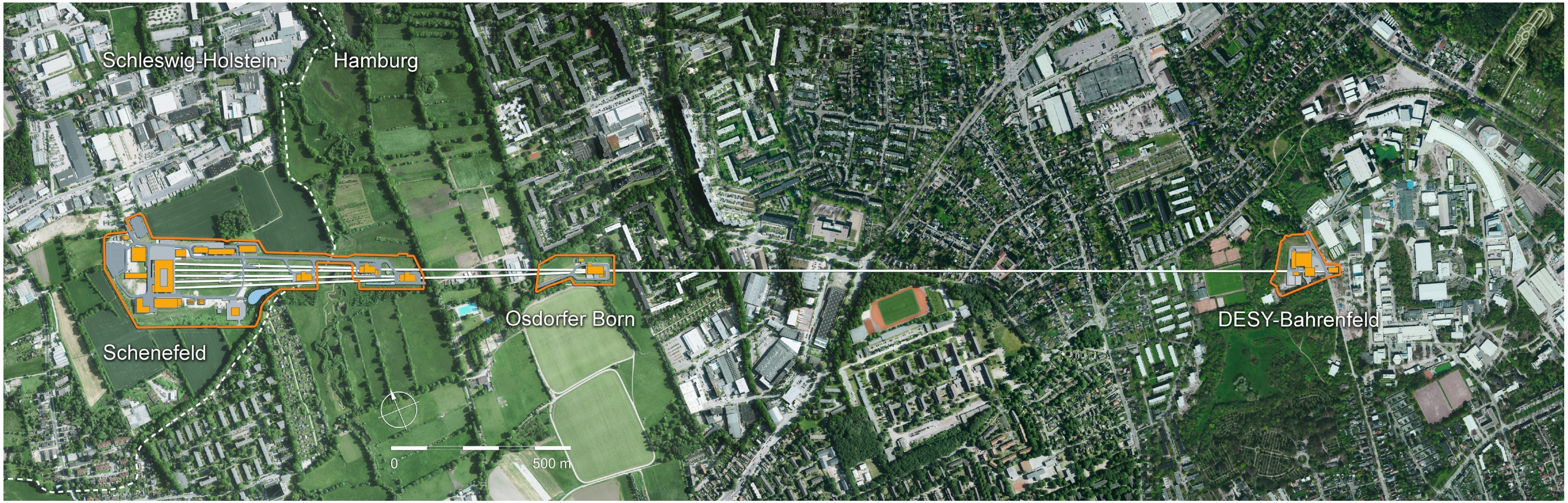
Using eBPF within your Python program using EBPFCat

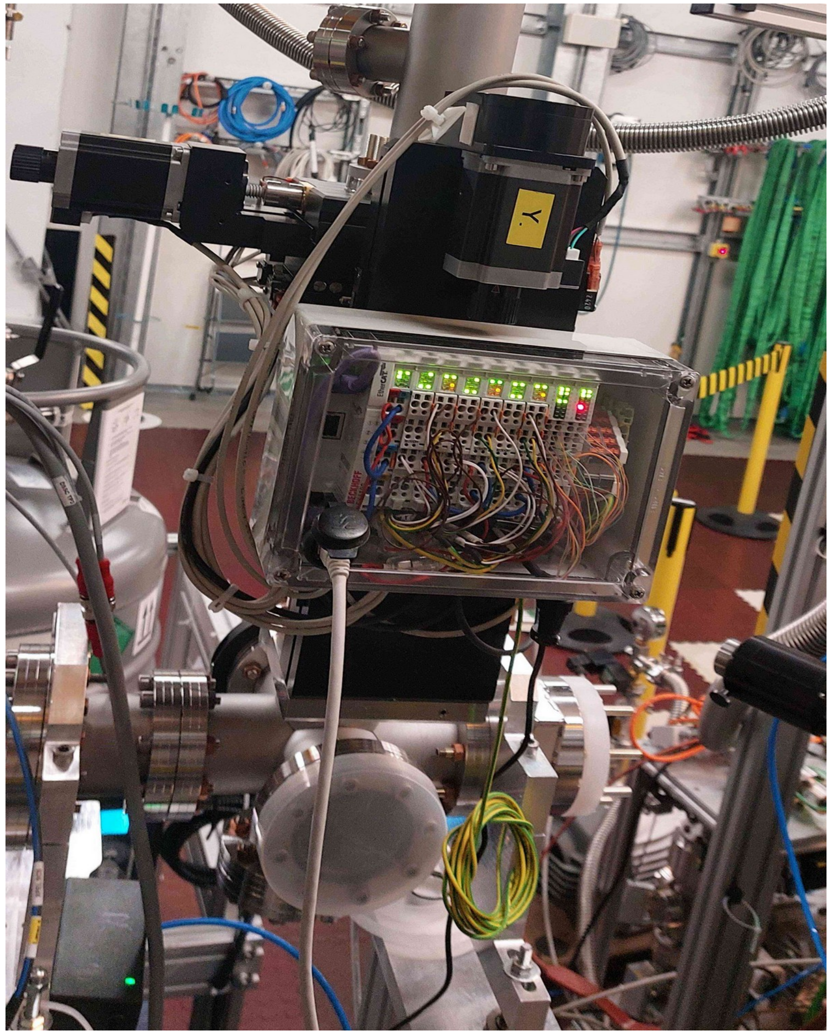
Martin Teichmann
Spectroscopy and Coherent Scattering (SCS) Instrument
European XFEL

