TERO PUF Linux Driver (Extended Doc)

Andrea Aspesi

June 2022

The PUF Device

Brief description of the hardware device realized on the FPGA

The PUF Device

Physical Unclonable Functions (PUFs) are becoming important in recent years for various purposes such as Device Identification, True Random Number Generators, Cryptographic Key Generators, IP Protection.

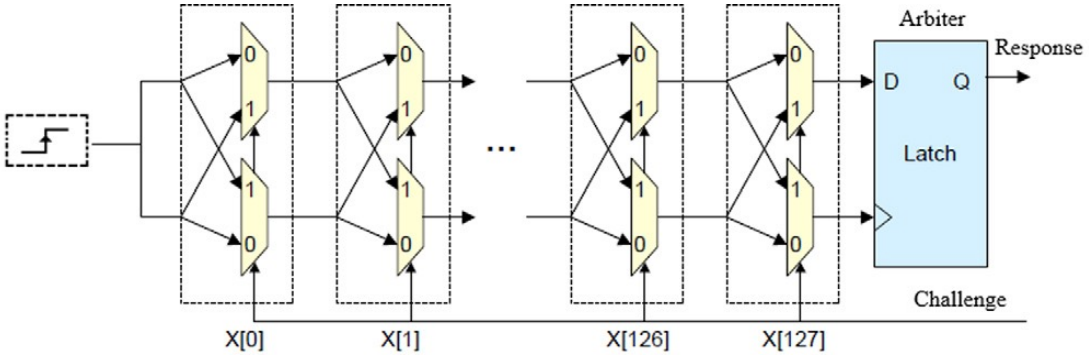
Different ways of realizing PUFs have been proposed in literature, exploiting process variations occurring inside chips.

The PUF Device

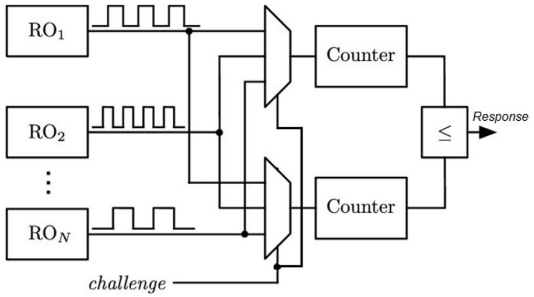
As these defects are an unwanted byproduct of the realization of the chip, they cannot be replicated nor controlled.

There are three main categories of PUFs:

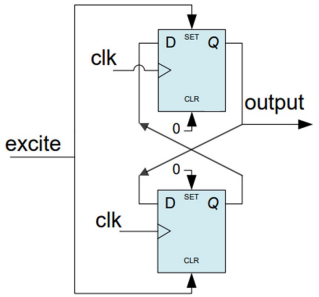
- **based on race conditions**: exploit the fact that signals propagating on fixed traces take a different time on each chip, as these traces are not perfectly equal.
- **based on oscillations**: after creating an instable condition in a loop, this starts to oscillate to resolve it. But since in each chip the loop has different characteristics, the frequency of oscillation will change
- **based on memory**: exploits the random configuration of memory cells at startup



(1) delay-based



(2) frequency-based



(3) memory-based

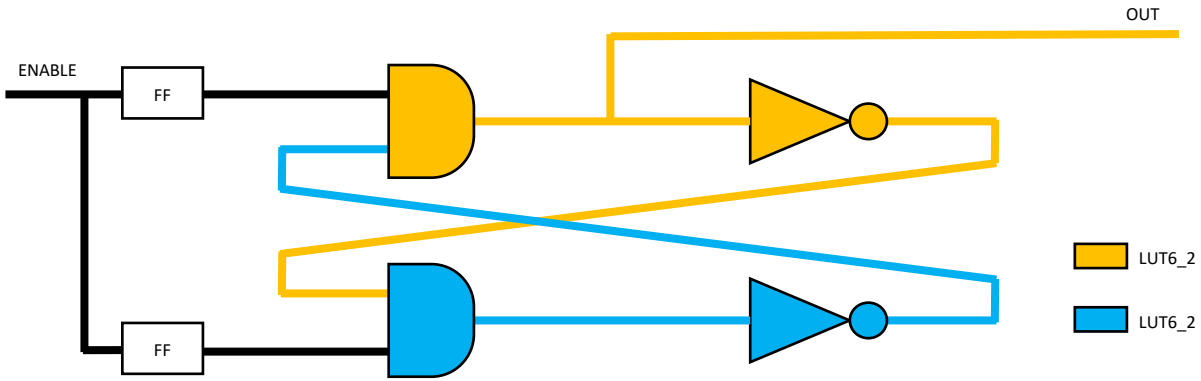
The PUF Device

The first two methods can be realized also on FPGA.

In our project, after analyzing the state of the art, we chose to implement a *Transient Effect Ring Oscillator* (**TERO**) PUF.

It offers high resistance to surrounding logic influence and temperature changes, providing at the same time high uniqueness and reliability (see next slides).

As the name suggests, a TERO PUF is an *oscillation-based PUF*, composed of two AND gates and to NOT gates.

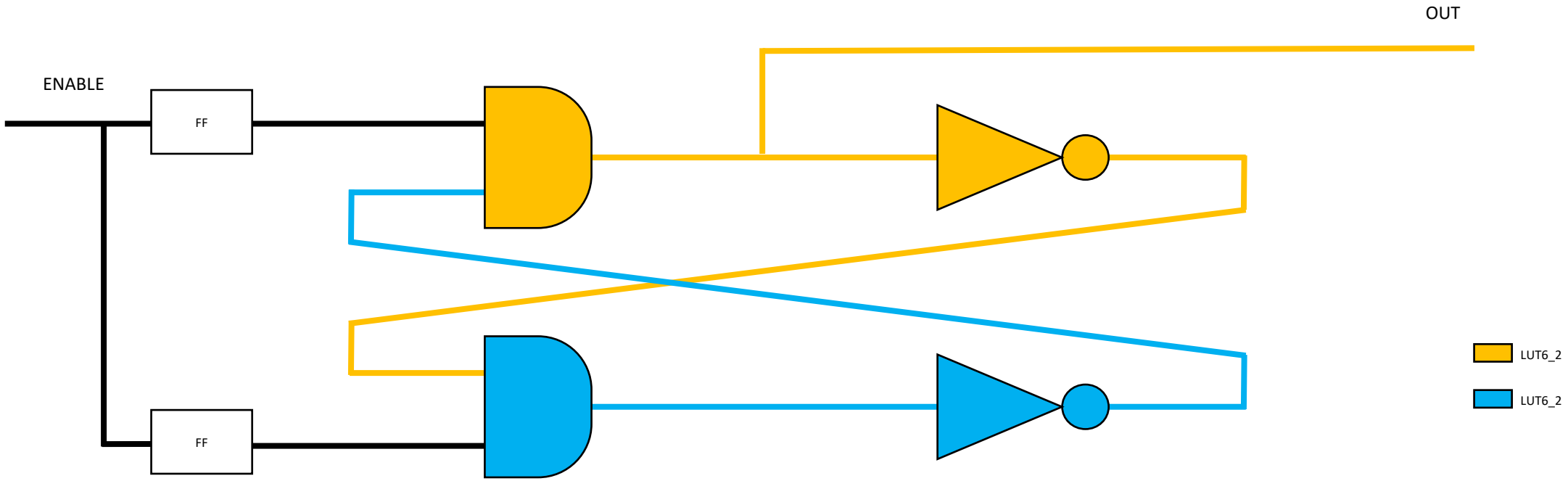


TERO loop implemented in ½ Artix-7 slice

The PUF Device

The loop is brought into a metastable condition using an enable signal. It then oscillates to settle into a stable state, but the number of oscillation needed depends strictly on the electrical properties of the loop.

Each loop performs a unique number of oscillations before settling composing this number of oscillations, for each loop of the PUF, permits to generate a unique identifier.



TERO loop implemented in ½ Artix-7 slice

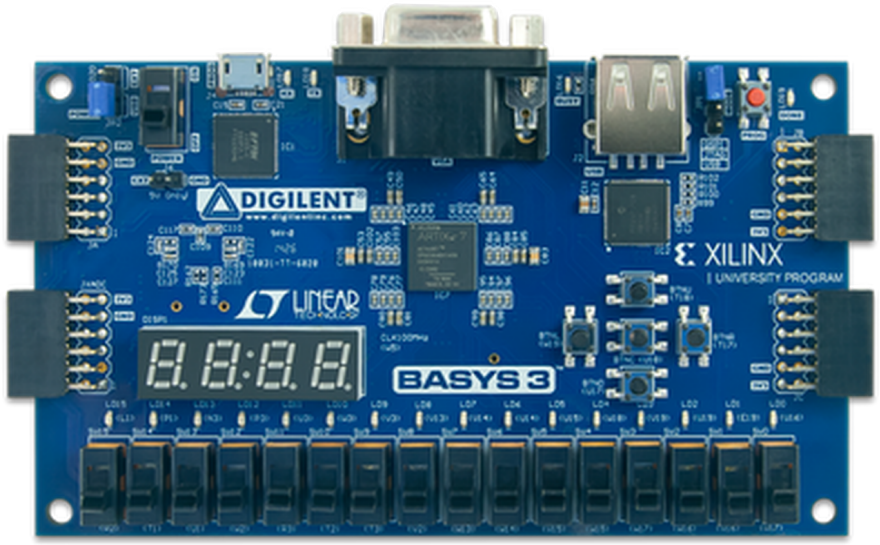
The PUF Device

The generated identifier is tested over two properties:

- **reliability**: over different measures in time (also at different temperatures), the identifier must remain ideally identical (100% ideal value), considering the *Hamming Distance*.
- **uniqueness**: taking two different PUFs, the chance of observing a 1 or 0 in the i-th position of the generated responses is equal (50%, flip of a balanced coin).

