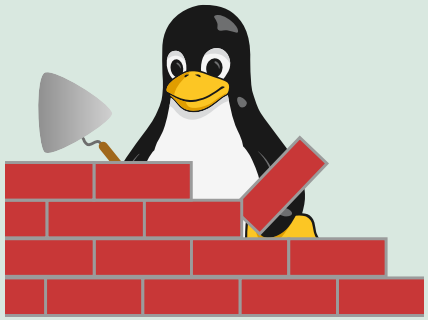
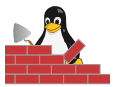
LINES
Linux for Embedded Systems
Lecture 1 - How to communicate with external devices connected to the embedded system?



The Tux image by lewing@isc.tamu.edu Larry Ewing and The GIMP

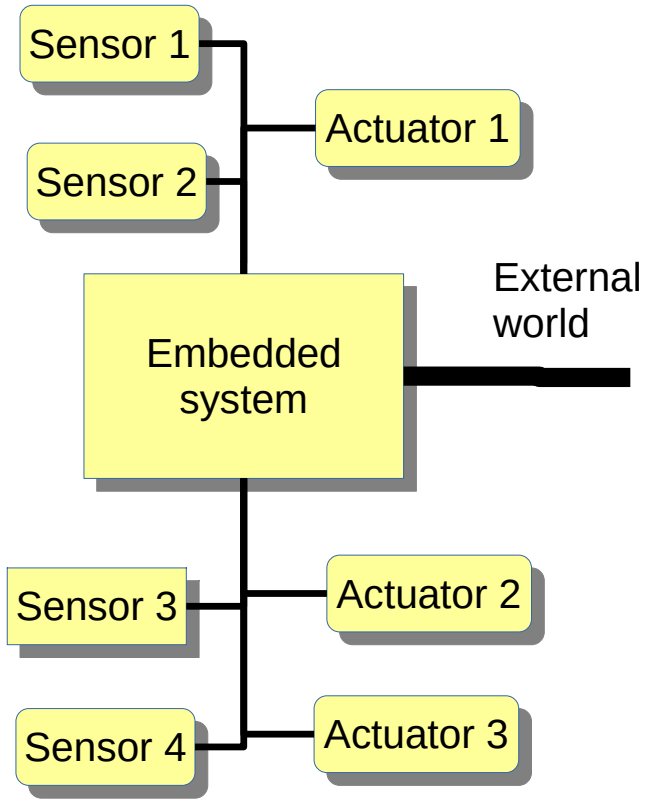
Wojciech M. Zabołotny, Ph.D., D.Sc.

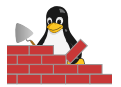
Faculty of Electronics and Information Technology WUT, room. 225,
wzab@ise.pw.edu.pl (please add [LINES] in the subject)



Communication with external devices

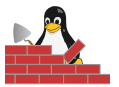
- The embedded system is usually used to control certain device.
- Therefore, it must be equipped with measurement and control functions.
- It requires communication with sensors and actuators connected to dedicated interfaces of the processor.





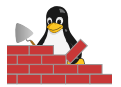
Interfaces available for connecting the external devices

- The processor bus (e.g., EBI in ARMs) – very rich interfacing possibilities, but difficult from the hardware point of view.
  - most single board computers do not provide access to the EBI pins on connectors
  - In the newest System on Chip (SoC) solutions, the processor bus – AXI or Avalon may be efficiently used
- USB – available like in standard PC computers
- UART – the serial port
- Specialized interfaces for simple peripheral devices: SPI, I2C, I2S
- Simple directly controlled pins – GPIO (General Purpose I/O)...



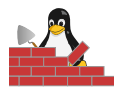
Usage of USB

- Similar to “standard” Linux
- There are two possibilities:
  - Use of the dedicated kernel device driver (may be our own?)
    - Many USB devices implement special predefined “USB Device Classes” they are then controlled by a common class driver.
  - Use of the libusb library
    - The libusb provides low level (individual USB transactions) control of the device in the user application.
    - Debugging and testing are significantly simplified, but the performance is limited...



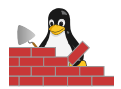
Dynamic devices management

- Certain interfaces (e.g., USB) allow connecting and disconnecting of devices while the system is running (hot-plugging, hot-swapping).
  Therefore, it is very important that the system automatically:
  - creates/deletes special files associated with connected/disconnected devices and assigns them the appropriate access rights,
  - uploads the firmware to the devices, enabling them to work.
- There are two tools delivering the above functionality:
  - mdev (short description)
  - eudev (documentation)
- Both tools use dedicated files defining the rules describing how the system should react on connection or disconnection of the device. The `eudev` is much more resource hungry then `mdev` but is also much more powerful.



