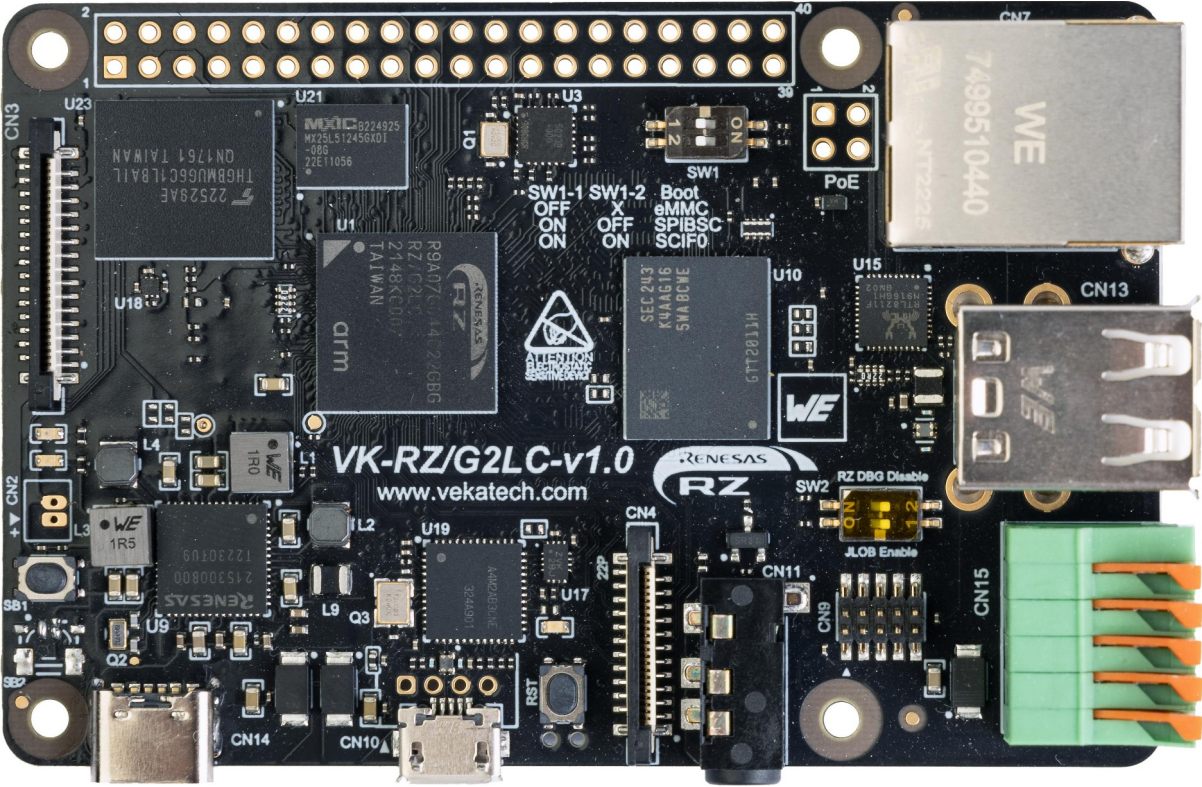


VK-RZ/G2LC How To



VK-RZ/G2LC v1.0 Board



How To manual

Content:

1. INTRODUCTION.....	3
1.1 CONNECTORS.....	3
1.2 SIGNALS.....	4
1.3 3D VIEW.....	5
1.4 DIMENSIONS.....	5
2. POWER UP.....	6
3. INSTALL U-BOOT (V2021.10).....	6
3.1 INTO SPI FLASH.....	6
3.2 INTO EMMC SSD.....	6
4. BOOT LOGIC.....	7
5. INSTALL LINUX (KERNEL V5.10.184).....	8
5.1 INTO SD CARD → DEBIAN V12.4 (BOOKWORM).....	8
5.2 INTO EMMC SSD → YOCTO V3.1.26 (DUNFELL).....	9
6. LAUNCH THE INSTALLED LINUX IMAGES.....	10
7. APPLY OVERLAYS.....	11
8. USE VARIOUS PERIPHERY IN LINUX.....	12
8.1 AUDIO.....	12
8.2 MIPI DSI DISPLAY.....	13
8.3 MIPI CSI CAMERA.....	16
8.4 ETHERNET.....	18
8.5 USB.....	19
8.6 GPIO.....	21
8.7 PWM.....	21
8.8 SPI.....	22
8.9 I2C.....	23
8.10 CAN.....	25
8.11 UART.....	26
8.12 RS485.....	26
9. USING .NET IN LINUX.....	27

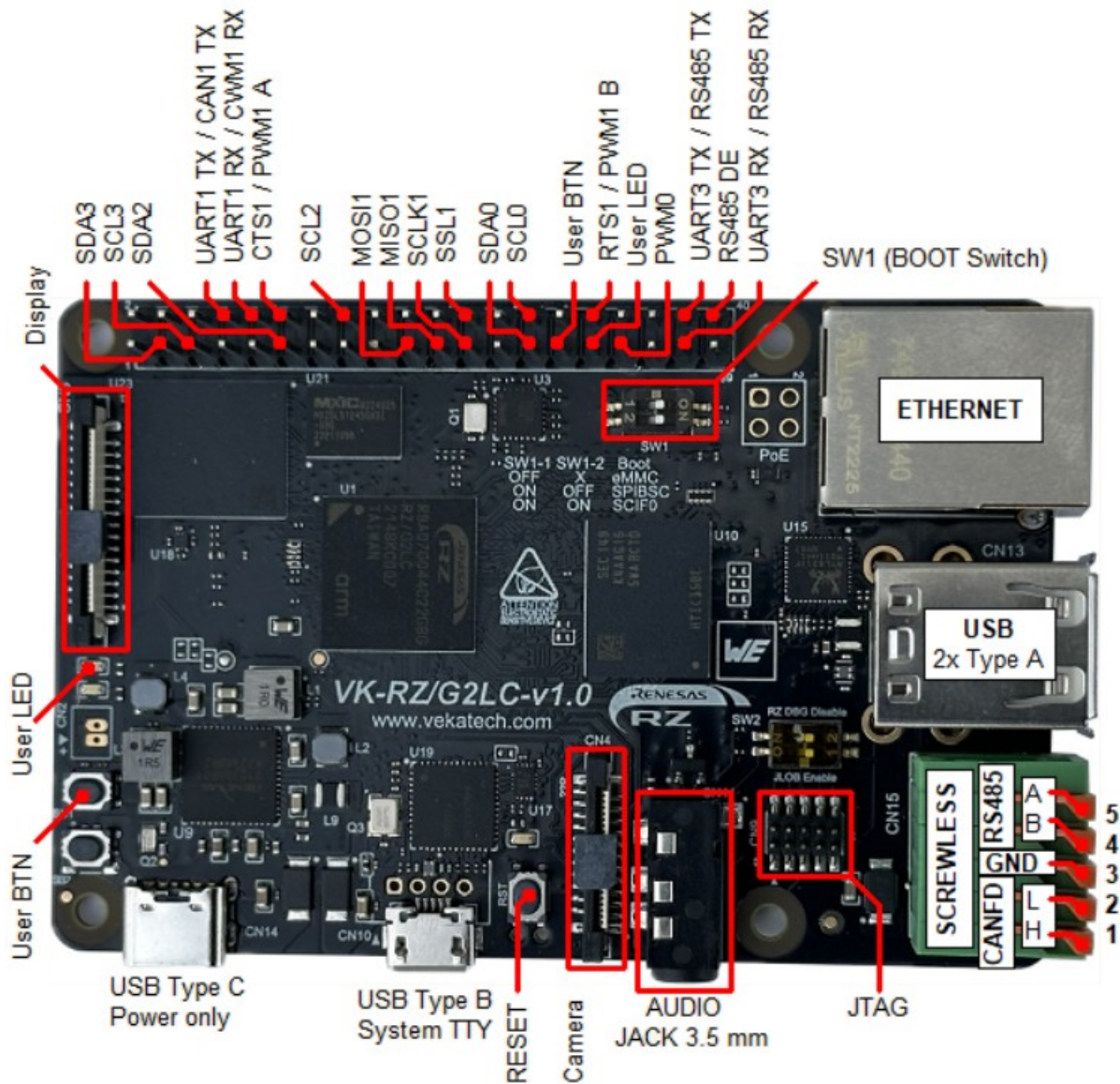


How To manual

1. Introduction

[VK-RZ/G2LC](#) is industrial oriented board, compatible with Raspberry Pi 4 shields. It is based on [Renesas R9A07G044C22GBG](#), **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