In our implementation, we obtained:

- a mean reliability of **97%** (*less than 3 bits can flip during successive readings for the same device*)
- a mean uniqueness of **50.25%** (*great capacity of distinguish two different devices*)



Target board (Artix-7 based)

The PUF Device

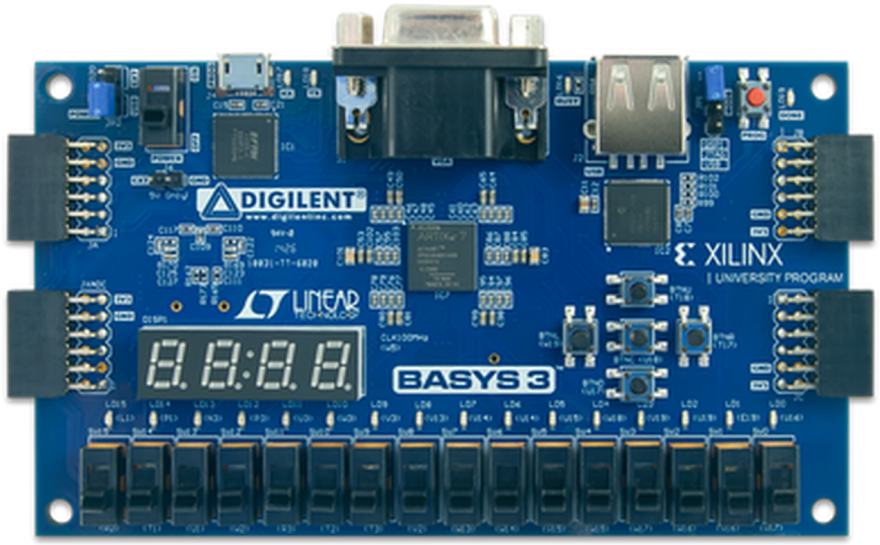
In our implementation, the device can generate not just a single unique identifier of *80bits*, but 120 identifiers (each of *80bits*).

Each identifier is associated with a number called **challenge number** (from 0 to 119).

The main advantage of using a PUF is that these identifiers do not exist until the hardware generates them, so in principle is impossible to tamper them in advance.

This permits to have 120 secure authentication exchanges. If some of the IDs are collected after device manufacturing, then it's in principle possible to compare these with the IDs generated by the same challenges once the device is put into an unsafe environment.

As the CRP (*Challenge-Response*) space is very small, it's still possible for an attacker to force the device to generate all the possible IDs in this case.



Target board (Artix-7 based)

The PUF Device

The target FPGA used in this work is the evaluation board **Basys3** from Digilent, shipped with **Xilinx Artix-7 XC7-A35T**.

The choice is due to this board being the target of the reference paper:

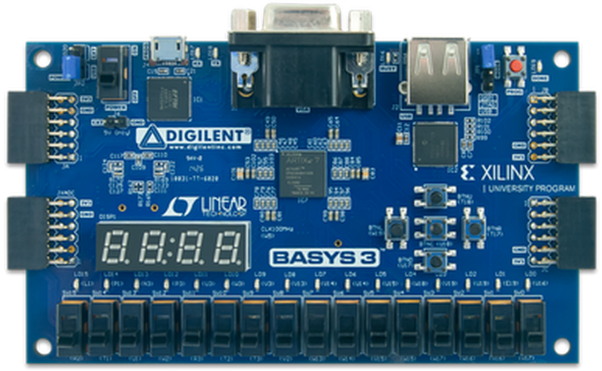
A Fair and Comprehensive Large-Scale Analysis of Oscillation-Based PUFs for FPGAs

- Alexander Wild, Georg T. Becker, Tim Guneysu (published in 2017)

https://ieeexplore.ieee.org/document/8056795

We used slightly less resources than the original implementation, with a total of:

- Slices: **19.47%** (1109 SliceL, 478 SliceM)
- LUTs: **21.66%** (4506 used over 20800 available)
- Flip Flops: **8.17%** (3397 used over 41600 available)
- Block RAMS: **4.00%** (2 used over 50 available)



Basys3

Table 1-2: **Artix-7 FPGA CLB Resources**

Device	Slices[1]	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
7A12T	2,000[2]	1,316	684	8,000	171	86	16,000
7A15T	2,600[2]	1,800	800	10,400	200	100	20,800
7A25T	3,650	2,400	1,250	14,600	313	156	29,200
7A35T	5,200[2]	3,600	1,600	20,800	400	200	41,600
7A50T	8,150	5,750	2,400	32,600	600	300	65,200
7A75T	11,800[2]	8,232	3,568	47,200	892	446	94,400
7A100T	15,850	11,100	4,750	63,400	1,188	594	126,800
7A200T	33,650	22,100	11,550	134,600	2,888	1,444	269,200

Resources available on Basys3

The PUF Driver

Brief description of the objective of the driver related to the PUF

The PUF Driver

In our implementation, the PUF is not able to generate the complete unique ID directly in hardware.

The PUF simply collects the number of oscillations (frequencies) of the loops, then sends this data to the PC.

Further elaboration is then required to extract the needed **ID**, or more specifically the 120 unique IDs the PUF can provide (*each associated to a challenge*).

The driver offers an abstraction over the communication protocol used to get the frequencies, and the process of ID generation.

It creates a simple character device for each connected PUF, that can be read to get the last generated response ID and can be written to select the challenge (and trigger the generation of the associated ID).

Up to 32 PUFs can be connected to the same PC.

The PUF Driver

This character device:

- can be read system-wide also from standard users. In this way every user-space program that needs a unique ID can easily get it.
- can be written to select the challenge only from a privileged user. This is needed as for the driver's design the communication with a device reader is not specialized: the response is shared across readers. If every reader asks for a different challenge, then unexpected behaviors can arise for the readers (the driver internally still would work fine).

The challenge number is seen as a parameter of the system set by a single privileged user, for example at system startup.

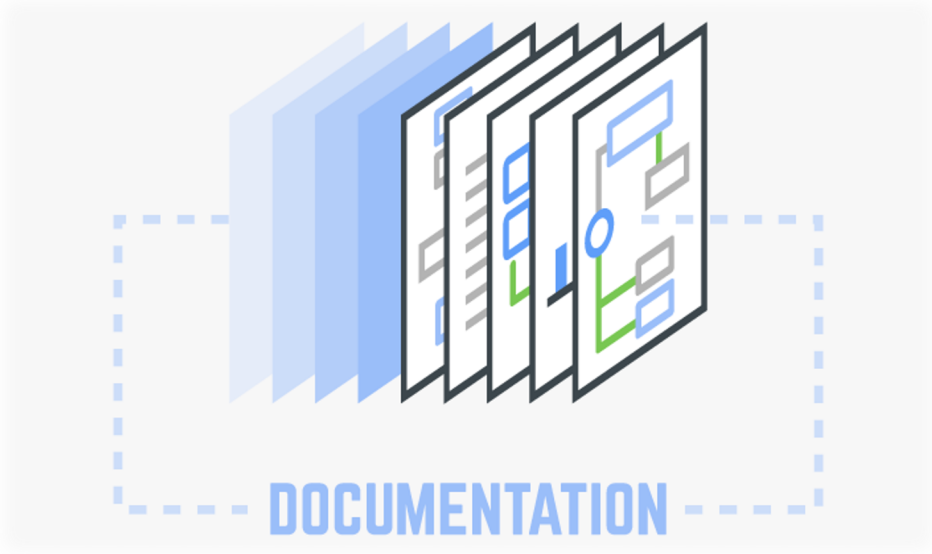
The PUF Driver

The following slides guide the reader to understand each key feature used in the driver.

Code extracts are also present, taken from the linked documentation or directly from the driver source code.

The analysis:

1. starts with an in-depth overview of how the current legacy driver permits to communicate with the PUF
2. shows how this approach was modified considering only the Basys3 specific hardware
3. deals with the Linux USB Subsystem and how the driver can be loaded in the Linux Kernel
4. explains how the driver manages its state
5. concludes with how a response is generated, and how the interaction with the driver is possible