The world’s largest X-Ray laser



Publicly funded research facility
Located in Hamburg, Germany

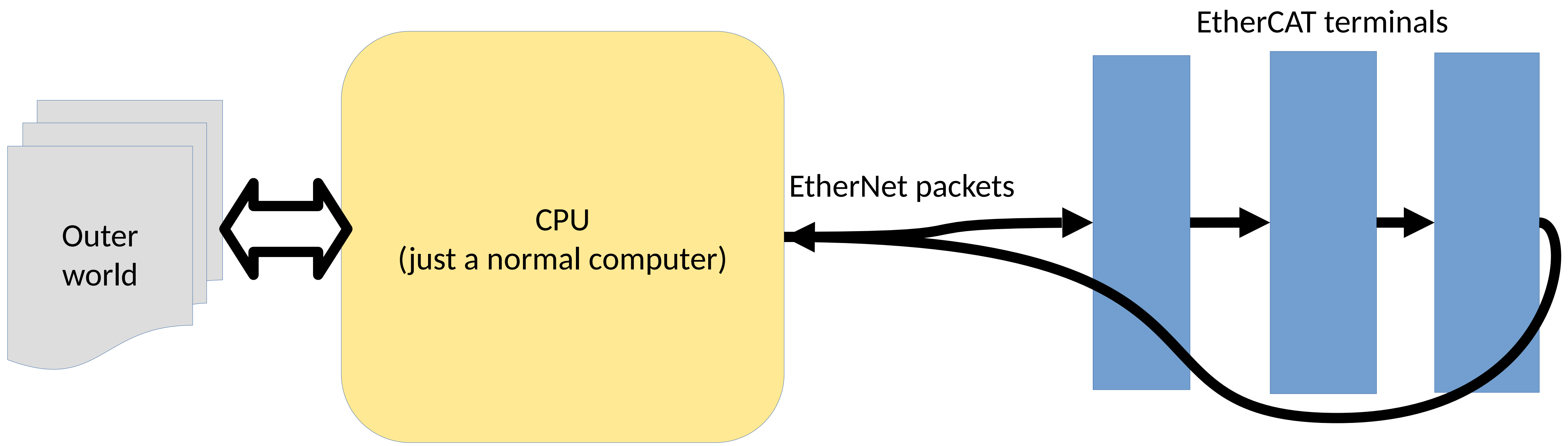
The Electron Birefringent Polarization Focus (eBPF)



- In our large scientific facility, many small machines need to be automated
- The automation is often done by scientists, not engineers
- The EtherCAT bus system used at the European XFEL is flexible enough to allow for this
- The software, however, is not. And it is not open source.