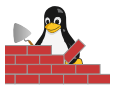
What about devices that can't be dynamically managed?

- Certain devices are not discovered automatically (e.g. they are connected to a bus that does not support device discovery, or they must be available in the early phase of system initialization)
- In the early stage of system development and debugging we can manually load the appropriate drivers with appropriate parameters
- When the system is stable, it is desired, that those devices are also handled in a more standard way
- It can be done in two ways:
  - With the "board file" compiled in the kernel, which describes the hardware configuration of our board (obsolete!)
  - With the "Device Tree", which is a kernel-independent description of the hardware configuration. It may be reused both by the bootloader and by the kernel.
- Presentation about porting the Linux and U-Boot to the new board.



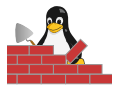
Board files

- Porting Linux on an ARM board (single slide 107)
- The board files contain the MACHINE_START and MACHINE_END macros (architecture-dependent), which define an architecture-dependent structure describing the basic properties of the board and its initialization procedure.
- Usually, when we want to create the “board file” for our board, we simply modify or extend the board file describing the most similar, already supported board.
- The most part of the board file are:
  - the declarations of the structures describing the board resources and addresses of their initialization procedures,
  - implementation of those initialization procedures.
- An example of the board file.
- Generally we should rather use the DT-based approach...



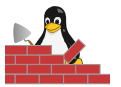
Device Tree

- Specification on devicetree.org, less formal description here.
- Simple presentation: T. Petazzoni - Device Tree for Dummies
- The basic elements of the Device Tree are the structures (“nodes”) describing the resources of the platform, their hierarchy (an example of a complex DTS), and defining which drivers can handle them (the example - device definition in the DTS ,compatibility declaration in the appropriate driver).
- Simple DTS for the platform, which may be used with QEMU:
  - vexpress-v2p-ca9.dts
- The “virt” platform automatically generates the DT information for the guest system.



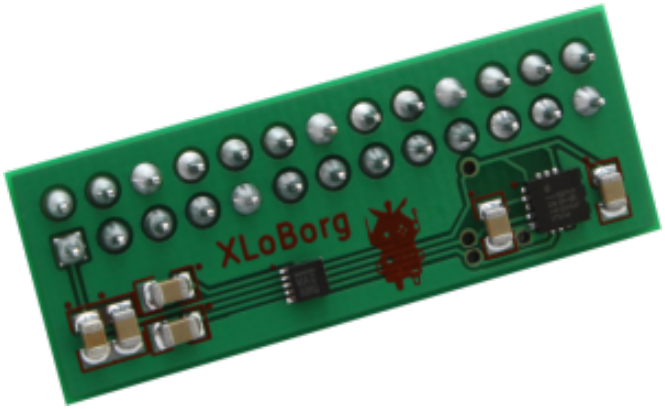
How is Raspberry Pi described?

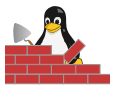
- Older versions of Buildroot used the board file bcm2708.c In the newer versions the device tree is used (the main file: bcm2708-rpi-b.dts)
- The usage of DT with RPi is described in the website.
- The important DT functionality in RPi is support for dynamically loaded and unloaded DT overlays.



So when do we need to deal with DT?

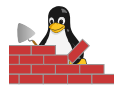
- If we create a new device that we want to connect to a bus that does not support autodetection. . .
- Then we need to create a DT describing the resources...





Example DT and audio in RPi

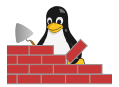
- In the default configuration of RPi kernel the built-in audio is disabled.
- We could fix it by applying the patch to the kernel, but unfortunately Buildroot for RPi does not compile DT from kernel source.
- Instead BR uses precompiled DT.
- There are modifications for BR to compile them from the sources. But they should be ported to the particular version.
- In a simple case of switching on the audio, we may add our own overlay and compile it independently.



What resources are available in RaspberryPi?