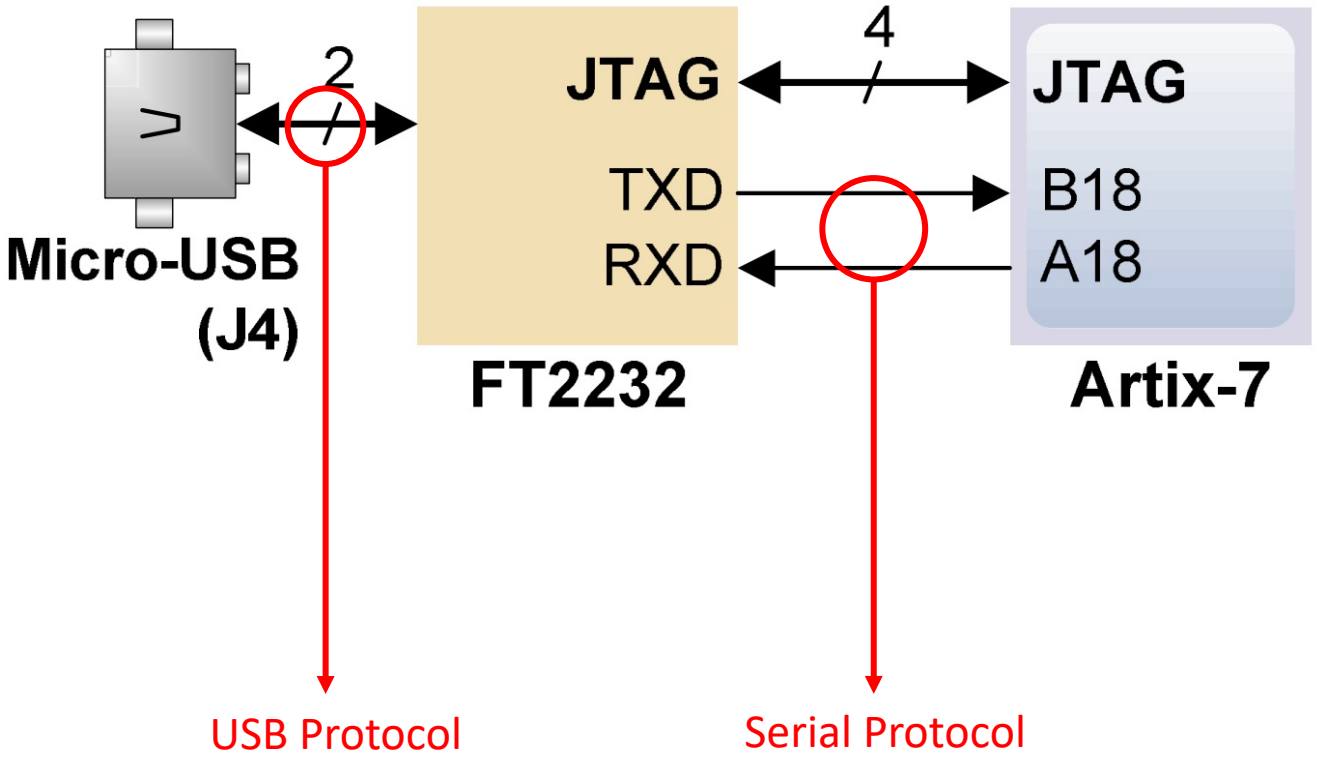


Legacy Driver Analysis

Analysis of the legacy driver supplying the Serial Port for the device, under Linux

Communication Hardware

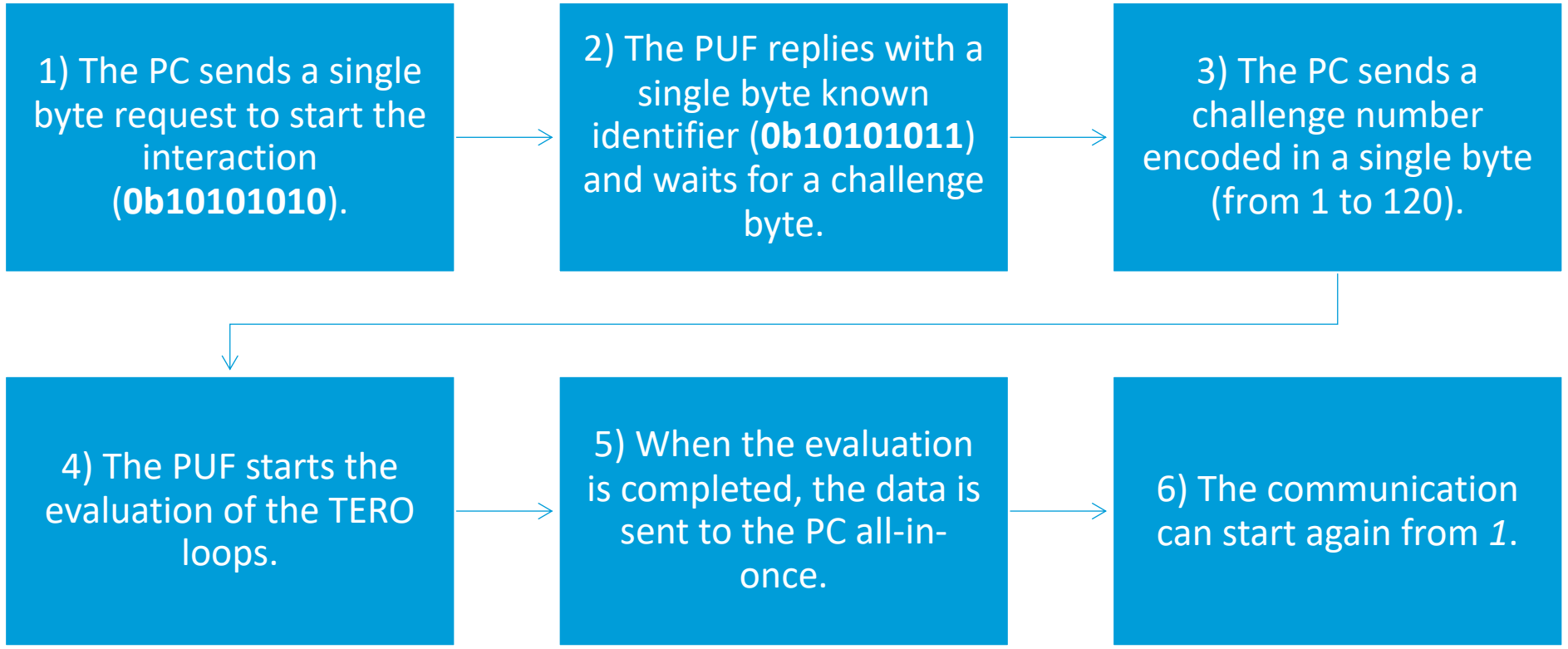
The FPGA communicates with the PC via **UART** (*Universal Asynchronous Receiver-Transmitter*). On the Basys3 board is present a FTDI chip[1], that makes possible to use the serial channel from a standard USB connection.



[1] *See page 7 of* https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/Basys3/Supporting%20Material/Basys3_RM.pdf

Serial Protocol of PUF

Considering the Serial Protocol, in order to acquire the device ID from the FPGA, the following steps must be followed:



USB Protocol

The USB protocol is well regulated under the **USB 2.0 Protocol Specification**, explained here[2].

As it does not depend on the specific device, and it's widely adopted/used in PCs of all ages (*almost all external devices are connected via USB bus*), ad-hoc features already exist in the Linux Kernel.
In particular, in the Linux Kernel, the **USB Subsystem** is responsible to provide an abstraction of the physical protocol to the many drivers, providing them pipes to send commands and send/receive data in different modalities.

For most commercially distributed USB to Serial Adaptors, device drivers are already present in the Linux Source Tree. This is the case for all devices produced by FTDI (and so the one of the Basys3).
Instead of writing a driver from scratch, the original FTDI driver will be analyzed and modified.
This driver currently provides a Serial Port interface, exposed to applications in user space.

The idea behind the new driver is to mask the serial communication with the FPGA, and instead provide a simple character device where applications can directly read the final response ID of any connected TERO PUF.

[2] *See* https://www.renesas.com/us/en/application/key-technology/usb-technology/usb1-2

FTDI Driver Analysis

The specific FTDI chip shipped with the **Basys3** board is the **FT2232HQ**. This chip provides a dual channel USB to Serial, as it will be shown in the next slides.

The commands needed to configure this devices will be extracted directly from the FTDI driver source code, as not well documented online.

In particular, the driver for all FTDI devices can be found in the linux kernel source code[3], under the path:

linux/drivers/usb/serial

split into three files:

- **ftdi_sio.c** (driver implementation)
- **ftdi_sio.h** (driver headers)
- **ftdi_sio_ids.h** (driver supported devices)

[3] *See* https://github.com/torvalds/linux

FTDI Driver Analysis

When connecting the device, without Xilinx custom driver, we observe via **dsmg** command:

```
[  359.914998] usb 1-1: new high-speed USB device number 2 using ehci-pci
[  360.269571] usb 1-1: New USB device found, idVendor=0403, idProduct=6010, bcdDevice= 7.00
[  360.269575] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  360.269579] usb 1-1: Product: Digilent USB Device
[  360.269582] usb 1-1: Manufacturer: Digilent
[  360.269583] usb 1-1: SerialNumber: 210183B46505
[  360.295092] usbcore: registered new interface driver usbserial_generic
[  360.295099] usbserial: USB Serial support registered for generic
[  360.300135] usbcore: registered new interface driver ftdi_sio
[  360.300146] usbserial: USB Serial support registered for FTDI USB Serial Device
[  360.300168] ftdi_sio 1-1:1.0: FTDI USB Serial Device converter detected
[  360.300218] usb 1-1: Detected FT2232H
[  360.301603] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB0
[  360.301657] ftdi_sio 1-1:1.1: FTDI USB Serial Device converter detected
[  360.301677] usb 1-1: Detected FT2232H
[  360.304190] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB1
```

The first useful information is:

- **idVendor=0x0403** (FTDI_VID, original FTDI vendor id)
- **IdProduct=0x6010** (FTDI_8U2232C_PID, dual channel device)

This data can be confirmed by inspecting the file **ftdi_sio_ids.h**

FTDI Driver Analysis

When connecting the device, without Xilinx custom driver, we observe via **dsmg** command:

```
[  359.914998] usb 1-1: new high-speed USB device number 2 using ehci-pci
[  360.269571] usb 1-1: New USB device found, idVendor=0403, idProduct=6010, bcdDevice= 7.00
[  360.269575] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  360.269579] usb 1-1: Product: Digilent USB Device
[  360.269582] usb 1-1: Manufacturer: Digilent
[  360.269583] usb 1-1: SerialNumber: 210183B46505
[  360.295092] usbcore: registered new interface driver usbserial_generic
[  360.295099] usbserial: USB Serial support registered for generic
[  360.300135] usbcore: registered new interface driver ftdi_sio
[  360.300146] usbserial: USB Serial support registered for FTDI USB Serial Device
[  360.300168] ftdi_sio 1-1:1.0: FTDI USB Serial Device converter detected
[  360.300218] usb 1-1: Detected FT2232H
[  360.301603] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB0
[  360.301657] ftdi_sio 1-1:1.1: FTDI USB Serial Device converter detected
[  360.301677] usb 1-1: Detected FT2232H
[  360.304190] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB1
```

As can be observed from the FTDI product page[3] the FT2232HQ offers a dual channel communication (seen as the two **FT2232H** devices from the kernel).
The second device is the actual serial channel with the FPGA, the first is used via JTAG to program the FPGA from Xilinx.