=> write a open source EtherCAT software driver flexible enough to used by users directly, while retaining all advantages of EtherCAT

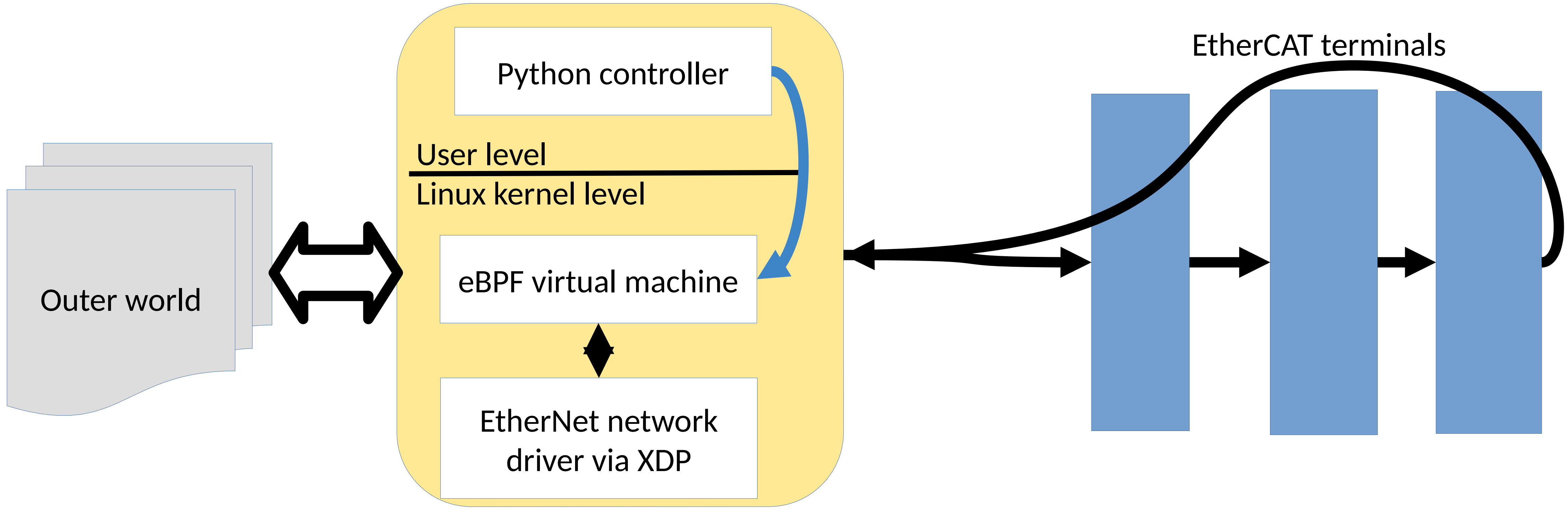
How does EtherCAT work?



EtherCAT uses normal EtherNET packets, which are handed from terminal to terminal, at the end returning at the beginning.

The terminals are able to modify the packet on-the-fly in order to send data to the computer

eBPF-based control



- The EBPF generator in EBPFCat may also be used for other EBPF use cases

We can inject a computer control loop directly into the kernel.

Topic of this talk

- How to use EBPFCat for XDP programs with simple examples
  - (XDP is the eBPF program type for processing network packets)
- How it actually works
- Going through some eBPF features already supported
- Show EtherCat motion as a cool application for eBPF

Constraints

- **Highly Dynamic**
  - The layout of the EtherNet packets is only known at run time
  - Even the necessary logic may only be discovered at run time
  - several eBPF programs run in parallel
  - The programs may be replaced at run time
- **Simplicity**
  - The programs are written by scientists with little programming experience
- **Flexibility**
  - The code should be generic, not limited to our particular problem
  - EBPFCat can be used as a speed boost for normal Python programs

A simple example

- Just count the number of packets arriving

- Code looks like normal Python code
- the Python code generates eBPF code: there is no compilation!

```
from asyncio import run, sleep
from ebpfcat.arraymap import ArrayMap
from ebpfcat.xdp import XDP, XDPExitCode

class Counter(XDP):
    license = "GPL"

    userspace = ArrayMap()
    count = userspace.globalVar()

    def program(self):
        self.count += 1
        self.exit(XDPExitCode.PASS)

async def main():
    c = Counter()

    async with c.run("eth0"):
        for i in range(10):
            await sleep(0.1)
            print("number of packets:", c.count)

if __name__ == "__main__":
    run(main())
```