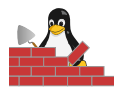
- I2C interface
- SPI interface
- I2S interface (the P5 connector must be soldered)
- GPIO – may be used for software ("bit banging") implementation of different interfaces, e.g.:
  - I2C, SPI, 1-Wire

Source: http://elinux.org/RPi_Low-level_peripherals



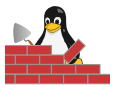
Usage of GPIO

- What is GPIO?
  - General purpose I/O
    - The pin of the SoC, which can be configured as output or as input. Depending on its configuration we can either read or set its state.
    - Sometimes we can configure more advanced functions like – interrupt generation, pull-up or pull-down resistors, etc.
- Availability from user applications
  - Direct access to the registers, e.g. via `/dev/mem`
  - Access via sysfs based driver (requires setting in the kernel configuration: `Device drivers->GPIO Support->sys/class/gpio...`, is portable)
  - Access via `/dev/mem` is faster but more risky (platform specific, possible race problems when multiple programs are accessing the same register, potential problems when we change the configuration of GPIO used for other purposes)
  - New library libgpiod (is portable)
  - Sometimes we can use dedicated libraries (e.g. wiringpi or python-rpi-gpio - also platform specific)



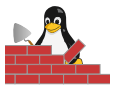
Access to GPIO via sysfs (old, deprecated, but still used)

- The documentation in kernel sources
- The presentation about usage of GPIO, SPI, and I2C
- Taking over and releasing the GPIO pin:
  - `echo number > /sys/class/gpio/export`
  - `echo number > /sys/class/gpio/unexport`
- Switching the pin direction:
  - `echo xx > /sys/class/gpio/gpioNN/direction`
  - `xx =[in` out | low | high]| (low & high switch on the output immediately setting the correct state without “glitches”)
- Setting and reading the pin state:
  - `echo [0 | 1] > /sys/class/gpio/gpioNN/value`
  - `cat /sys/class/gpio/gpioNN/value`
- Control of interrupt generation:
  - `echo [none | rising| falling| both] > /sys/class/gpio/gpioNN/edge`



Access to GPIO via sysfs - nuances

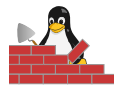
- If we read the GPIO from the application, and we keep the `.../value` file open, it has certain side effects.
  - To read the value again, we have to set the read position to the beginning of the file
  - If we set the read position to the **beginning** of the file, the **poll** function called for that file will put the current thread to sleep, until the appropriate change of the level occurs (ATTENTION! poll must be sensitive to POLLERR or POLLPRI events).



Example of a Python code waiting for pressing of the button

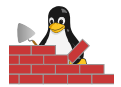
CAUTION! First you need to export GPIO and switch on interrupts generation

```
#!/usr/bin/python
import select
f=open("/sys/class/gpio/gpio27/value","r")
e=select.epoll()
e.register(f,select.EPOLLPRI)
while True:
        print f.read()
        e.poll()
        f.seek(0,0)
```



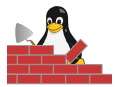
New method for accessing GPIO

- Since a few years a new method for accessing GPIOs is being proposed.
- In the new approach the GPIO is treated as a character device and is “owned” by the process.
  - When the process is ended, the status of the GPIO is restored
- The access is faster, but scripting possibilities are reduced *(the utilities like gpioget and gpioset are available, but the state of the pin after gpioset returns is unknown)*.
- Much more complex behavior of pins may be defined



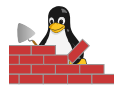
Where to learn the new method of GPIO access?

- The presentation from Linux Piter 2018,
- Kernel sources include/uapi/linux/gpio.h,
- C library encapsulating ioctl calls. *(Warning! The version used by BR may be different. Please see the* `gpiod.h` *file in a directory, where* `libgpiod` *was build, to know what are the functions prototypes.)*
- For Python - official examples from ligpiod sources, or "help(gpiod)" in Python.
- Examples in my repository BR_Internet_Radio branch gpio
- Demo project on the course Moodle website.



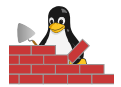
Difficulties in using GPIOs and mechanical switches

- The mechanical contacts exhibit so called “bounce effect”.
- It means, that when the state of the switch is changing, the value read from GPIO may oscilate.
- E.g., when switch changes from 0 to 1, reading at high speed we may get a sequence of values: ...(stable 0),0,1,0,0,1,1,0,1,0,1,1,0,1,1,(stable 1)...
- If the problem is not resolved in hardware, we must introduce a special mechanisms in our software.