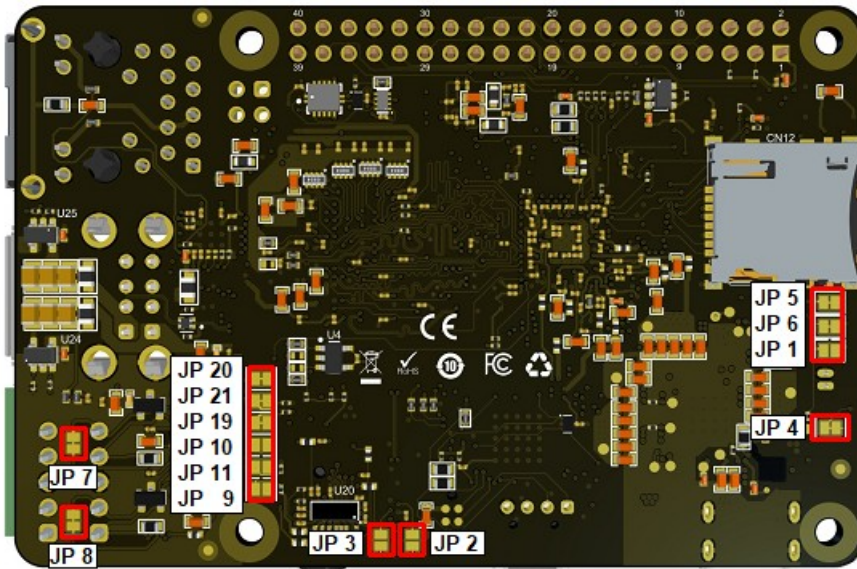


Connectors & Signals



How To manual

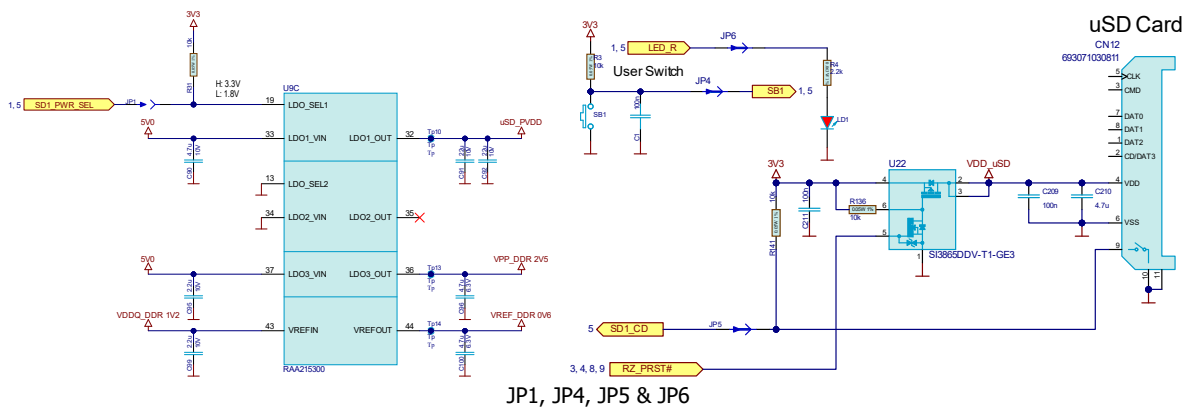
1.2 Signals



Joint Solderpads

Some of the pads below are connected by default other are disconnected. You can change them to suit your needs:

- **JP 1** (open) → Connects **PMIC's μSD PWR EN** to RZ MCU & 40 pin Header.
- **JP 2** (open) → Connects **Camera's RST** to RZ MCU & 40 pin Header.
- **JP 3** (open) → Connects **Camera's GPIO** to RZ MCU & 40 pin Header.
- **JP 4** (short) → Connects onboard **User BTN** to RZ MCU & 40 pin Header.
- **JP 5** (short) → Connects **μSD's Card Detect** signal to RZ MCU.
- **JP 6** (short) → Connects onboard **User LED** to RZ MCU & 40 pin Header.



JP 2 & JP 3 are displayed in **8.3**, and the rest below (**JP 7 - JP 11, JP 19 - JP 21**) are in **8.10**.



How To manual

- **JP 7** (short) → Connects **120Ω termination** resistor between RS485's **A** & **B** lines.
- **JP 8** (short) → Connects **120Ω termination** resistor between CAN's **H** & **L** lines.
- **JP 9** (open) → Connects **CAN Transceiver's D** signal to RZ MCU & 40 pin Header.
- **JP 10** (open) → Connects **CAN Transceiver's R** signal to RZ MCU & 40 pin Header.
- **JP 11** (open) → Connects **CAN Transceiver's STBY** signal to RZ MCU & 40 pin Header.
- **JP 19** (short) → Connects **RS485 Transceiver's R** signal to RZ MCU & 40 pin Header.
- **JP 20** (short) → Connects **RS485 Transceiver's D** signal to RZ MCU & 40 pin Header.
- **JP 21** (short) → Connects **RS485 Transceiver's DE** signal to RZ MCU & 40 pin Header.

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



How To manual

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 (Power only, intelligent USB adapters do not work with **v1.0** of VK-RZ/G2LC)
- 40pin Extension header

In idle state (without plugged periphery) the board consumes ~ **300 mA**. The consumption however, can jump up to ~ **1,1 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 for **VK-RZ/V2L** board, but let's stick to the precompiled binaries for now:

- Connect **VK-RZ/G2LC** to the PC (through **USB Type B micro**) & see what COM port is assigned by the OS in the Device Manager.
- Unzip **vkzrg2lc-program-utility.zip**.
- Edit **cfgcom_vkzrg2lc.ini** file, altering the **COM** number to match with **VK-RZ/G2LC**.
- Set the **SW1** switch on **VK-RZ/G2LC** (**1:ON | 2:ON**) so it can boot from **SCIF0**.

3.1 *into SPI Flash*

- Execute **vkzrg2lc_bootloader-sf.bat**.
- Press **reset** button on the **VK-RZ/G2LC**

3.2 *into eMMC SSD*

- Execute **vkzrg2lc_bootloader-emmc.bat**.
- Press **reset** button on the **VK-RZ/G2LC**



How To manual

After download is ready, switch **SW1** back to boot from **SPIBSC** (1:ON | 2:OFF) in case you executed 3.1 or **eMMC** (1:OFF | 2:OFF) in case of 3.2. Open VK-RZ/G2LC'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.

```
COM75 - PuTTY
NOTICE: BL2: v2.9(release):c314a39-dirty
NOTICE: BL2: Built : 14:49:18, Sep 19 2023
NOTICE: BL2: Booting BL31
NOTICE: BL31: v2.9(release):c314a39-dirty
NOTICE: BL31: Built : 14:49:18, Sep 19 2023

U-Boot 2021.10 (Sep 20 2023 - 02:21:30 +0000)

CPU:   Renesas Electronics CPU rev 1.0
Model: Vekatech vkrzg2lc
DRAM:  1.9 GiB
WDT:   watchdog@00000000012800800
WDT:   Started with servicing (60s timeout)
MMC:   sd@11c00000: 0, sd@11c10000: 1
Loading Environment from SPIFlash... SF: Detected mx25l51245g with page size 256
6 Bytes, erase size 4 KiB, total 64 MiB
OK
In:    serial@1004b800
Out:   serial@1004b800
Err:   serial@1004b800
U-boot WDT started!
Net:
Warning: ethernet@11c20000 (eth0) using random MAC address - ae:82:8e:b1:2b:1f
eth0: ethernet@11c20000
Hit any key to stop autoboot:  0
=> █
```

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: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:OFF**) 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, leads to booting from μSD. You can always change those parameters & boot from wherever you want.