A simple example

This generates eBPF {

```python
from asyncio import run, sleep
from ebpfcat.arraymap import ArrayMap
from ebpfcat.xdp import XDP, XDPExitCode

class Counter(XDP):
    license = "GPL"

    userspace = ArrayMap()
    count = userspace.globalVar()

    def program(self):
        self.count += 1
        self.exit(XDPExitCode.PASS)

async def main():
    c = Counter()

    async with c.run("eth0"):
        for i in range(10):
            await sleep(0.1)
            print("number of packets:", c.count)

if __name__ == "__main__":
    run(main())
```

A simple example

```
from asyncio import run, sleep
from ebpfcat.arraymap import ArrayMap
from ebpfcat.xdp import XDP, XDPExitCode

class Counter(XDP):
    license = "GPL"

    userspace = ArrayMap()
    count = userspace.globalVar()

    def program(self):
        self.count += 1
        self.exit(XDPExitCode.PASS)

async def main():
    c = Counter()

    async with c.run("eth0"):
        for i in range(10):
            await sleep(0.1)
            print("number of packets:", c.count)

if __name__ == "__main__":
    run(main())
```

same variable visible from eBPF and Python

Accessing the Packet

```
class Counter(XDP):
    minimumPacketSize = 16   # keep the eBPF verifier happy
    etherType = PacketVar(12, "!H")   # !H from Python struct package

    userspace = ArrayMap()
    count = userspace.globalVar()

    def program(self):
        with self.etherType == 0x800:   # always generate code: no if possible
            self.count += 1
        self.exit(XDPExitCode.PASS)
```

Accessing the Packet

```
class Counter(XDP):
    minimumPacketSize = 16   # keep the verifier happy
    etherType = PacketVar(12, "!H")

    userspace = ArrayMap()
    ipv4count = userspace.globalVar()
    ipv6count = userspace.globalVar()

    def program(self):
        with self.etherType == 0x800 as Else:
            self.ipv4count += 1
        with Else, self.etherType == 0x86DD:   # read as: elif
            self.ipv6count += 1
        self.exit(XDPExitCode.PASS)
```

How eBPF code is generated

```
class Counter(XDP):
    speed = PacketVar(12, "I")

    userspace = ArrayMap()
    max_speed = userspace.globalVar("i")

    def program(self):
        with self.speed > self.max_speed:
            self.speed = self.max_speed
```

using descriptors
with `__get__`
and `__set__`

$\longrightarrow$

```
1: r2 = *(u32 *)(r8 + 12)
2: r3 = *(i32 *)(r9 + 16)
3: if r3 <= r2 goto pc + 2
4: *(i32 *)(r9 + 16) = r2
```

Prerequisites

- EBPFCat has no dependencies
  - OK, technically Python 3, Linux and libc

So a simple

```
pip install ebpfcat
```

And you have a working setup

Permanent and local programs

eBPF programs used as local speedup within one Python program, with automatic cleanup at the end:

```
program = Program()
async with program.run("eth0"):
    ... # Python business logic
```

An eBPF program can also run independently from the running Python program, the starting program executes:

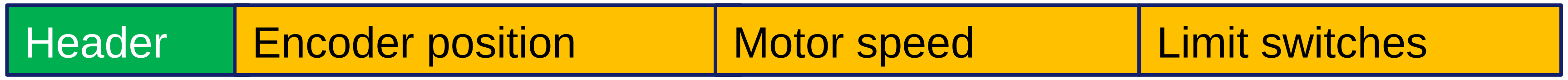
```
program = Program()
await program.attach("eth0")
program.pin_maps("/sys/fs/bpf/my_maps")
# some business logic
```

at a later time:

```
program = Program(load_maps="/sys/fs/bpf/my_maps")
# some more business logic
# and once finally done:
await program.detach("eth0")
```

Motion control with EtherCat

1. Read the self-description for all involved terminals
2. Prepare an EtherNet packet for all data to be transferred, e.g.:



3. Program the terminals to accept such a packet
4. Generate the EBPF program, and load into Linux kernel

```
with self.speed > self.max_speed:
    self.speed = self.max_speed
```

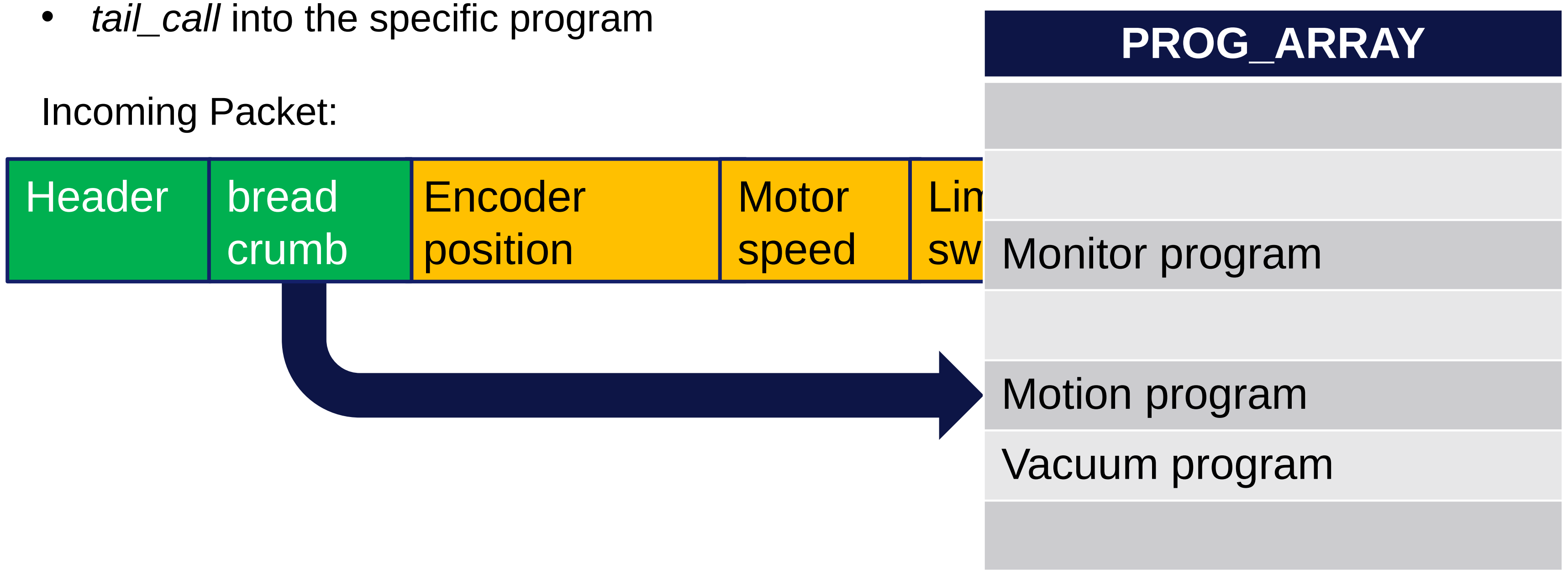
```
1: r2 = *(u32 *)(r1 + 12)
2: r3 = *(i32 *)(r9 + 16)
3: if r3 <= r2 goto pc + 2
4: *(i32 *)(r9 + 16) = r2
```

5. Send prepared packets => eBPF program will cyclically process them
6. Execute slow Python code for communication with outside world

dispatching on packet content

- A permanent program with a pinned PROG_ARRAY map
- Packets contain breadcrumbs for dispatching
- *tail_call* into the specific program

Incoming Packet:



A local program picks up its data

```
class Motion(XDP):
    ...
```

Programming a motor

Python allows us to write very high level code:

```
class Motor(Device):
    velocity = TerminalVar()  # communication with the hardware
    encoder = TerminalVar()

    target = DeviceVar()   # communication with the user
    proportional = DeviceVar()


    def program(self):
        self.velocity = (self.target – self.encoder) * self.proportional
```

Summary

- EBPFCat can generate eBPF code from Python
- It can be used generically to write eBPF programs, especially XDP programs
- As a special case it can be used for computer control of EtherCAT compatible hardware