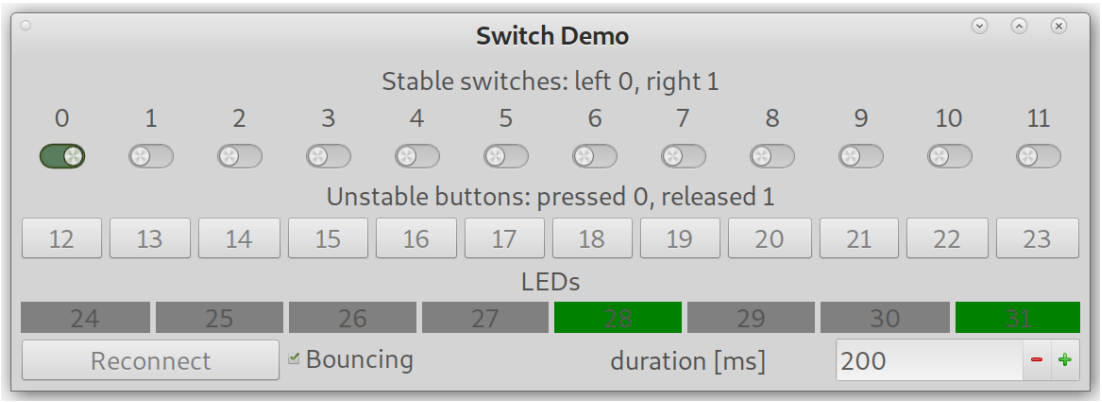
Eliminating the „contact bounce” in software

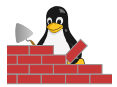
- One of the most efficient approaches using the available GPIO handling routines is based on the expected maximum duration of bouncing t_{bmax}.
- The program waits for the change of the GPIOs state.
- After the change is detected, it starts to wait to the next change, setting the timeout to t_{tbmax}.
- If the change is detected before timeout, we repeat the previous step.
- If there was no change before the timeout, we assume that switch is in a stable state and read its current value (or find it from the direction of the last change).