How To manual

5. Install Linux (Kernel v5.10.184)

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-vkrzg2lc.img.xz**.
- Launch Rufus, for **Device** select µSD card drive, for **boot section** select desired Linux image file (in this case **debian-bookworm-vkrzg2lc.img**).
- Hit **START** & wait completion (Status **READY**), and eject µSD card from the PC.
- Plug µSD card into VK-RZ/G2LC's holder, set **SW1** to (1:ON | 2:OFF) & press reset.
- You should now be able to see login screen, use user: **vkrz** & password: **vkrzg2lc**.

```
COM75 - PuTTY
[ OK ] Reached target graphical.target - Graphical Interface.
Starting systemd-update-ut... Record Runlevel Change in UTMP...
[ OK ] Finished systemd-update-ut... - Record Runlevel Change in UTMP.
[ 18.438219] RTL8211F Gigabit Ethernet 11c20000.ethernet-ffffffff:00: attached PHY driver [RTL
i_bus:phy_addr=11c20000.ethernet-ffffffff:00, irq=141)

Debian GNU/Linux 12 vkrzg2lc ttySC0

vkrzg2lc login: [ 628.236756] rcar-du 10890000.display: vertical blanking timeout

vkrzg2lc login: vkrz
Password:
Linux vkrzg2lc 5.10.184-cip36-yocto-standard #1 SMP PREEMPT Tue Apr 5 23:00:00 UTC 2011 aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
vt220 80x24 -> 123x65
vkrz@vkrzg2lc:~$
```

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:



How To manual

```
sudo growpart /dev/mmcblk1 2 && sudo resize2fs /dev/mmcblk1p2.
```

➤ 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`.

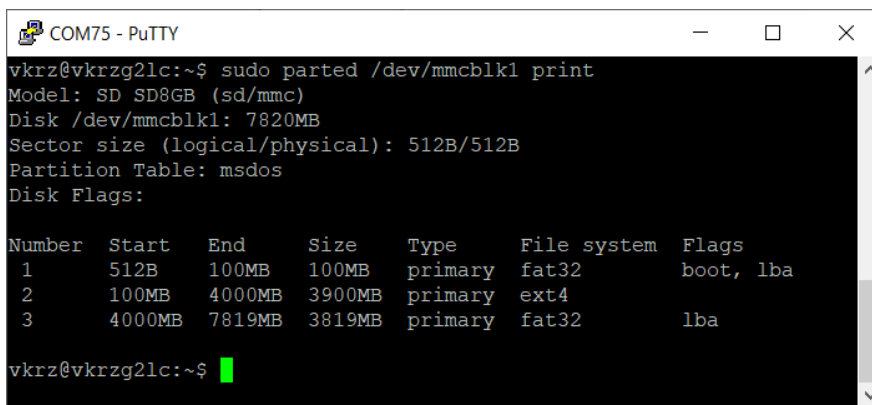
Extend the root partition to **N** GB size: `sudo parted /dev/mmcblk1 resizepart 2 NG`.

Fix filesystem block size: `sudo resize2fs /dev/mmcblk1p2`.

Make new partition: `sudo parted /dev/mmcblk1 "mkpart primary fat32 NG -1"`.

Format the new partition: `sudo mkfs.vfat -F 32 /dev/mmcblk1p3`.

To verify how the μ SD card look like, type that: `sudo parted /dev/mmcblk1 print`.



```
COM75 - PuTTY
vkrz@vkrzg21c:~$ sudo parted /dev/mmcblk1 print
Model: SD SD8GB (sd/mmc)
Disk /dev/mmcblk1: 7820MB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type    File system  Flags
  1      512B   100MB   100MB   primary fat32        boot, lba
  2      100MB  4000MB  3900MB  primary ext4
  3      4000MB 7819MB  3819MB  primary fat32        lba

vkrz@vkrzg21c:~$ █
```

SD card formatting

With this the μ SD card is considered ready and the newly created partition can help uploading linux image into the onboard **eMMC SSD**.

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

Get the **μ SD** card prepared in 4.1 and plug it into the PC.

- Download [.gz](#) and [.bmap](#) files of the desired Linux image (in this case Yocto) and place them in the latest newly created partition of the μ SD card (i.e. partition 3)
- Eject μ SD card from the PC and plug it into VK-RZ/G2LC's holder.
- Make sure **SW1** is set to boot from **SPI** (1:ON | 2:OFF) & press reset.
- Login to Debian with user: **vkrz** & password: **vkrzg21c**.
- Install tool to transfer the image: `sudo apt-get install bmap-tools`.
- Make partition 3 accessible: `sudo mount /dev/mmcblk1p3 /mnt`.
- Transfer the image in to eMMC: `sudo`

```
bmaptool copy /mnt/core-image-weston-vkrzg21c.wic.gz /dev/mmcblk0.
```

Or alternatively you can unzip the image, & then transfer it with the well known dd command:

```
dd if=core-image-weston-vkrzg21c.wic of=/dev/mmcblk0 bs=1M iflag=fullblock oflag=direct conv=fsync status=progress.
```