[3] *See* https://ftdichip.com/products/ft2232hq/

FTDI Driver Analysis

When connecting the device, without Xilinx custom driver, we observe via **dsmg** command:

```
[  359.914998] usb 1-1: new high-speed USB device number 2 using ehci-pci
[  360.269571] usb 1-1: New USB device found, idVendor=0403, idProduct=6010, bcdDevice= 7.00
[  360.269575] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  360.269579] usb 1-1: Product: Digilent USB Device
[  360.269582] usb 1-1: Manufacturer: Digilent
[  360.269583] usb 1-1: SerialNumber: 210183B46505
[  360.295092] usbcore: registered new interface driver usbserial_generic
[  360.295099] usbserial: USB Serial support registered for generic
[  360.300135] usbcore: registered new interface driver ftdi_sio
[  360.300146] usbserial: USB Serial support registered for FTDI USB Serial Device
[  360.300168] ftdi_sio 1-1:1.0: FTDI USB Serial Device converter detected
[  360.300218] usb 1-1: Detected FT2232H
[  360.301603] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB0
[  360.301657] ftdi_sio 1-1:1.1: FTDI USB Serial Device converter detected
[  360.301677] usb 1-1: Detected FT2232H
[  360.304190] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB1
```

We can also see that the driver (**ftdi_sio**) exploits the common base driver **usbserial**.
This driver is a generic serial driver that manages the connection between the device specific driver (*ftdi_sio* in this case) and the *tty_driver* (that supplies the tty character device, used by the applications).

FTDI Driver Analysis

When connecting the device, without Xilinx custom driver, we observe via **dsmg** command:

```
[  359.914998] usb 1-1: new high-speed USB device number 2 using ehci-pci
[  360.269571] usb 1-1: New USB device found, idVendor=0403, idProduct=6010, bcdDevice= 7.00
[  360.269575] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  360.269579] usb 1-1: Product: Digilent USB Device
[  360.269582] usb 1-1: Manufacturer: Digilent
[  360.269583] usb 1-1: SerialNumber: 210183B46505
[  360.295092] usbcore: registered new interface driver usbserial_generic
[  360.295099] usbserial: USB Serial support registered for generic
[  360.300135] usbcore: registered new interface driver ftdi_sio
[  360.300146] usbserial: USB Serial support registered for FTDI USB Serial Device
[  360.300168] ftdi_sio 1-1:1.0: FTDI USB Serial Device converter detected
[  360.300218] usb 1-1: Detected FT2232H
[  360.301603] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB0
[  360.301657] ftdi_sio 1-1:1.1: FTDI USB Serial Device converter detected
[  360.301677] usb 1-1: Detected FT2232H
[  360.304190] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB1
```

This can be noted when launching the command "**ls -lh /dev**" :
/dev/ttyUSB1 is registered with the driver major number **188**, the same defined in **usb-serial.c** (the implementation source of **usbserial** driver)

Towards a TERO Driver

Presentation of the
methods of the USB Subsystem
used to communicate to the
FTDI chip

Towards a TERO Driver

To implement the custom driver, the link between the **tty_driver** and the **ftdi_sio** must be removed.
Moreover, **ftdi_sio** must be modified to communicate with the FPGA, in order to collect frequencies and compute the device ID.

Before starting the communication, it's necessary to configure the FTDI chip to set the proper Serial Configuration.

In particular, the configuration is the following:

- Baudrate=**115200**
- Databits=**8**
- Stopbits=**1**
- Parity=**None**

Let's see in details how this can be achieved exploting the USB Subsystem of the Linux Kernel, in the next slides.

Towards a TERO Driver

Crafting an URB (USB Request Block), a kernel structure that is properly initialized using helper functions, we can submit commands and data to a device connected via USB protocol. In this way we can avoid all the complexity of the USB protocol itself.

To use an URB, we need:

- the pointer to the **usb_struct** of the device (obtained during *probe* method, see later)
- the channel of communication, called **pipe** (depends on the usb device)
- a **buffer** where data is stored (and consumed/put by the USB Subsystem). The buffer must be allocated with kmalloc/kzalloc, in order to be accessible inside kernel functions.

In case of synchronous (blocking) communication, helper functions exist to avoid directly initializing URBs.
For example, to send a command to the device, we can use:

```
int usb_control_msg(struct usb_device * dev,
unsigned int pipe, __u8 request, __u8 requesttype,
__u16 value, __u16 index, void * data, __u16 size,
int timeout);
```

See https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

Towards a TERO Driver

The **pipe** argument can be found by enumerating the endpoints of the device, or by reading the device documentation (*provided that exists*).

The first information can be extracted from the **interface structure**, supplied at device connection (details later)*:*

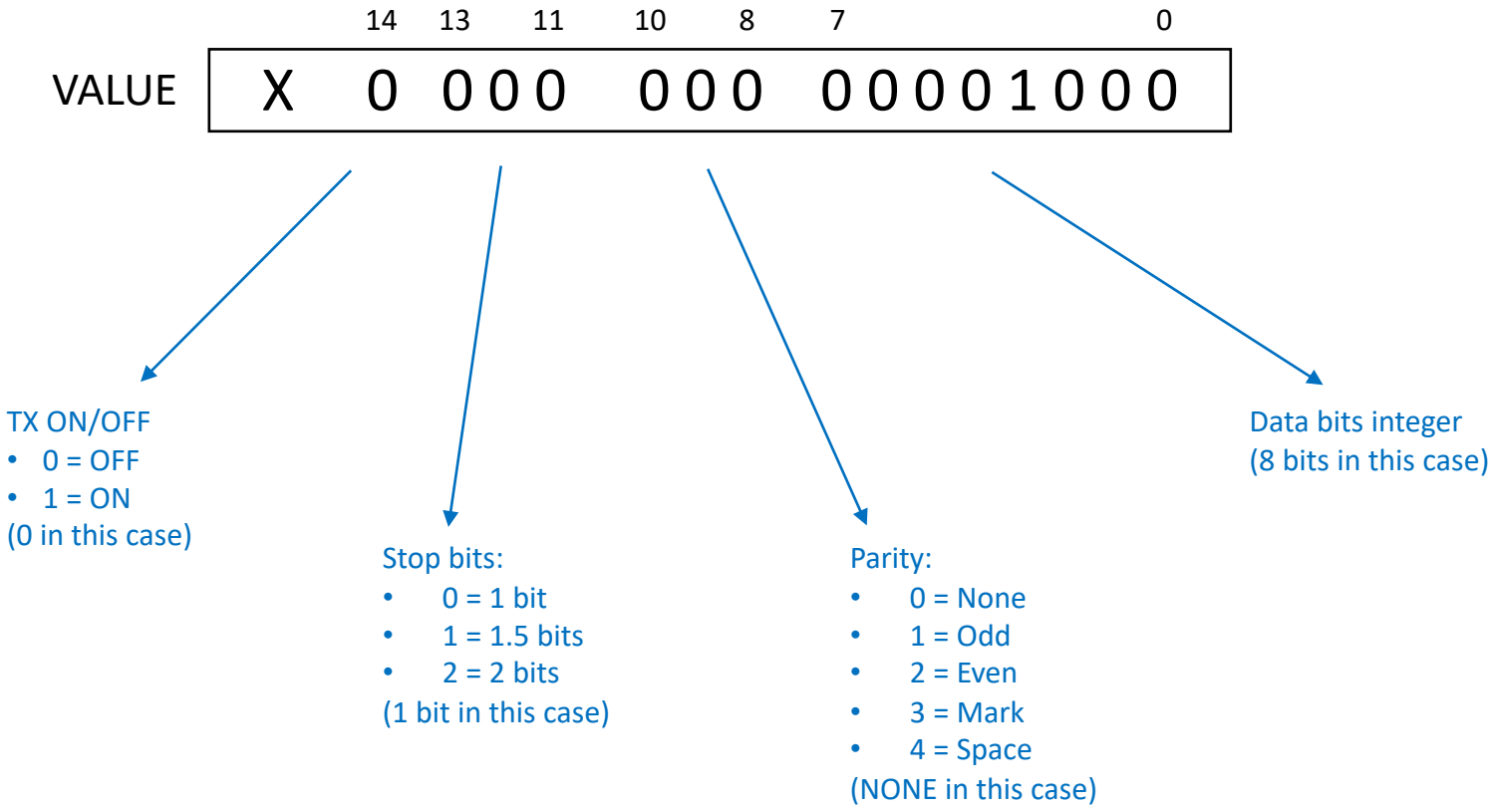
```
num_endpoints = interface->cur_altsetting->desc.bNumEndpoints;

for (i = 0; i < num_endpoints; ++i) {
    ep_desc = &interface->cur_altsetting->endpoint[i].desc;
    printk("Endpoint [%d]: Max packet size %d, DirIN: %d, DirOUT: %d",
        usb_endpoint_num(ep_desc),
        usb_endpoint_maxp(ep_desc),
        usb_endpoint_dir_in(ep_desc),
        usb_endpoint_dir_out(ep_desc));
}
```

This is the pipe number used in URB

Towards a TERO Driver

The *request, request type, value, index* are found in the datasheet of the device (*if available*).