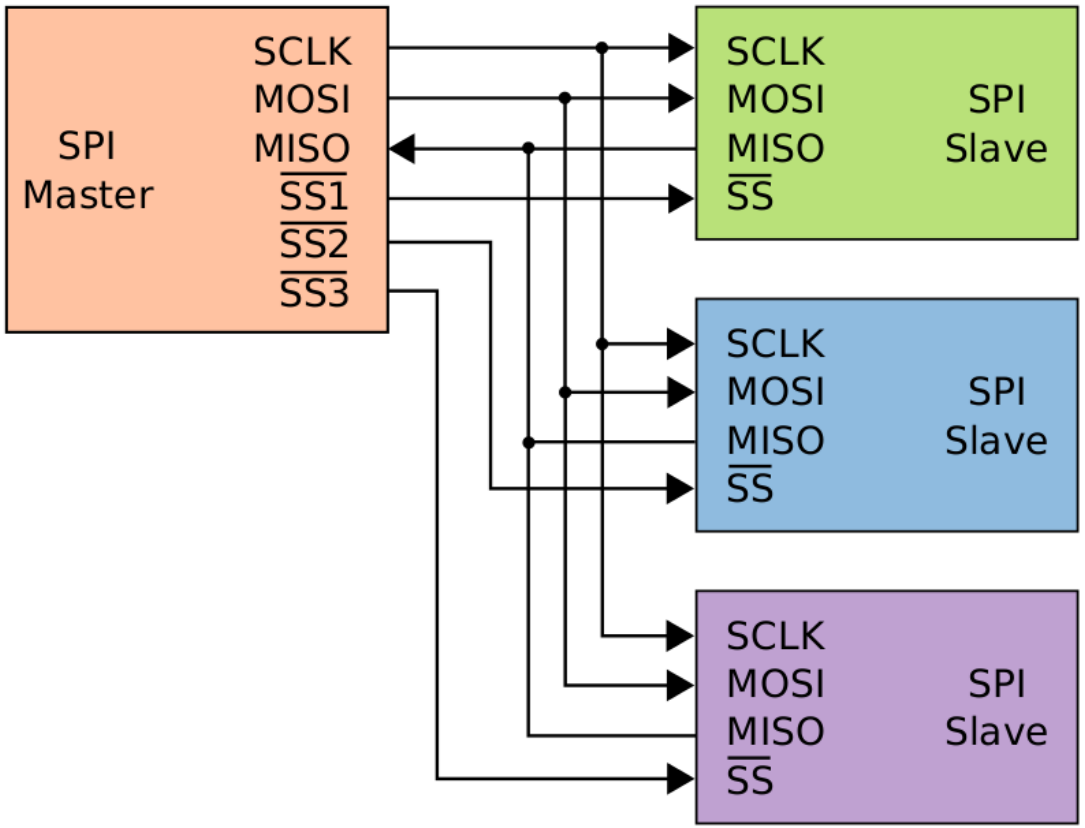
GPIOs in the virtual lab

- Especially for our lab, an emulation of GPIOs for QEMU has been prepared based on Virt 64 board (there is also an older version based on Vexpress A9). It is available in branch "gpio_simple_2021.02" of BR_Internet_Radio repository.
- GPIOs 0-11 are assigned to switches, 12-23 to buttons, and 24-31 to LEDs (outputs). GUI is provided by „gui2.py" Python script to be run on the host. You should set the direction accordingly.
- It is possible to emulate the "bounce" effect (run e.g., button13_read_irq.py).
- When using the "sysfs" approach, those GPIOs are numbered from 480 to 511, Using the new approach they are available as line 0-31 in "gpiochip0" or "9008000.gpio".

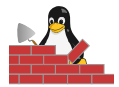




SPI interface

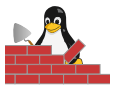


Source:GPIO, SPI and I2C from Userspace, the True LinuxWay



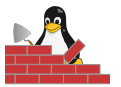
Examples of SPI-connected devices

- Accelerometer
- 24-channel PWM controller
- Long-range radio communication module



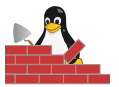
Handling SPI

- From the user application, we can access the SPI interface via the “spidev” driver.
- It creates special files `/dev/spidevX.Y`, where `X` is the bus number, and `Y` is the device number.
- Basic operations: read and write allow to read **or** write the block of data. To perform the typical for SPI operation of simultaneous reading and writing, we have to use the more advanced **ioctl** function.



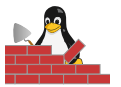
Handling SPI with ioctl

- The user application fills the spi_ioc_transfer structure and calls ioctl with command SPI_IOC_MESSAGE.
- Examples – programs spidev_fdx.c and spidev_test.c in kernel sources.



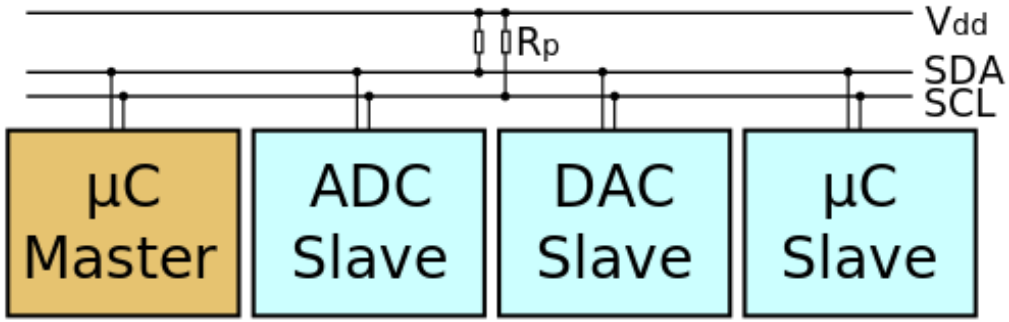
Handling of SPI via GPIO

- Often we have to control the SPI device connected to the GPIO pins.
- In such case, we can use the spi-gpio driver, but it requires that our device is defined in the board file or in the DT, example.
- The alternative approach may be to use the spi-gpio-custom.c module (the similar solution may be used in case of other drivers dedicated for “platform devices”).



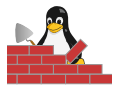
I2C interface

- I2C is a bus allowing to control multiple devices connected via two lines: `SDA` and `SCL` (+ ground)



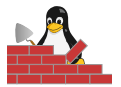
Source: Wikipedia

- Example devices: ADC, Accelerometer, Pressure, temperature & humidity sensor



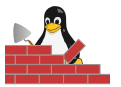
Handling I2C

- The user applications may control I2C connected devices via the i2c-dev driver described in kernel sources.
- The driver creates special files: `/dev/i2c-N` , where `N` is the controller number.
- Similarly to SPI, simple transfers may be performed using the write and read functions, but more complex operations require a use of **ioctl**.
- Particularly the ioctl with the `I2C_SLAVE` is necessary to set the address of the device we want to control!
- The `I2C_RDWR` allows to perform a complex sequence of read and write operations (in any order). Its argument is the address of the structure i2c_rdwr_ioctl_data that contains the list of addresses of the i2c_msg structures, describing the consecutive elementary operations.
- The code examples: [1] , [2]
- Warning! In certain situations, Raspberry Pi incorrectly handles I2C transfers.