How To manual

- If you want to extend the root partition to use the full capacity of the eMMC:
`sudo growpart /dev/mmcblk0 2 && sudo resize2fs /dev/mmcblk0p2.`
- If you want to extend the root partition to a fixed size **N** (where N is number in GB)
`sudo parted /dev/mmcblk0 resizepart 2 NG.`
Fix filesystem block size: `sudo resize2fs /dev/mmcblk0p2.`
You can make additional partitions & format them if you like:
`sudo parted /dev/mmcblk0 "mkpart primary ext4 NG MG".`
`sudo mkfs.ext4 /dev/mmcblk0p3.`
- Discard partition 3: `sudo umount /mnt.`
- Make sure **SW1** is set to boot from **eMMC** (1:OFF | 2:OFF) & press reset.
- You should now be able to see login screen of the Yocto, use `root` to login.

```
COM75 - PuTTY
ted Hostname Service.
[ OK ] Started User Manager for UID 0.
[ OK ] Started Session c1 of user root.
[ 12.485001] ravb 11c20000.ethernet eth0: Link is Up - 1Gbps/Full - flow control off
[ 12.492701] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 12.539768] 8021q: 802.1Q VLAN Support v1.8

Poky (Yocto Project Reference Distro) 3.1.26 vkrzg2lc ttySC0

BSP: RZG2LC/VK-RZ/G2LC-v1.0/3.0.5
LSI: RZG2LC
Version: 3.0.5
vkrzg2lc login: [ 44.764636] audit: type=1334 audit(1707914991.072:13): prog-id=10 op=UNLOAD
[ 44.771664] audit: type=1334 audit(1707914991.072:14): prog-id=9 op=UNLOAD

vkrzg2lc 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 Linux from μ SD: set the switch to **SPIBSC (1:ON | 2:OFF)**.
- To boot Linux from eMMC: set the switch to **eMMC (1:OFF | 2:OFF)**.
- To boot Linux from Ethernet: set the switch to **SPIBSC** and make sure μ SD slot is Empty.
When U-boot can't find SD card, it will try to boot from its **serverip**, **tftpdnir** & **netrootfs** environment parameters. (serverip is the address, where U-boot will look for **NFS** & **TFTP** servers, tftpdnir shows where the boot directory is on the server and netrootfs → where root directory is on the server) Same server can be used to boot multiple boards, every board can have its own boot and root directories and that's why tftpdnir and netrootfs parameters should be filled with the correct paths on the server.



How To manual

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-audio.dtbo
- vkrz-cm33.dtbo
- vkrz-csi-imx219.dtbo
- vkrz-dsi-av_disp2.dtbo / vkrz-dsi-vklcd07.dtbo / vkrz-dsi-vklcd-ee0700.dtbo
- vkrz-exp-i2c2.dtbo
- vkrz-exp-i2c3.dtbo
- vkrz-exp-pwm0.dtbo / vkrz-exp-scfi1_rts_cts.dtbo
- vkrz-exp-pwm1.dtbo
- vkrz-can.dtbo / vkrz-exp-scfi1.dtbo / vkrz-exp-scfi1_rts_cts.dtbo
- vkrz-exp-scif3.dtbo
- vkrz-exp-spi1.dtbo
- vkrz-udma.dtbo



How To manual

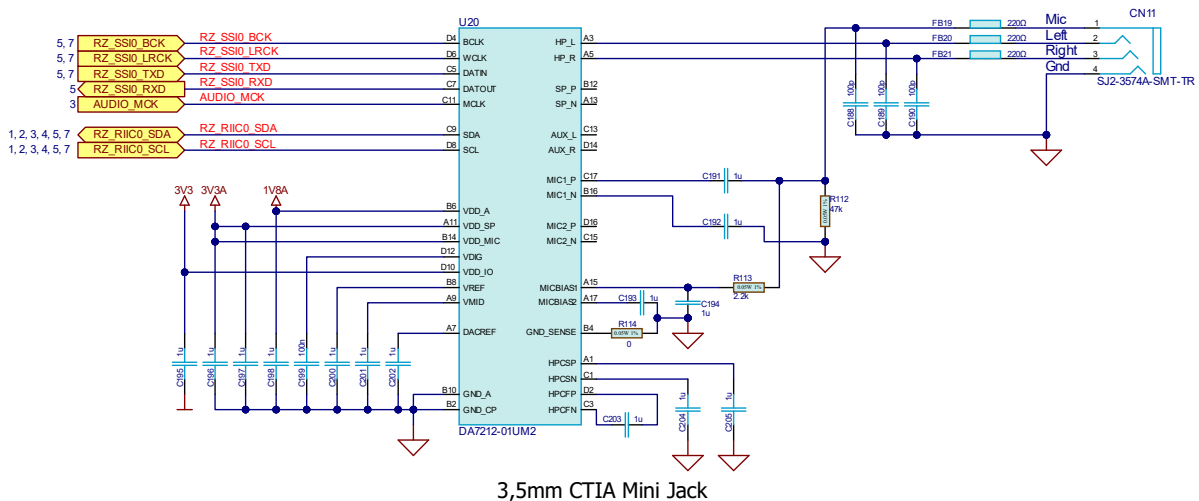
8. Use various periphery in Linux

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

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



- 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**.

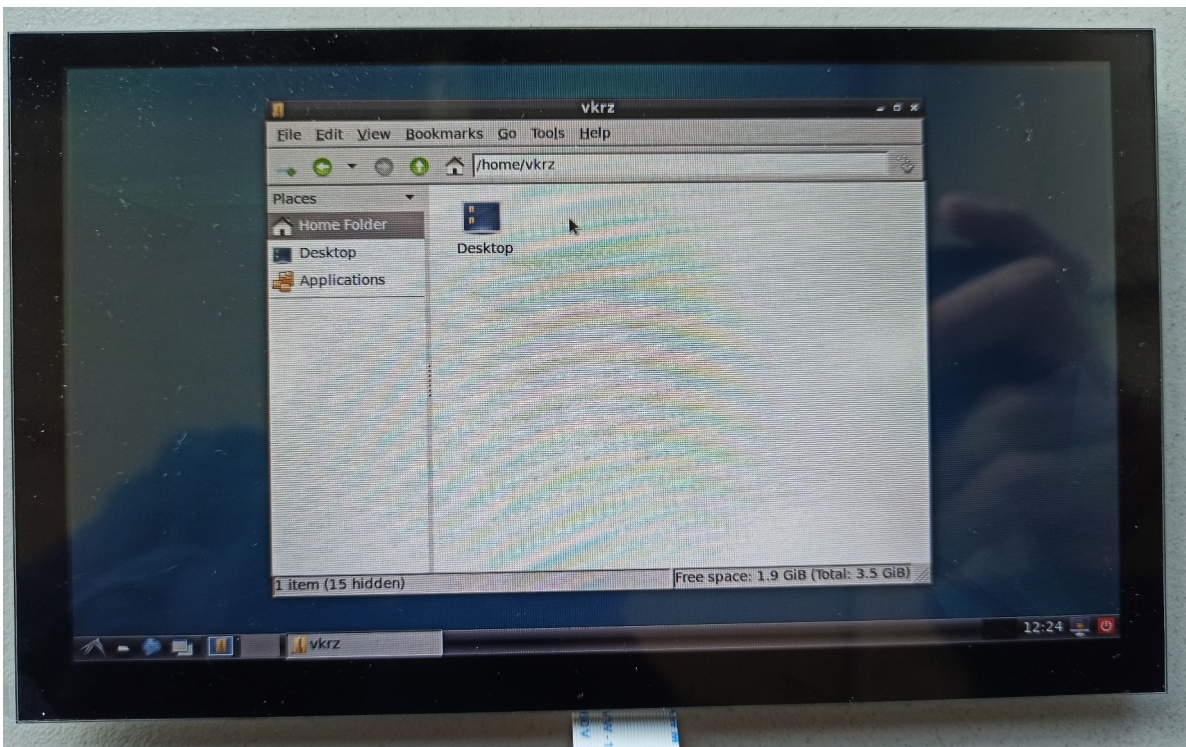


How To manual



vkcz-dsi-vklcd-ee0700 display FPC cable orientation

➤ `fdt_extra_overlays=vkcz-dsi-vklcd07.dtbo.`

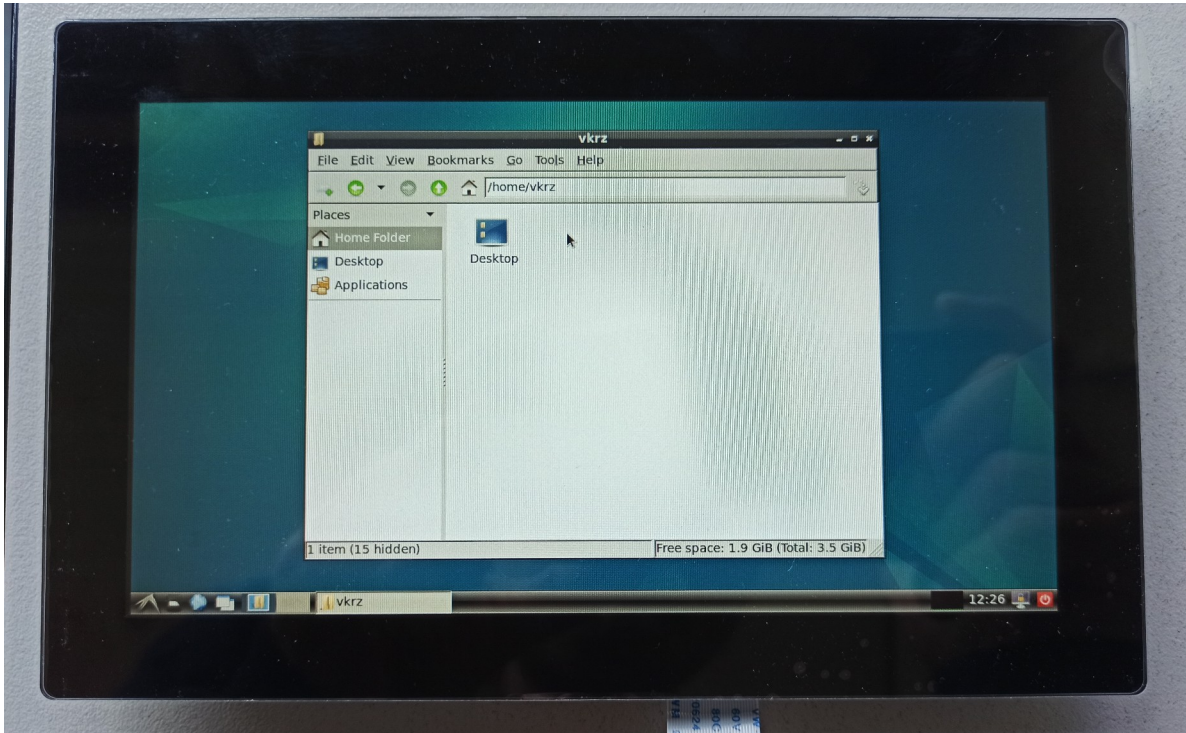


FORMIKE KWH070KQ40-C08 (1024 x 600)



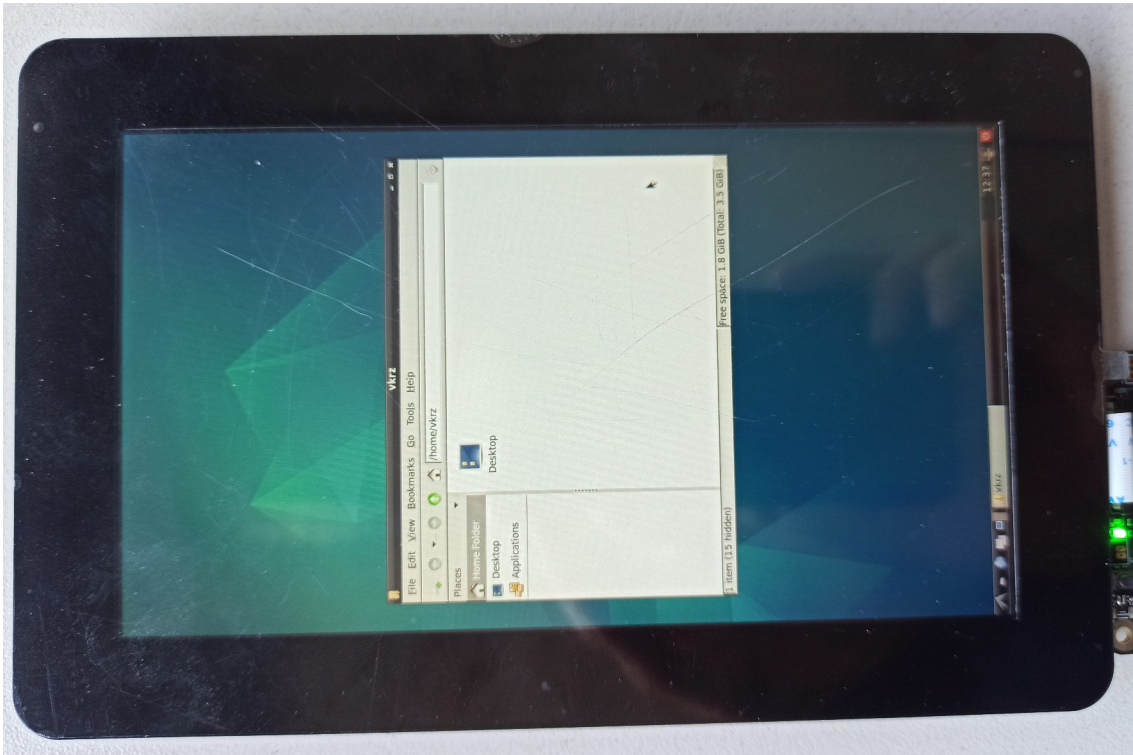
How To manual

- `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)

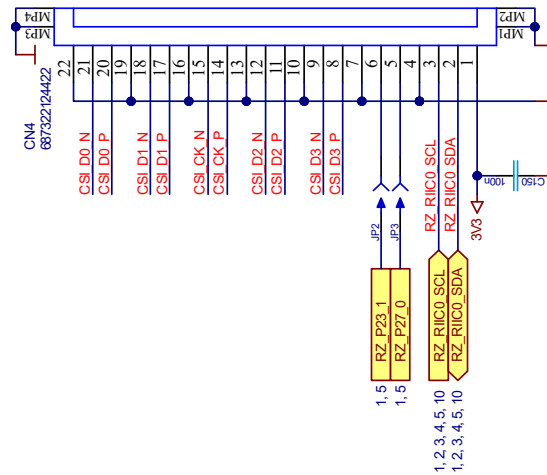


How To manual

8.3 MIPI CSI Camera

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

fdt_extra_overlays=vkcrz-csi-imx219.dtbo vkcrz-dsi-vklcd-ee0700.dtbo



22 pin FPC MIPI CSI Connector

Make sure you make connection on **JP2** solderpad, so **RST** signal of the camera to get fixed potential and not be floating.

- 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]"
```