Everything can be found at
`https://github.com/tecki/ebpfcat`

Documentation at
`https://ebpfcat.readthedocs.io/en/latest/`

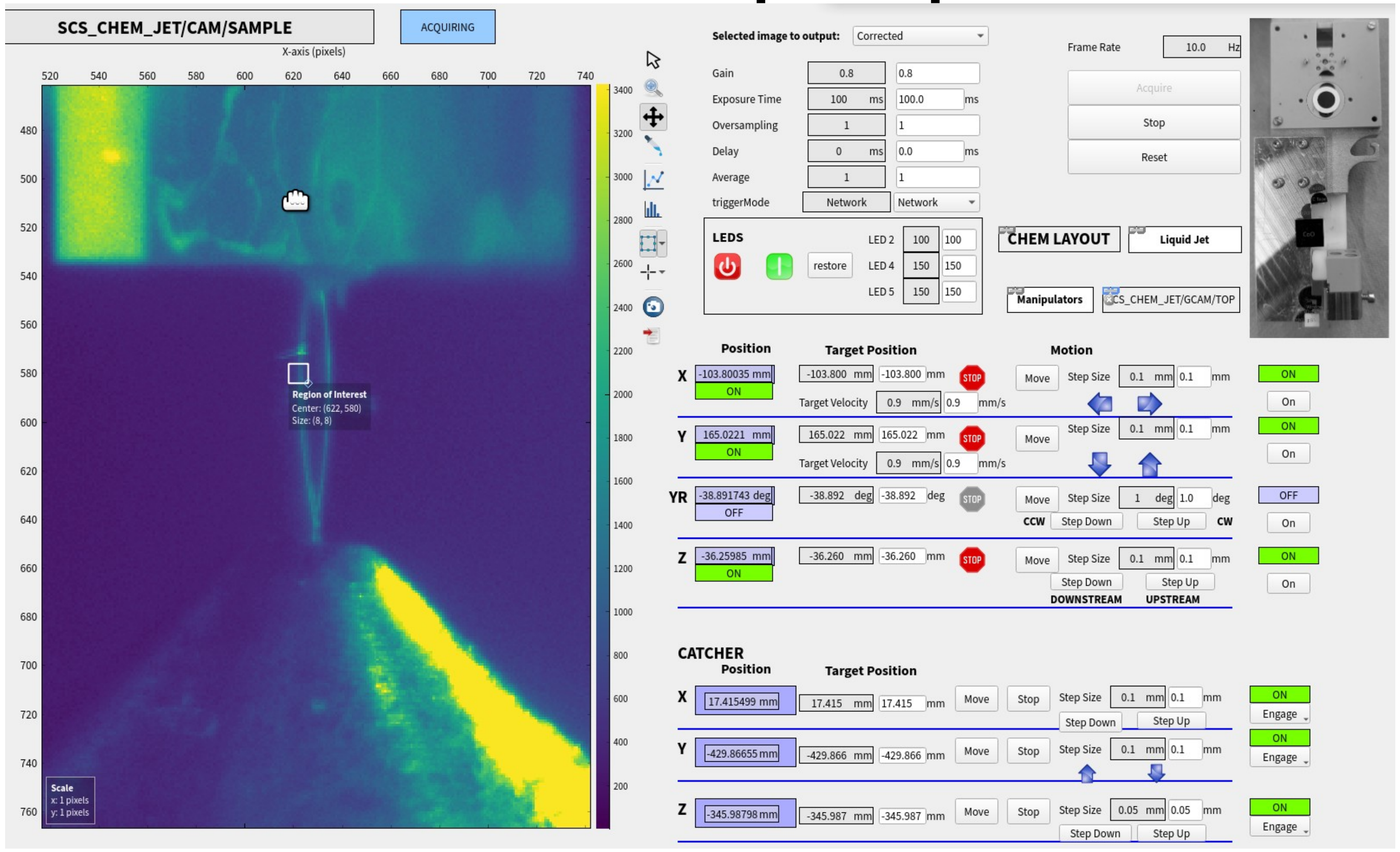
Bonus material for the really interested:
`https://git.xfel.eu/karaboDevices/karahoff`

Thanks to the SCS group at the European XFEL, Carsten Broers and Jan Tolkiehn for the hardware,
Tobias Freyermuth for useful discussions

Thank you!