Accessing I2C via GPIO

- Similarly like in the SPI case, it is necessary that our device is defined in the board file or in the DT, example.
- The workaround may be an i2c-gpio-custom.c module allowing to declare dynamically the GPIO lines used to implement the bit-banged I2C interface.

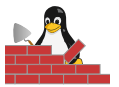


1-Wire interface

- The 1-Wire interface allows to control the device using only a single line (+ ground). In addition, the same line may be used to supply the device!
- The protocol support is described in the kernel sources.
- Connection via GPIO requires the w1-gpio driver and support in DT.
- The devices are automatically discovered and handled by dedicated drivers. If there is no dedicated driver, the device may be handled via sysfs (the virtual “rw” file), but this option is poorly documented.

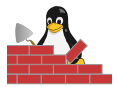
Authentication device with 1-wire interface (source: Wikipedia)

DS18B20 – thermometer with I2C and 1-Wire interfaces



IIO - Industrial I/O subsystem

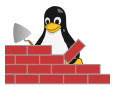
- General concept of the solution is described in Linux Industrial I/O Subsystem.
- It is oriented mainly on A/D and D/A converters, and digitally controlled sensors and actuators.
- The devices are usually connected via SPI, I2C or a similar interface, but they are taken over by the appropriate IIO driver (see the drivers/iio directory).
- The devices are accessible through `sysfs` via files in `/sys/bus/iio/devices`, or via character devices `/dev/iio:deviceX`.
- There is also a libiio API library.



IIO - Example of script configured data acquisition 1

- Data acquisition triggered with “high resolution timer”.

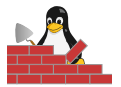
```
modprobe iio-trig-hrtimer
mkdir /sys/kernel/config/iio/triggers/hrtimer/trigger0
echo 10 > /sys/bus/iio/devices/trigger0/sampling_frequency
echo 1 > /sys/bus/iio/devices/iio\:device0/scan_elements/in_voltage3_en
echo trigger0 > /sys/bus/iio/devices/iio\:device0/trigger/current_trigger
echo 1 > /sys/bus/iio/devices/iio\:device0/buffer/enable
cat /dev/iio:device0 | xxd -
```



IIO - Example of script configured data acquisition 2

- Multichannel data acquisition triggered with “high resolution timer” (not all drivers support that!).

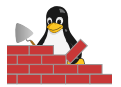
```
modprobe iio-trig-hrtimer
mkdir /sys/kernel/config/iio/triggers/hrtimer/trigger0
echo 10 > /sys/bus/iio/devices/trigger0/sampling_frequency
echo 1 > /sys/bus/iio/devices/iio\:device0/scan_elements/in_voltage0_en
echo 1 > /sys/bus/iio/devices/iio\:device0/scan_elements/in_voltage1_en
echo 1 > /sys/bus/iio/devices/iio\:device0/scan_elements/in_voltage3_en
echo trigger0 > /sys/bus/iio/devices/iio\:device0/trigger/current_trigger
echo 1 > /sys/bus/iio/devices/iio\:device0/buffer/enable
cat /dev/iio:device0 | xxd -
```



IIO - Example of script configured data acquisition 3

- Data acquisition triggered with “high resolution timer” with additional time marker.

```
modprobe iio-trig-hrtimer
mkdir /sys/kernel/config/iio/triggers/hrtimer/trigger0
echo 10 > /sys/bus/iio/devices/trigger0/sampling_frequency
echo 1 > /sys/bus/iio/devices/iio\:device0/scan_elements/in_voltage3_en
echo 1 > /sys/bus/iio/devices/iio\:device0/scan_elements/in_timestamp_en
echo trigger0 > /sys/bus/iio/devices/iio\:device0/trigger/current_trigger
echo 1 > /sys/bus/iio/devices/iio\:device0/buffer/enable
cat /dev/iio:device0 | xxd -
```



Thank you for your attention!