For example, to set the stop bits, parity and data bits of the FTDI chip a 16bits integer is used, setting the corresponding bits:

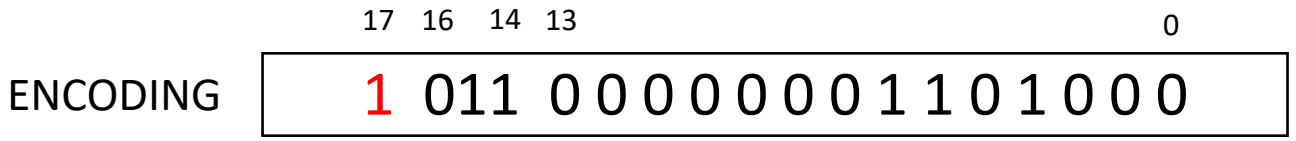


To apply this command:

- REQUEST = 0x4
- REQUEST_TYPE = 0x40
- INDEX = 0

Towards a TERO Driver

Setting the baudrate, instead, is more complicated. A divisor of the clock must be supplied to the device, instead of the baudrate. The encoding of this divisor was modified during the years to support higher baudrates (up to 12Mhz).
As we have a clock of 120Mhz, prescaled to 12Mhz (?), we encode the divisor as follows (down to 1200bit/s):



To use directly the 12Mhz as base for the divisor, the MSB must be set to 1.

Decimal part:
- 000 = 0.0 (0)
- 011 = 0.125 (3)
- 010 = 0.25 (2)
- 100 = 0.375 (4)
- 001 = 0.5 (1)
- 101 = 0.625 (5)
- 110 = 0.75 (6)
- 111 = 0.875 (7)

(0.125 in this case)

Integer part
(104 in this case)

To use 115200bit/s, we need a divider of 12Mhz/115200=104.166 -> 104.125

To apply this command:
- REQUEST = 0x3
- REQUEST_TYPE = 0x40
- VALUE = ENCODING & 0xFFFF
- INDEX = (ENCODING >> 16) << 8 | INTERF.

Where INTERF = 2 in this case for FT2232H, INTERFACE_B

See http://www.ftdichip.com/Documents/AppNotes/AN_120_Aliasing_VCP_Baud_Rates.pdf

Towards a TERO Driver

After the configuration is applied, it is possible to send data to the device using:

```
int usb_bulk_msg(struct usb_device * usb_dev,
unsigned int pipe, void * data, int len,
int * actual_length, int timeout);
```

where *pipe is usb_sndbulkpipe(data->udev, FTDI_PIPE_OUT)*.

FTDI_PIPE_OUT is the **usb_endpoint_num** with DirOUT = 1 of slide 27.

If the returned value is negative, the operation failed. In *actual_length* is also stored the actual number of bytes sent, that should be equal *len* if everything went fine.

See https://manpages.debian.org/jessie-backports/linux-manual-4.8/usb_bulk_msg.9.en.html

Towards a TERO Driver

To receive data synchronously, is sufficient to change the pipe of the previous command:

```
int usb_bulk_msg(struct usb_device * usb_dev,
unsigned int pipe, void * data, int len,
int * actual_length, int timeout);
```

where *pipe is* *usb_rcvbulkpipe(data->udev, FTDI_PIPE_IN)*.

FTDI_PIPE_IN is the **usb_endpoint_num** with DirIN = 1 of slide 24.

If the returned value is negative, the operation failed. In *actual_length* is also stored the actual number of bytes read, of the maximum requested in *len*, that can be up to **usb_endpoint_maxp** (read buffer FIFO size, 512).

See https://manpages.debian.org/jessie-backports/linux-manual-4.8/usb_bulk_msg.9.en.html

Towards a TERO Driver

To avoid blocking while waiting data, it's possible to directly create the read URB, and set its callback.
To create an URB, use the following function, and create a read buffer.

data->in_urb = **usb_alloc_urb**(0, *GFP_KERNEL*);

To setup the URB to read, use:

```
void usb_fill_bulk_urb(struct urb * urb, struct usb_device * dev,
unsigned int pipe, void * transfer_buffer, int buffer_length,
usb_complete_t complete_fn, void * context);
```

where:

- *pipe* is the same value of the sync. reception case
- *complete_fn* is the callback function of signature: `void read_callback(struct urb * urb)`
- *context* is a pointer to a structure we want to access during the callback, with *urb->context*.

The callback is called in interrupt context, so no blocking/sleeping is allowed.

See https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

Towards a TERO Driver

To start reading, send the configured URB to the USB Subsystem via:

```
int usb_submit_urb(struct urb * urb, gfp_t mem_flags);
```

In normal context, **mem_flags** *= GFP_KERNEL*. In the callback (interrupt context), the flag *GFP_ATOMIC* must be used. At the end of the callback, the URB can be sent again to the Subsystem with the previous function.

FDTI chips always return data every 40ms, reporting the status in the first two bytes. If actual data is present, it starts from the third byte. *This must be considered during the implementation*.

Byte 0: Modem Status

- B0 Reserved - must be 1
- B1 Reserved - must be 0
- B2 Reserved - must be 0
- B3 Reserved - must be 0
- B4 Clear to Send (CTS)
- B5 Data Set Ready (DSR)
- B6 Ring Indicator (RI)
- B7 Receive Line Signal Detect (RLSD)

Byte 1: Line Status

- B0 Data Ready (DR)
- B1 Overrun Error (OE)
- B2 Parity Error (PE)
- B3 Framing Error (FE)
- B4 Break Interrupt (BI)
- B5 Transmitter Holding Register (THRE)
- B6 Transmitter Empty (TEMT)
- B7 Error in RCVR FIFO

See https://www.kernel.org/doc/html/latest/driver-api/usb/URB.html

Towards a TERO Driver

When the device disconnects, to provide a clean exit, the driver must stop all pending operations on that device.
To this purpose, all sent URBs must be stopped and deallocated.

To stop a sent URB (blocking call), use:

void usb_kill_urb(**struct urb *** urb);

To free the URB, after it is no longer in use, call:

void usb_free_urb(**struct urb *** urb);

See https://www.kernel.org/doc/html/latest/driver-api/usb/URB.html

TERO Driver Initialization

Description of the generic start-up of the driver

TERO Driver Initialization

The driver is a Linux Kernel Module. Exploting the LKM Framework of the Linux Kernel, it's possible to load/unload these modules without the need to restart the system.

To this purpose, the module must provide two entry-points (**init** and **exit**).

These can be set calling the following macros:

module_init(puf_driver_init);
module_exit(puf_driver_exit);

In the **init** function (*puf_driver_init*), the driver can allocate needed structures, register itself to the USB Subsystem, create character devices, …

When **exit** function (*puf_driver_exit*) is called by the framework, the driver must stop all operations, and deallocate all structures it has allocated during **init**.

TERO Driver Initialization

The first step is to register the driver for that specific *vendor* and *product id*. To this purpose, a structure must be created, initialized and linked with the help of ad-hoc macros:

```
static struct usb_device_id id_table [] = {
    { USB_DEVICE(PUF_VENDOR_ID, PUF_PRODUCT_ID) },
    { }
};
MODULE_DEVICE_TABLE(usb, id_table);
```

Then the *driver structure* is created, containing pointers to the two functions *probe* and *disconnect*. These will be invoked when a matching usb device is connected (**probe**) to the PC or disconnected (**disconnect**).

```
static struct usb_driver puf_driver = {
    .name = "tero_puf_driver",
    .id_table = id_table,
    .probe = puf_probe,
    .disconnect = puf_disconnect,
};
```

See https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

TERO Driver Initialization

In order exploit the USB subsystem, in the **init** method we must call the ad-hoc hook function:

usb_register(&puf_driver);

where *puf_driver* is a pointer for the previous structure (returns 0 if succeded).

Symmetrically, when the module is unloaded (and **exit** function is called), the function to call is:

usb_deregister(&puf_driver);

See https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

TERO Driver Initialization

When a device is plugged into the USB bus, and matches the device ID pattern that the driver registered within the USB core, the driver's probe function is called.

The driver now needs to verify that this device is actually one that it can accept. If so, the probe function returns 0. If not, or if any error occurs during initialization, an errorcode (such as *-ENOMEM* or *-ENODEV*) is returned.
As specified in the structure documentation in the kernel code:

*<< The probe() and disconnect() methods are **called in a context where they can sleep, but they should avoid abusing the privilege**. Most work to connect to a device should be done when the device is opened, and undone at the last close. The disconnect code needs to address concurrency issues with respect to open() and close() methods, as well as forcing all pending I/O requests to complete (by unlinking them as necessary, and blocking until the unlinks complete) >>*

See https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

TERO Driver Initialization

In the *probe* function, the driver must allocate its structures to manage the connected device and verify the usb device is actually the TERO PUF. The FTDI chip on the Basys3 is in fact pretty common: it's not enough to use only the vendor and product id.

