppk"
    ],
    "debuggerPath": "~/.NET/vsdbg/vsdbg"
}
```

=> keyless:

```
"pipeTransport": {
    "pipeCwd": "${workspaceFolder}",
    "pipeProgram": "plink",
    "pipeArgs": [
        "-ssh",
        "vkrz@<board's IP>",
        "-pw",
        "vkrzv2l"
    ],
    "debuggerPath": "~/.NET/vsdbg/vsdbg"
}
```

Once you change the launch.json, like you want it, you are ready to go debugging. Place a breakpoint at some row in Program.cs file and hit **F5** or ▷ in **Run and Debug** tab.

→ On the VS Code, in the **DEBUG CONSOLE**, you should see the output of the program in blue and debug messages in yellow. Once the breakpoint is hit use **`F11`** to step into **`F10`** to step over, **`F5`** to continue, **`Shift+F5`** to Stop, & so on … you know what to do.

Revision overview list

Revision number	Description changes
0.1	Initial

Vekatech Ltd.

63, Nestor Abadzhiev st.
4023 Plovdiv
Bulgaria
Tel.: +359 (0) 32 262362
info@vekatech.com

www.vekatech.com