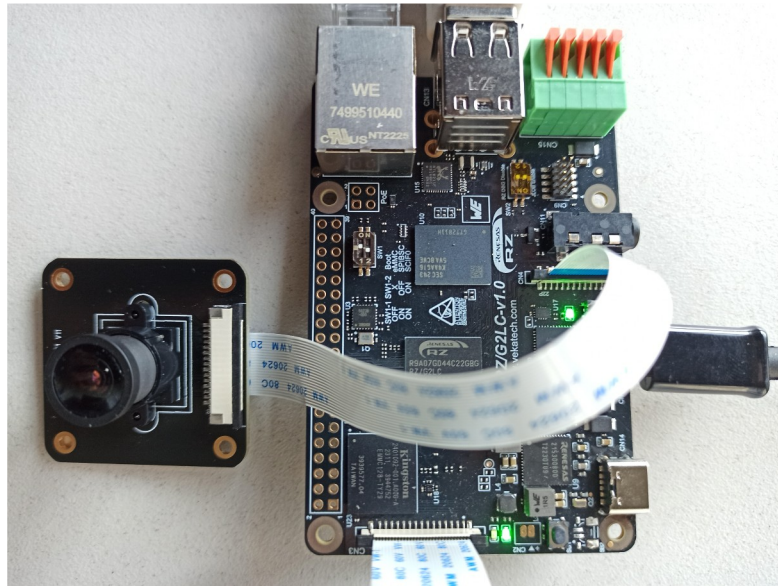
```
media-ctl -d /dev/media0 -V ""imx219 0-0010':0 [fmt:UYVY8_2X8/$imx219_res field:none]"
```

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

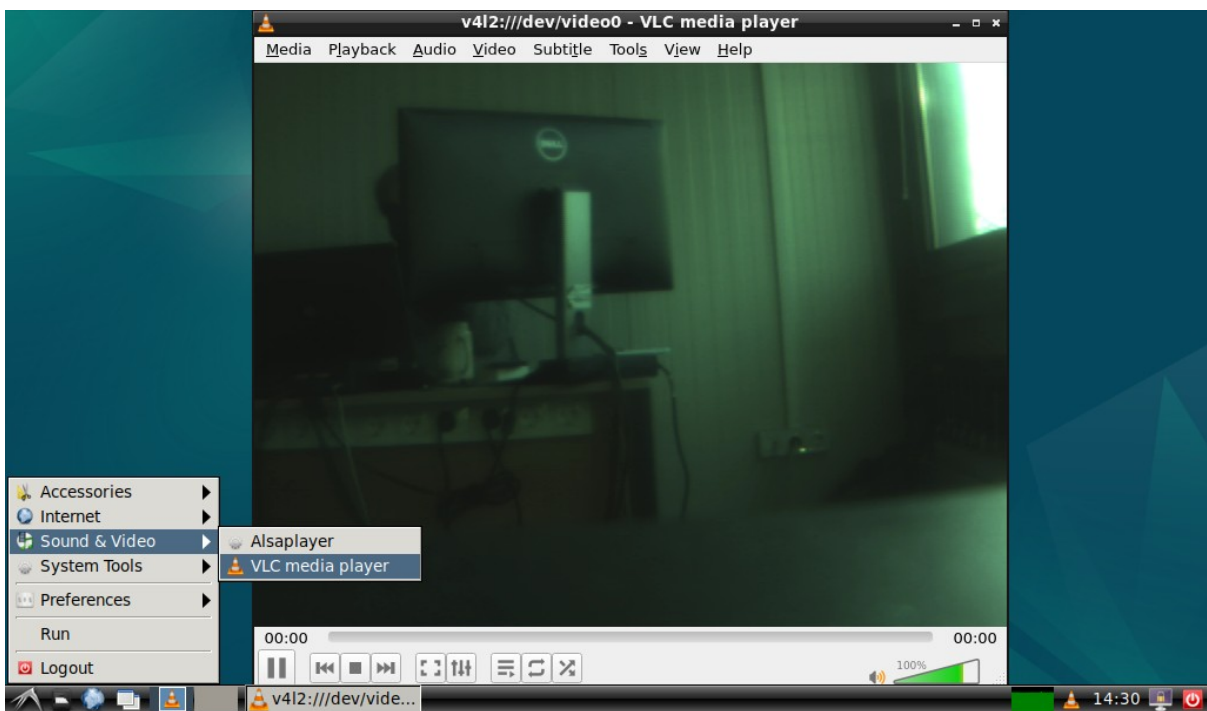
- When you are happy with settings in the script, execute it: `~/v4l2-init.sh`.
- Open VLC through the GUI: go to **Start/Sound & Video/VLC media player**.
- Go to **Media** → **Open Capture Device ...** → **Capture Device** Tab
for **Video device name** select **/dev/video0** and press **Play**.
- You should now be able to see the video input stream.