In order to allocate the structure, we make use of the malloc function of the kernel:

data = **kzalloc**(**sizeof**(puf_data_t), *GFP_KERNEL*);

where the flag *GFP_KERNEL* indicates the normal behavior (could be blocking).
If used in interrupt context, *GFP_ATOMIC* must be used.

The pointer to this structure is then associated with the interface, in order to retrieve it later, with:

usb_set_intfdata(interface, data);

See https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

TERO Driver Initialization

A reference to the usb structure of the device is also needed in order to communicate using URBs (see later). We get a reference and save it to the custom driver data, with:

```
data->udev = usb_get_dev(udev);
```

This helper function is used to inform the kernel we detain a reference to the object, and let it control its deallocation without causing crashes in the system (*using an internal reference counter*)

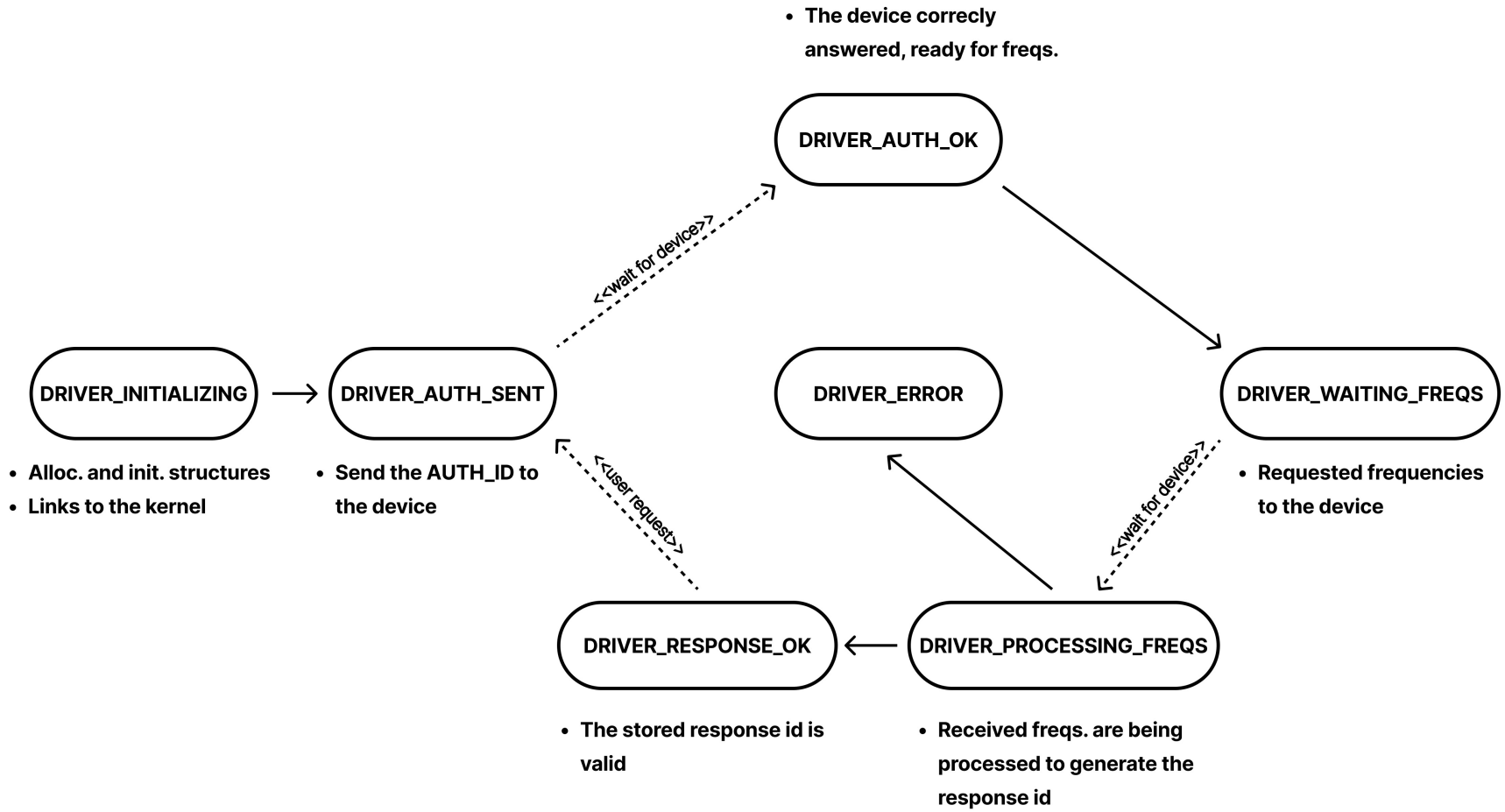
See https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

TERO Driver States

Description of the operational states of the driver once a device has been connected to the PC

DRIVER

The driver implements a finite state machine, to encode allowed operations during its lifetime.
In the next slides each state will be explained.



DRIVER_INITIALIZING

When the probe function is called, after a new device was detected, the driver allocates a structure specifically for this device. In this structure there are all the fields/references needed to communicate with the device and also to provide a character device specific for this PUF.

Before creating the character device, that will be visibile to userspace programs under **/dev**, the driver checks that this FTDI device is actually to a TERO PUF. To this purpose, configures the FTDI serial parameters, and then sends a byte of authentication:

0b10101010

The driver now enters the next state **DRIVER_AUTH_SENT**.

DRIVER_AUTH_SENT

The TERO PUF is expected to respond to this check with a known byte (**0b10101011**).
This must happen within *5s*, or the driver will refuse to link to this USB device.

To this purpose, after sending the auth byte, the driver performs a synchronous (blocking) receive request to the USB Subsystem, specifying the URB timeout. The kernel will return a failed URB if this timeout exceeds.

All these operations are done in the context of the probe function, blocking it for a while. This is allowed by the kernel, as previously stated. The kernel, if we return an error message from the probe method, will interpret that this driver does not recognize the USB device, and simply search for another driver.

If the device responds with the correct byte, the driver enters **DRIVER_AUTH_OKAY**, then the driver asks for frequencies in order to compute the response id for the first challenge (and enters **DRIVER_WAITING_FREQS**).

DRIVER_WAITING_FREQS

The driver waits for the frequencies to be received asynchronously.
This is achieved taking advantage the USB Subsystem features as previously specified.

Whenever data arrives, it's placed inside a FIFO (exploiting the data structure provided by the kernel). In this phase, the driver directly stores the arrived bytes, but does not perform any processing operation on them.
This is because the callback for the data arrived is called in interrupt context.

As soon as all the expected bytes are arrived, processing can be started. Because processing data can be expensive, it is scheduled as a kernel work. This is done using a macro for initializing the apposite structure (that must stored inside the driver data for that PUF, for reasons that can be seen later):

INIT_WORK(&data->worker_freqs, process_frequencies);

The work is then scheduled with the following call, and the driver changes state to **DRIVER_PROCESSING_FREQS**.

schedule_work(&data->worker_freqs);

To cancel the execution of a work, use the following method (blocking):

cancel_work_sync(&data->worker_freqs);

DRIVER_PROCESSING_FREQS

A kernel work must be defined with the following prototype:

void **process_frequencies**(struct work_struct *work)

where work is a pointer to the structure created with **INIT_WORK**. A custom pointer can be passed to the work, during the call to *INIT_WORK*. Here another approach is used, that seems to be pretty common in the kernel, and exploits the macro **container_of**.

This macro takes three arguments: the *pointer* (work in this case), the *parent structure type*, the *field name of the parent structure* where the first pointer is stored. It perform address calculations to return the address of beginning of the parent structure.
This is used to access the driver structure to store the new response id at the end of the processing.

Every access to the FIFO is protected via a spinlock with disable interrupts (as the FIFO is modified in interrupt context). *This is necessary as additional data could arrive to the driver from the device while the FIFO is being read, even if this should not happen if the TERO device works correctly.*

After the response is processed, the driver enters **DRIVER_RESPONSE_OK**.
If the device sends an incorrect amount of data, the driver enters **DRIVER_ERROR**, and it cannot recover.

See https://linux-kernel-labs.github.io/refs/heads/master/labs/deferred_work.html

DRIVER_RESPONSE_OK

Once the driver is once in this state, a valid response is present in the driver memory.
The driver keeps the last computed response with the respective challenge, in order to provide it to the user program upon request (from the character device, as shown in the next slides).

When a new challenge is requested, the driver repeats the exchange of data with the device.

See https://linux-kernel-labs.github.io/refs/heads/master/labs/deferred_work.html

Generating a Response

Description of the algorithm
used to generate a response id,
given a challenge

Mapping Challenge Number to Frequencies

In this TERO PUF, there are eight types of loops. This is because there are two different configurations of the loop in each Slice, and there are four adjacent columns of Slices used.

The current proposed algorithms compares loops of the same type to generate a response.
Having all 1280 frequencies available, they are grouped in 8 buckets.