Thanks to my group SCS at European XFEL

Example: parallel motion



Selected image to output: Corrected

Frame Rate 10.0 Hz

Gain	0.8	0.8
Exposure Time	100 ms	100.0 ms
Oversampling	1	1
Delay	0 ms	0.0 ms
Average	1	1
triggerMode	Network	Network

LEDS

LED 2	100	100
LED 4	150	150
LED 5	150	150

	Position	Target Position		Motion		
X	-103.80035 mm ON	-103.800 mm	-103.800 mm	Move	Step Size 0.1 mm 0.1 mm	ON / On
		Target Velocity 0.9 mm/s	0.9 mm/s			
Y	165.0221 mm ON	165.022 mm	165.022 mm	Move	Step Size 0.1 mm 0.1 mm	ON / On
		Target Velocity 0.9 mm/s	0.9 mm/s			
YR	-38.891743 deg OFF	-38.892 deg	-38.892 deg	Move	Step Size 1 deg 1.0 deg; CCW Step Down, Step Up CW	OFF / On
Z	-36.25985 mm ON	-36.260 mm	-36.260 mm	Move	Step Size 0.1 mm 0.1 mm; Step Down DOWNSTREAM, Step Up UPSTREAM	ON / On

CATCHER

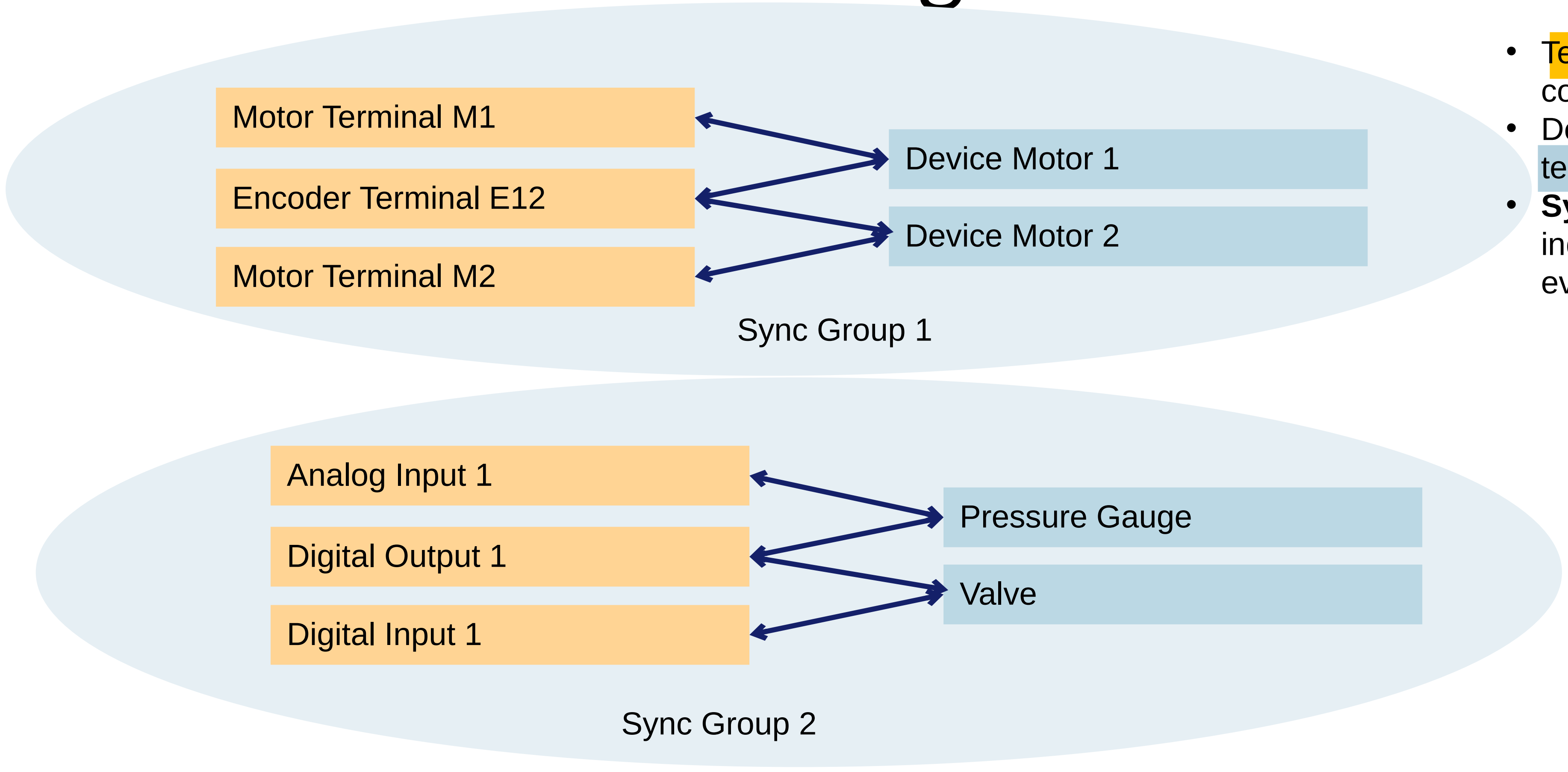
	Position	Target Position					
X	17.415499 mm	17.415 mm	17.415 mm	Move	Stop	Step Size 0.1 mm 0.1 mm; Step Down, Step Up	ON Engage
Y	-429.86655 mm	-429.866 mm	-429.866 mm	Move	Stop	Step Size 0.1 mm 0.1 mm	ON Engage
Z	-345.98798 mm	-345.987 mm	-345.987 mm	Move	Stop	Step Size 0.05 mm 0.05 mm; Step Down, Step Up	ON Engage