How To manual



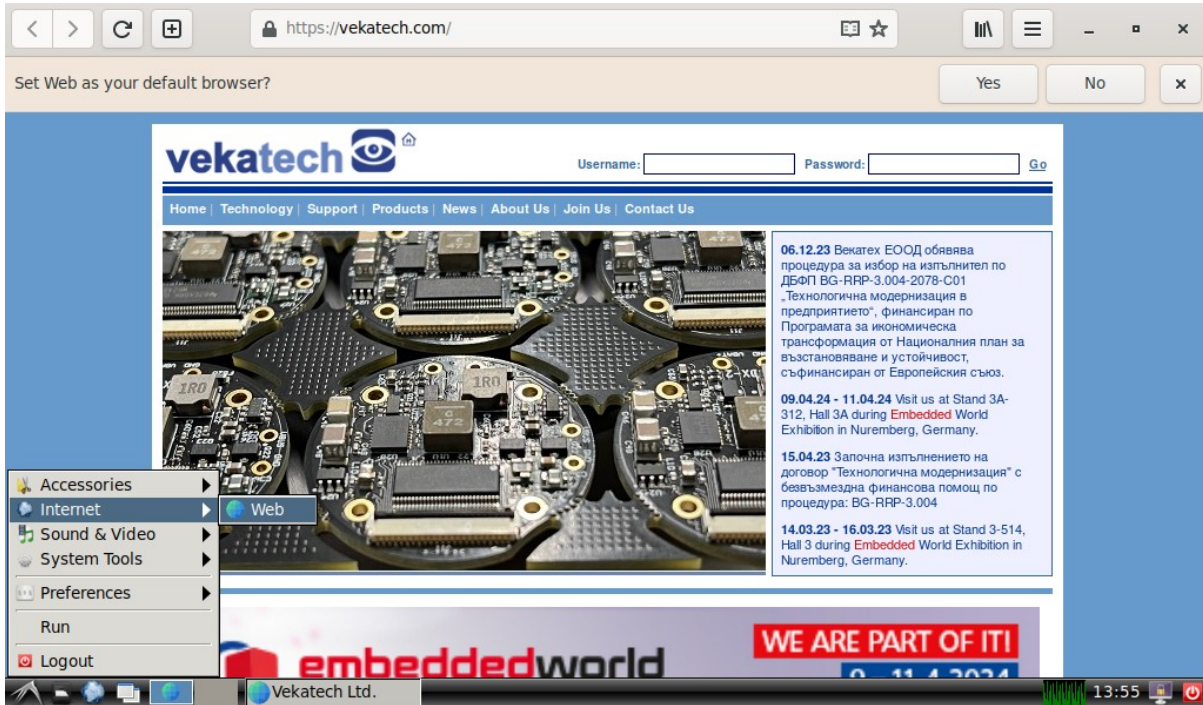
vkrcz-csi-imx219 camera FPC cable orientation



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



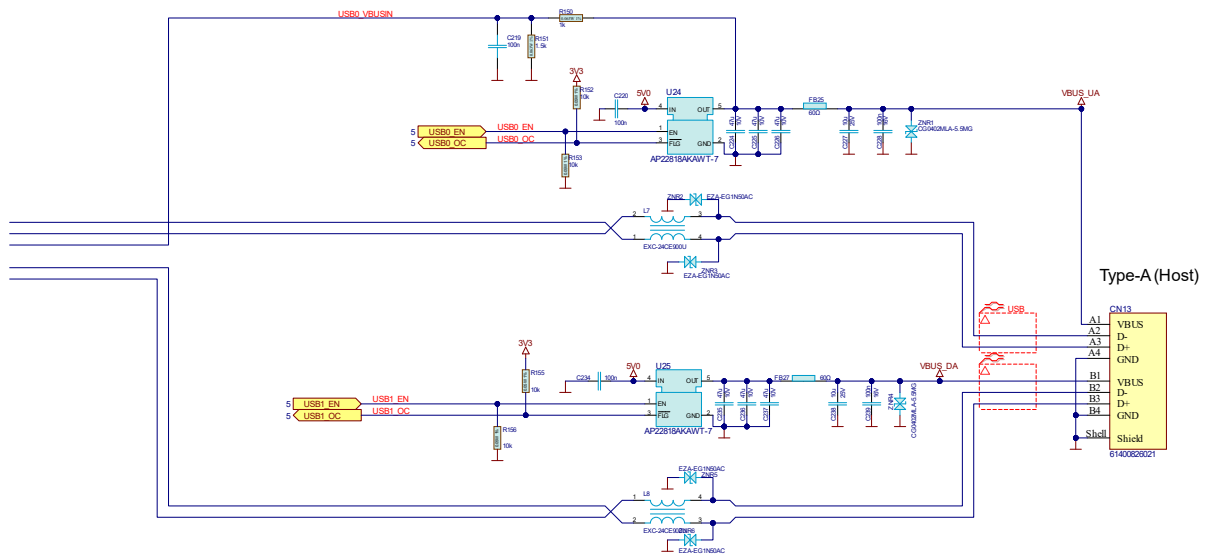
How To manual



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.

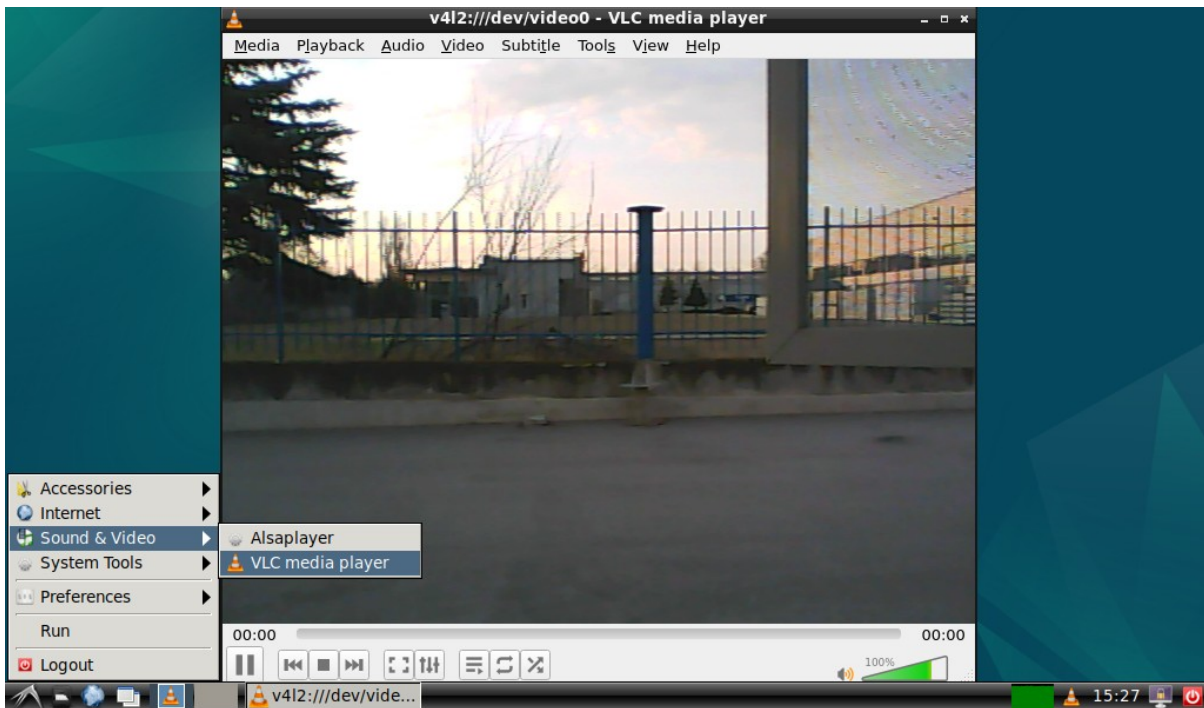


USB Type A connectors



How To manual

- 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.
- 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 Device** Tab for **Video device name** select **/dev/video0** and press **Play**.
 - You should now be able to see the USB video stream.