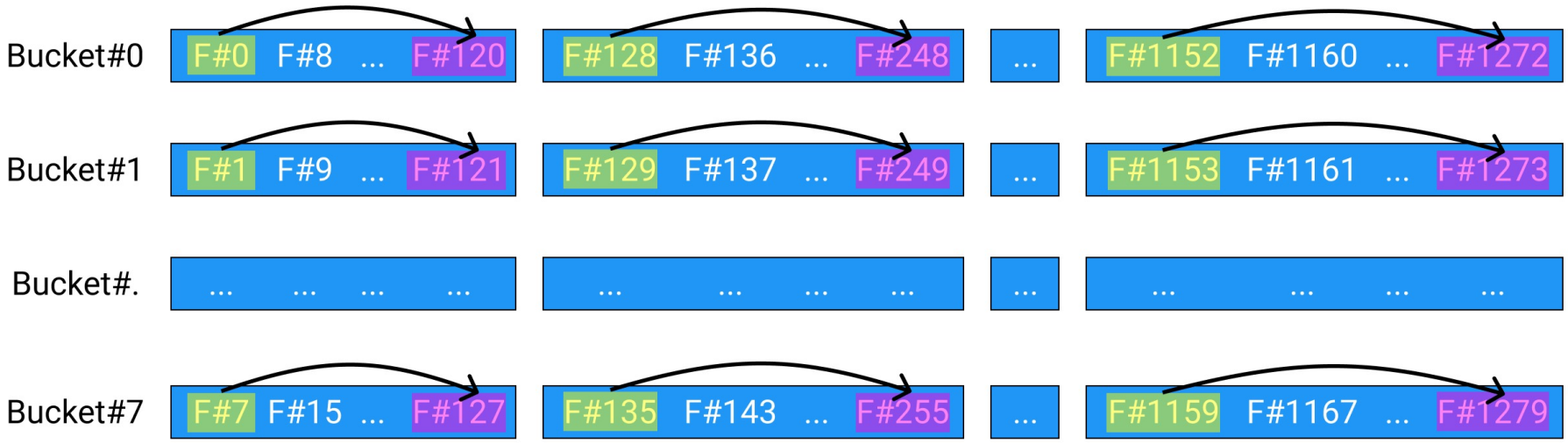
Mapping Challenge Number to Frequencies

Then each bucket is split in multiple columns (10 colums per bucket), in order to have 16 frequencies per column.

Bucket#0	F#0	F#8	...	F#120	F#128	F#136	...	F#248	...	F#1152	F#1160	...	F#1272
Bucket#1	F#1	F#9	...	F#121	F#129	F#137	...	F#249	...	F#1153	F#1161	...	F#1273
Bucket#2	F#2	F#10	...	F#122	F#130	F#138	...	F#250	...	F#1154	F#1162	...	F#1274
Bucket#3	F#3	F#11	...	F#123	F#131	F#139	...	F#251	...	F#1155	F#1163	...	F#1275
Bucket#4	F#4	F#12	...	F#124	F#132	F#140	...	F#252	...	F#1156	F#1164	...	F#1276
Bucket#5	F#5	F#13	...	F#125	F#133	F#141	...	F#253	...	F#1157	F#1165	...	F#1277
Bucket#6	F#6	F#14	...	F#126	F#134	F#142	...	F#254	...	F#1158	F#1166	...	F#1278
Bucket#7	F#7	F#15	...	F#127	F#135	F#143	...	F#255	...	F#1159	F#1167	...	F#1279

Mapping Challenge Number to Frequencies

In order to generate a response, taken *i,j* indices between 0 and 15, for each batch the two frequencies in position i and j are compared. If *freq[i] > freq[j]*, then the response bit is 1, otherwise 0.



Comparisons for i=0, j=15

Mapping Challenge Number to Frequencies

Having 80 batches, each composed of 16 frequency, in principle there are 256 possibilities to choose i and j (16*16).
Some of these are invalid as:

- i=j has no sense, will always give a 0 bit
- in freq[i] > freq[j], swapping i and j generate simply a negated response (totally correlated)

In the end only 120 challenges are meaningful (=(256-16)/2).

Mapping the indices i,j to the challenge number is not trivial: many possibilities exists.
From the communication perspective, the important part is to be able to transfer all and only the frequencies needed to compute the response.
This necessarily means the computation of the indices and the selecting algorithm must be implemented in hardware, so must be as simple as possible.

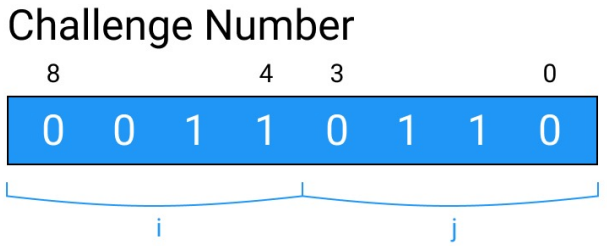
Two groups of frequencies can be identified from the slide before. The ones used in the left side of the comparison (pointed by i), and the ones in the right side (pointed by j).
Considering the indices:

- Left side: i*8 + 128*k + l with 0<=k<=9, 0<=l<=7
- Right side: j*8 + 128*k + l with 0<=k<=9, 0<=l<=7

These operation can be easily done in hardware, because they can be transformed in the form: (j<<3) + (k<<7) + l

Mapping Challenge Number to Frequencies

As i,j are between 0 and 15, 4 bits are enough to store them. Given 8 bits, it's possible to write them as an integer



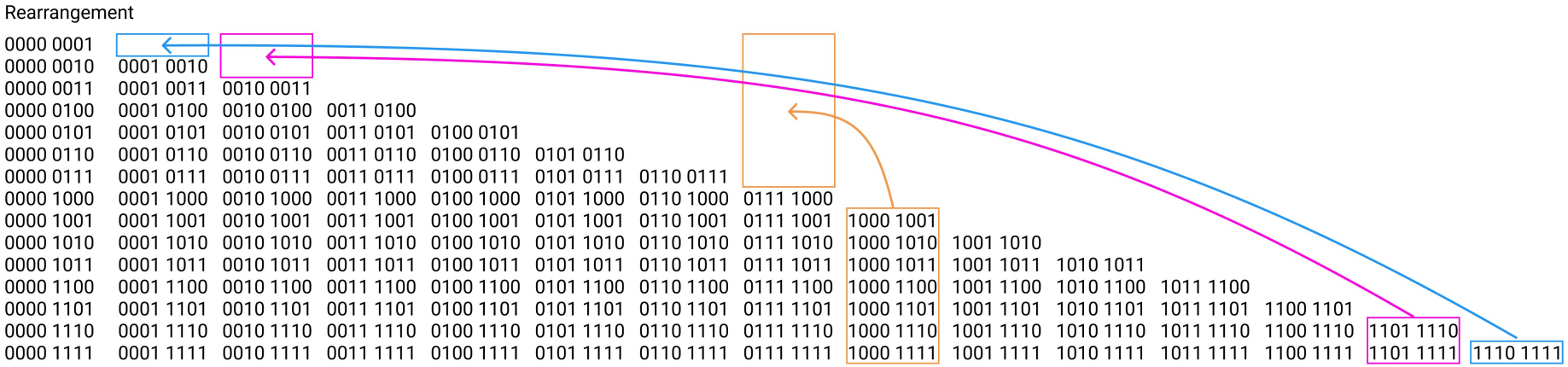
The only problem is that not all values of i,j are valid generated directly from the challenge number in this way.
In particular, i must be strictly less than j:

Valid Combinations

0000 0001														
0000 0010	0001 0010													
0000 0011	0001 0011	0010 0011												
0000 0100	0001 0100	0010 0100	0011 0100											
0000 0101	0001 0101	0010 0101	0011 0101	0100 0101										
0000 0110	0001 0110	0010 0110	0011 0110	0100 0110	0101 0110									
0000 0111	0001 0111	0010 0111	0011 0111	0100 0111	0101 0111	0110 0111								
0000 1000	0001 1000	0010 1000	0011 1000	0100 1000	0101 1000	0110 1000	0111 1000							
0000 1001	0001 1001	0010 1001	0011 1001	0100 1001	0101 1001	0110 1001	0111 1001	1000 1001						
0000 1010	0001 1010	0010 1010	0011 1010	0100 1010	0101 1010	0110 1010	0111 1010	1000 1010	1001 1010					
0000 1011	0001 1011	0010 1011	0011 1011	0100 1011	0101 1011	0110 1011	0111 1011	1000 1011	1001 1011	1010 1011				
0000 1100	0001 1100	0010 1100	0011 1100	0100 1100	0101 1100	0110 1100	0111 1100	1000 1100	1001 1100	1010 1100	1011 1100			
0000 1101	0001 1101	0010 1101	0011 1101	0100 1101	0101 1101	0110 1101	0111 1101	1000 1101	1001 1101	1010 1101	1011 1101	1100 1101		
0000 1110	0001 1110	0010 1110	0011 1110	0100 1110	0101 1110	0110 1110	0111 1110	1000 1110	1001 1110	1010 1110	1011 1110	1100 1110	1101 1110	
0000 1111	0001 1111	0010 1111	0011 1111	0100 1111	0101 1111	0110 1111	0111 1111	1000 1111	1001 1111	1010 1111	1011 1111	1100 1111	1101 1111	1110 1111

Mapping Challenge Number to Frequencies

The following rearrangement is performed:



In the end, the final disposition becomes:

i	j	Challenge Num														
0000 0001		0000 0000	1110 1111	0000 1111	1101 1110	0001 1110	1100 1101	0010 1101	1011 1100	0011 1100	1010 1011	0100 1011	1001 1010	0101 1010	1000 1001	0110 1001
0000 0010		0000 0001	0001 0010	0001 0000	1101 1111	0001 1111	1100 1110	0010 1110	1011 1101	0011 1101	1010 1100	0100 1100	1001 1011	0101 1011	1000 1010	0110 1010
0000 0011		0000 0010	0001 0011	0001 0001	0010 0011	0010 0000	1100 1111	0010 1111	1011 1110	0011 1110	1010 1101	0100 1101	1001 1100	0101 1100	1000 1011	0110 1011
0000 0100		0000 0011	0001 0100	0001 0010	0010 0100	0010 0001	0011 0100	0011 0000	1011 1111	0011 1111	1010 1110	0100 1110	1001 1101	0101 1101	1000 1100	0110 1100
0000 0101		0000 0100	0001 0101	0001 0011	0010 0101	0010 0010	0011 0101	0011 0001	0100 0101	0100 0000	1010 1111	0100 1111	1001 1110	0101 1110	1000 1101	0110 1101
0000 0110		0000 0101	0001 0110	0001 0100	0010 0110	0010 0011	0011 0110	0011 0010	0100 0110	0100 0001	0101 0110	0101 0000	1001 1111	0101 1111	1000 1110	0110 1110
0000 0111		0000 0110	0001 0111	0001 0101	0010 0111	0010 0100	0011 0111	0011 0011	0100 0111	0100 0010	0101 0111	0101 0001	0110 0111	0110 0000	1000 1111	0110 1111
0000 1000		0000 0111	0001 1000	0001 0110	0010 1000	0010 0101	0011 1000	0011 0100	0100 1000	0100 0011	0101 1000	0101 0010	0110 1000	0110 0001	0111 1000	0111 0000
0000 1001		0000 1000	0001 1001	0001 0111	0010 1001	0010 0110	0011 1001	0011 0101	0100 1001	0100 0100	0101 1001	0101 0011	0110 1001	0110 0010	0111 1001	0111 0001
0000 1010		0000 1001	0001 1010	0001 1000	0010 1010	0010 0111	0011 1010	0011 0110	0100 1010	0100 0101	0101 1010	0101 0100	0110 1010	0110 0011	0111 1010	0111 0010
0000 1011		0000 1010	0001 1011	0001 1001	0010 1011	0010 1000	0011 1011	0011 0111	0100 1011	0100 0110	0101 1011	0101 0101	0110 1011	0110 0100	0111 1011	0111 0011
0000 1100		0000 1011	0001 1100	0001 1010	0010 1100	0010 1001	0011 1100	0011 1000	0100 1100	0100 0111	0101 1100	0101 0110	0110 1100	0110 0101	0111 1100	0111 0100
0000 1101		0000 1100	0001 1101	0001 1011	0010 1101	0010 1010	0011 1101	0011 1001	0100 1101	0100 1000	0101 1101	0101 0111	0110 1101	0110 0110	0111 1101	0111 0101
0000 1110		0000 1101	0001 1110	0001 1100	0010 1110	0010 1011	0011 1110	0011 1010	0100 1110	0100 1001	0101 1110	0101 1000	0110 1110	0110 0111	0111 1110	0111 0110
0000 1111		0000 1110	0001 1111	0001 1101	0010 1111	0010 1100	0011 1111	0011 1011	0100 1111	0100 1010	0101 1111	0101 1001	0110 1111	0110 1000	0111 1111	0111 0111

Mapping Challenge Number to Frequencies

With this disposition, is possible to calculate i,j as:

```
a = challenge_num >> 4
b = challenge_num & 0xF
c = a + 1
d = b + c
if d & 0x10:
    i = (~c) & 0xF
    j = b
else:
    i = a
    j = d & 0xF
```

From i,j it's easy to generate the indices of the frequencies using the previous equations and two counters: one for k, one for l.

The PUF is programmed to send first all the left sides of the comparison, then the right sides.
The driver must simply receive this data, then perform the comparison.

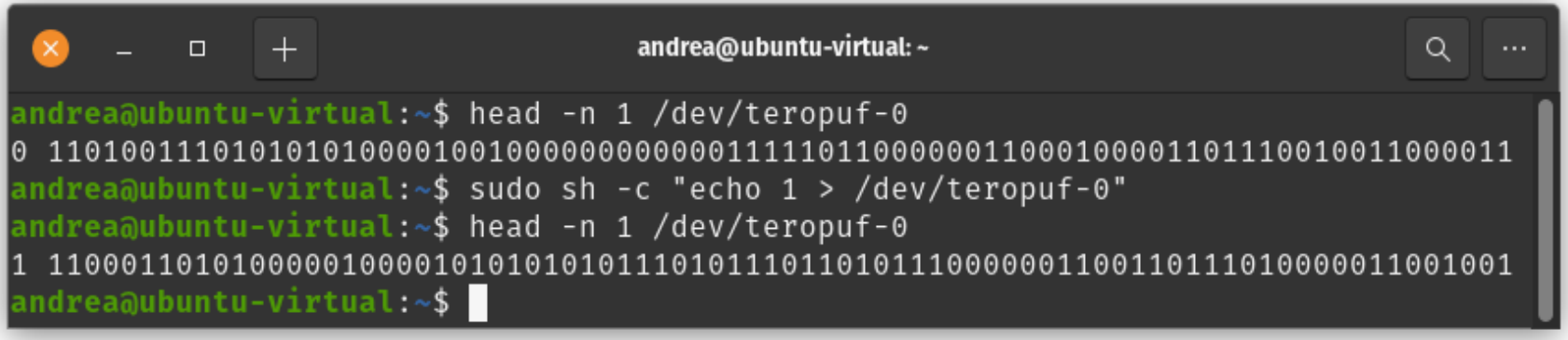
Interacting with the Driver

Description of the way an application can interact with the driver

Character Device

In Linux, devices are integrated into the system as special files.
The driver creates a new file under **/dev** for each PUF connected to the PC, that behaves as a character device.

The character device is the simplest available in Linux (the other is the block device).
An application can write in this special file the number of the challenge and by reading this file can get the response id and its associated challenge.



Character Device

To create a character device, the kernel needs an instance of the **struct file_operations**:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    [...]
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    [...]
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    [...]
}
```

*For the purpose of this driver only **open**, **release**, **read** and **write** will be defined.*

See https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

Character Device

The registration/deregistration of a device is made by specifying the major and minor.
First a region of devices must be created with the following method, by specifying the maximum number of device supported and the base name:

int register_chrdev_region (**dev_t** from, **unsigned** count, **const char *** name);

As soon as a new device connects, a new character device is requested using:

void cdev_init (**struct cdev *** cdev, **const struct file_operations *** fops);

int cdev_add (**struct cdev *** p, **dev_t** dev, **unsigned** count);

See https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

Character Device

To create the file under */dev*, device classes are needed.
In order to exploit this functionality, first initialize once the class, saving the returned instance:

```
struct class * __class_create(struct module * owner,
const char * name, struct lock_class_key * key);
```

Permissions for this device can be set by associating a custom function to the returned **class->devnode**:

```
static char *tero_devnode(struct device *dev, umode_t *mode) {
    if (!mode)
        return NULL;
    *mode = 0664;
    return NULL;
}
```

where *mode* is the standard permission mask of Linux.

See https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

Character Device

Then after any call to **cdev_add**, call the following function:

```
struct device * device_create(struct class * class,
struct device * parent, dev_t devt, void * drvdata,
const char * fmt, ...);
```

To remove a device, call:

void device_destroy (**struct class *** class, **dev_t** devt);

void cdev_del (**struct cdev *** p);

When the driver is unloaded, also the character region must be removed with:

void class_destroy (**struct class *** class);

void unregister_chrdev_region (**dev_t** from, **unsigned** count);

See https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

Character Device

If the pointer to the structure **dev_t** is saved inside the driver structure created during the **probe**, then it can be easily recovered with the macro *container_of* during the **open** method:

```
static int tero_open(struct inode *inode, struct file *file) {
    puf_data_t* data;
    // Retrieve puf context (as cdev is a variable of the structure)
    data = container_of(inode->i_cdev, puf_data_t, cdev);
    // Ref. the structure for easier access in the other methods
    file->private_data = data;
....
```

Saving the reference inside *private_data,* helps to easily recover it during the **read** and **write** method.

```
static ssize_t tero_read(struct file *file, char __user *user_buffer, size_t size, loff_t *offset) {
    puf_data_t* data;
    ...
    data = (puf_data_t*)file->private_data;
....
```

See https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

Character Device

At the moment, the driver does not support seeking.
This means that with standard library functions (i.e. fopen, fread, fclose), the device file must be opened, read and closed before being able to read the next challenge.

When reading from the device, at least 86 bytes must be requested. This constraint avoids reading by error a partial id.
The driver will return data in the format seen at slide 58:

<challenge_num> <challenge_response>\n\0

The terminator will be included in the count of the total bytes read.

When writing to the device, the application must output only a single integer (between 0 and 119) in ASCII encoding.
If it's a valid challenge number, the driver will begin generating the corresponding response id. The correctness of the operation can be deduced from the number of bytes written (> 0 means success).

While the driver is busy computing a new response id, applications can still read the device: the last generated response id with the corresponding challenge will be supplied. As soon as the new response id is ready, it will be supplied starting from the next read.

Installing the Driver

Description of the steps needed to install the driver in the system

Installing the Driver

This driver is in conflict with the legacy **ftdi_sio**. Whichever takes the device first will manage the device.
To avoid problems, the legacy driver should be blacklisted, by adding `*blacklist ftdi_sio*` to the file `*/etc/modprobe.d/blacklist.conf*`.

Currently, there is a few seconds of delay between when the board is powered up and the completion of the programming of the FPGA using the flash memory (green light on). If the device is detected by the driver before this period, the driver will not recognise the device. This can be solved by waiting these seconds in the driver.

To load the driver, navigate to the folder of the driver, then issue:

make all
make load

In order to unload the driver, issue:

make unload

To load and unload the driver, root privileges are needed.
On EFI systems with Secure Boot enabled, the driver must be signed. This can be done by using the commands already available in the Makefile.