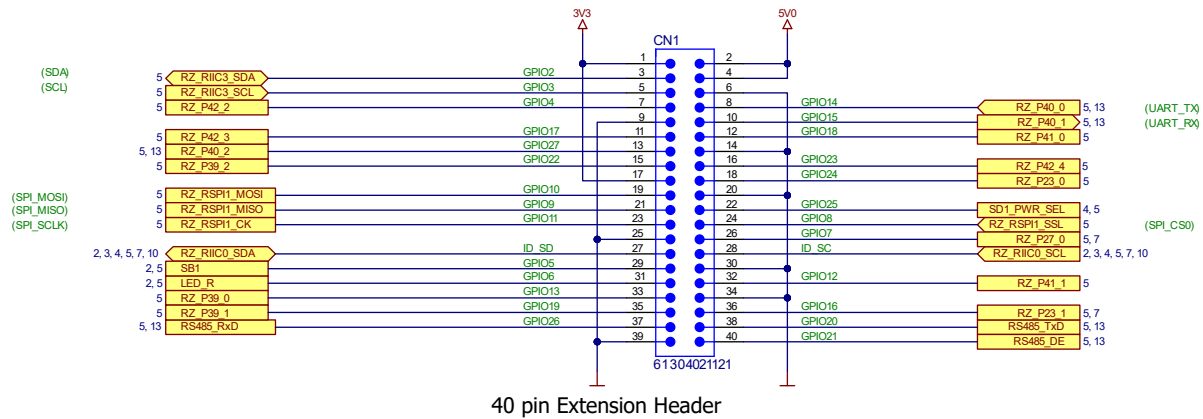
Logitech C170



How To manual

8.6 GPIO

GPIOs are enabled by default and do not need overlay.



Pay attention that **RZ_P23_0**, **RZ_P23_1** & **RZ_P27_0** are on the **Low Voltage Power Domain** (i.e. their logical **HI** levels are **1,8V**) and take that in consideration when using these GPIOs.

- 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: $(\text{port} \times 8) + \text{pin} + 120$, if **P23_0** is desired to be controlled, the internal number is: $(23 \times 8) + 0 + 120 = 304$. The way to tell the system is with these commands: `sudo su`
`echo 304 > /sys/class/gpio/export.`
- If you want to get the value of **P23_0** pin, you can type this:
`echo in > /sys/class/gpio/P23_0/direction.`
`cat /sys/class/gpio/P23_0/value.`
- If you want to set **P23_0** pin to **HI**, you can type this:
`echo out > /sys/class/gpio/P23_0/direction.`
`echo 1 > /sys/class/gpio/P23_0/value.`
- If you don't want to use the pin anymore you can release the resource:
`echo 304 > /sys/class/gpio/unexport.`

8.7 PWM

Make sure you are applied **vkcrz-exp-(pwm0/pwm1).dtbo** overlay in **/boot/uEnv.txt**.

```
fdt_extra_overlays=vkcrz-exp-pwm0.dtbo vkcrz-exp-pwm1.dtbo.
```

Make sure if **vkcrz-exp-scfi1_rts_cts.dtbo** is also **excluded** from **fdt_extra_overlays**, (in case **vkcrz-exp-pwm0.dtbo** is used), because **uart1** flow control pins use the same pins as **PWM0**.



How To manual

PWM[0] pins go to Pin12 (**GTIOC7A**) & Pin32 (**GTIOC7B**) of 40pin extension connector.

PWM[1] pin go to Pin33 (**MTIOC4C**) 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 3 `pwmchip` folders: **pwmchip0**, **pwmchip1**, **pwmchip2**. Usually this is the case when some of the displays & both pwm overlays are used (the current example is exactly that case), so the mapping is as follows:

Display **brightness** → `pwmchip0` | PWM[0] → `pwmchip1`, | PWM[1] → `pwmchip2`.

Let'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 **MTIOC4C** should be available as **pwm0**, and can be configured:
period in μ s `echo 1000000 > /sys/class/pwm/pwmchip2/pwm0/period.`
duty_cycle in μ s `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 **vkz-exp-spi1.dtbo** overlay in `/boot/uEnv.txt`.

```
fdt_extra_overlays=vkrz-exp-spi1.dtbo vkz-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.



How To manual

- 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 **vk rz-exp-(i2c2/i2c3).dtbo** or both overlay in **/boot/uEnv.txt**.

```
fdt_extra_overlays=vkrz-exp-i2c2.dtbo vkrz-exp-i2c3.dtbo.
```

I²C[3] pins go to Pin3 (**SDA**) & Pin5 (**SCL**) of 40pin extension connector.

I²C[2] pins go to Pin11 (**SDA**) & Pin16 (**SCL**) of 40pin extension connector.

- Install I²C software: `sudo apt-get install i2c-tools.`
- Now we can see all I²C devices on the system bus 0: `sudo i2cdetect -y -a -r 0.`

```
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- UU -- -- -- -- -- -- -- -- 1a -- -- --
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 0 0x51 0x00 b.`
- Write byte **0x55** to addr **0xF0** of E²PROM: `sudo i2cset -y 0 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.`



How To manual

```
import sys
import smbus

bus = 0

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:  --  --  --  --  --  --  --  --  --  --  1a  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  51 52 53 54 55 56 57  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```



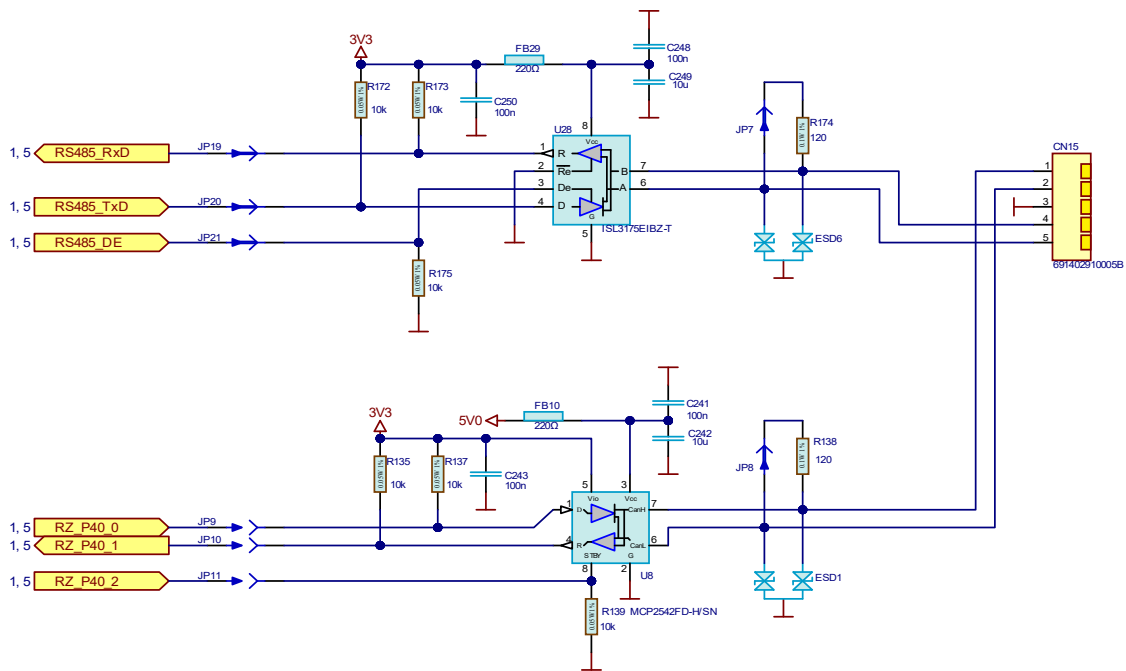

How To manual

8.10 CAN

Make sure you are applied **vkrrz-can.dtbo** overlay in **/boot/uEnv.txt**.

```
fdt_extra_overlays=vkrrz-can.dtbo vkrrz-dsi-vklcd-ee0700.dtbo
```

Make sure **vkrrz-exp-scfil.dtbo** and/or **vkrrz-exp-scfil_rts_cts.dtbo** are **excluded**, from **fdt_extra_overlays**, because they use same pins as CAN1.



CAN & RS485 screwless connector.

CAN[1] pins go simultaneously to:

- Pin8 (**TX**), Pin10 (**RX**), Pin13 (**EN**), of 40pin extension connector.
- Through onboard **CANFD** Transceiver (Pin1 (**CANH**) & Pin2 (**CANL**) & Pin3 (**GND**) of screwless connector.) **Make sure you make connections on J9, J10 & J11 solderpads, so MCU and Transceiver can talk with each other (there is no connection by default and signals go only to the 40pin ext. header).**

Use the CAN module:

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



How To manual

- Here a Loopback test is possible if only a special mode exist in the can module, but this is not the case here, so you will need a second can device to talk to.
- If you want to receive messages type: `candump can0`.
- If you want to send message type: `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 `vk rz-exp-scfi1.dtbo` / `vk rz-exp-scfi1_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=vk rz-exp-scfi1.dtbo vk rz-dsi-vklcd-ee0700.dtbo
```

Make sure `vk rz-can.dtbo` is **excluded**, from `fdt_extra_overlays`, because it uses same pins as UART1. Check if `vk rz-ext-pwm0.dtbo` is also **excluded** from `fdt_extra_overlays`, (in case `vk rz-exp-scfi1_rts_cts.dtbo` is used), because `pwm0` uses the same pins as flow control pins of UART1.

UART[1] pins go to Pin8 (**TX**), Pin10 (**RX**), Pin12 (**CTS**) Pin32 (**RTS**) of 40pin extension connector.

UART[3] pins go to Pin37 (**RX**), Pin38 (**TX**) 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/ttySC1` or `screen -f /dev/ttySC1` (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**)

8.12 RS485

Make sure you are applied `vk rz-exp-scfi3.dtbo` overlay in `/boot/uEnv.txt`.

```
fdt_extra_overlays=vk rz-exp-scfi3.dtbo vk rz-dsi-vklcd-ee0700.dtbo
```

UART[3] pins go simultaneously to:

- Pin37 (**RX**), Pin38 (**TX**), Pin40 (**DE**) of 40pin extension connector.
- Through onboard **RS485** Transceiver (Pin3 (**GND**), Pin4 (**B**) & Pin5 (**A**) of screwless connector.)



How To manual

Set RS485 module to 19200 8 N 1 → `stty -F /dev/ttySC3 19200 cs8 -cstopb -parenb.`

- Init DE pin. RS485 is actually UART with **Data Enable** pin, which goes **Hi** before **TX** transaction and goes back to **Low** after the transaction, so you need to manually control that pin, this means the internal pin should be **P5_1**, which is $(5 \times 8) + 1 + 120 = 161$.
`sudo su && echo 161 > /sys/class/gpio/export.`
`echo out > /sys/class/gpio/P5_1/direction.`
`echo 0 > /sys/class/gpio/P5_1/value.`
- TX some data: `echo 1 > /sys/class/gpio/P5_1/value.`
`echo "Helow world" > /dev/ttySC3.`
Wait data to be fully transmited & exec. `echo 0 > /sys/class/gpio/P5_1/value.`
- RX some data: `cat /dev/ttySC3.`
Note that the receiving terminal can have buffering enabled, so the data read might not be displayed immediatly. The buffer is flushed once enough data is entered, or when enough newlines are encountered.
- Release DE pin resource: `echo 161 > /sys/class/gpio/unexport.`

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 --install-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 I²C devices on the bus 0. For this you are going to need **I²C 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**.



How To manual

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

class Program
{
    static void Main(string[] args)
    {
        const int i2cBusId = 0; // 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:



How To manual

```
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](#) **dlls** on the target device. However when your project uses a board specific periphery, which is only available on the board (let's say SPI, I²C, 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: **Visual Studio** & **VS Code**. For the last one, you will also going to need **Remote - SSH** & **C# Dev Kit** extensions.

→ Install [.NET SDK](#) on the **development 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.`



How To manual

→ 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 `vk rz` & `vk rzg2lc`, 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 vk rz@<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 vk rz@<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 vk rz@<bord's IP> -pw <vk rz's password i.e. vk rzg2lc>.
```

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`.

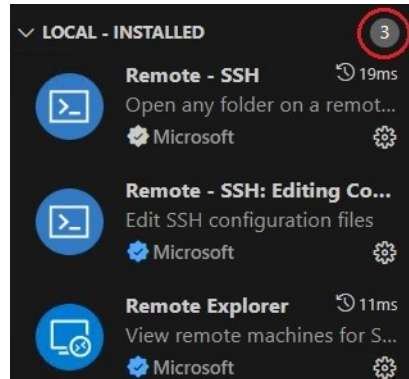
→ On the dev PC, launch VS Code & install the extensions mentioned earlier. Go to:

Extensions tab (`Ctrl+Shift+X`) and type **Remote-SSH** in the Search Extensions bar,

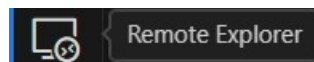


How To manual

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.



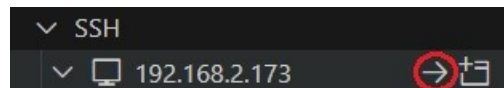
→ On the VS Code, go to that new tab (**Remote Explorer**) hit New Remote (the + sign)



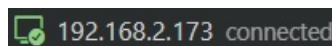
For *Enter SSH Connection command*, type this: `ssh vkrz@<board's IP>`. Keep in mind that only `ssh` works here. For now it's only it, fully supported by the **Remote Explorer**.

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.

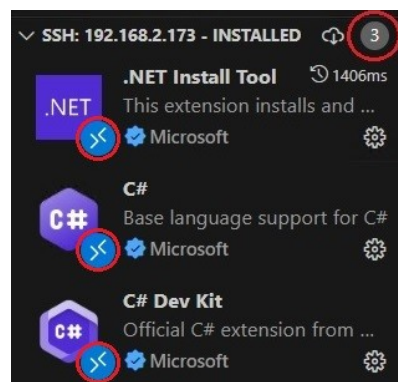
→ On the VS Code, go to **Remote Explorer** tab & hit Connect in Current Window... (→)



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



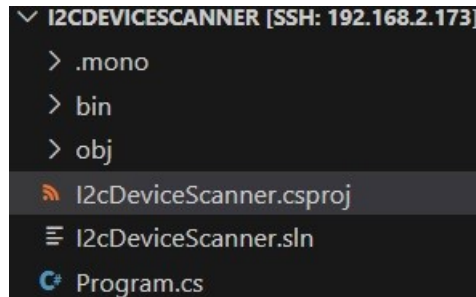
→ On the VS Code, go to **Extensions** tab again (`Ctrl+Shift+X`) & type **C# Dev Kit** in the Search Extensions bar, select it & install it. Extensions **C#** and **.NET Install Tool** comes as companions to it, so you end up with all 3 installed **remotely** (on the board).





How To manual

→ 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.



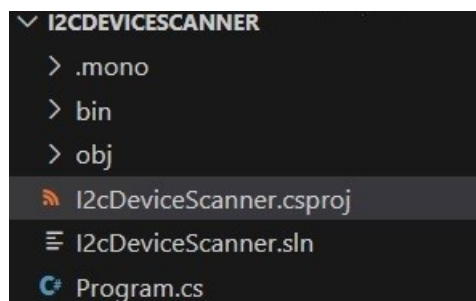
That is useful when developing project entirely on the target board, which is slow and excruciating process. The point is only debug to be on the target board, so open the cmd.

→ 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": [
    {
```




How To manual

```
"name": ".NET Remote Launch - Framework-dependent [SSH-key]",
"type": "coreclr",
"request": "launch",
"program": "~/.NET/dotnet",
"args": ["bin/Debug/net6.0/I2cDeviceScanner.dll"],
"cwd": "~/.NET/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",
    "vkrzg21c"
  ],
  "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.



How To manual

Revision overview list

Revision number	Description changes
0.1	Initial
0.2	Added chapter: Using .NET

Vekatech Ltd.

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

www.vekatech.com