- A liquid jet is injected into a vacuum chamber, and pumped out by a catcher system
- The two systems are moved by independent motors, we lock the motor movement between them using EBPFCat
- Everything here is user-generated, no experts are needed

The Code for parallel motion

```
class MotorDevice(Device):
    velocity = TerminalVar()
    encoder = TerminalVar()
    low_switch = TerminalVar()
    high_switch = TerminalVar()
    enabled = TerminalVar()
    control_word = TerminalVar()
    reduced_current = TerminalVar()
    stepcounter = TerminalVar()
    encoder_error = TerminalVar()
    set_control_word = DeviceVar('H')
    deadband_exceeded = DeviceVar('b')
    max_velocity = DeviceVar('q')
    last_velocity = DeviceVar('q')
    max_acceleration = DeviceVar('Q')
    deadband = DeviceVar('q')
    target = DeviceVar('q')
    proportional = DeviceVar('q')
    lasttime = DeviceVar('Q')
    velocity_diff = LocalVar('q')

    def calculate_speed(self):
        self.ebpf.stmp = (self.proportional * self.ebpf.stmp) >> 16

    def program(self):
        with self.ebpf.tmp:
            self.ebpf.tmp = ktime(self.ebpf)
            self.velocity_diff = (self.max_acceleration
                                  * (self.ebpf.tmp - self.lasttime)) >> 32
            with self.velocity_diff > 0 as Else:
                self.lasttime = self.ebpf.tmp
        self.control_word = self.set_control_word
        with abs(self.target - self.encoder) > self.deadband as Else:
            if self.reduced_current is not None:
                self.reduced_current = False
            self.deadband_exceeded = 1
        if self.reduced_current is not None:
            with Else:
                self.reduced_current = True
        with self.ebpf.stmp:
            self.ebpf.stmp = self.target - self.encoder
            self.calculate_speed()
            with self.ebpf.stmp > self.max_velocity:
                self.ebpf.stmp = self.max_velocity
            with self.ebpf.stmp + self.max_velocity < 0:
                self.ebpf.stmp = -self.max_velocity
            if not self.isLimitless:
                with ((self.low_switch != 0) & (self.ebpf.stmp < 0)):
                    self.ebpf.stmp = 0
                with ((self.high_switch != 0) & (self.ebpf.stmp > 0)):
                    self.ebpf.stmp = 0
            if self.encoder_error is not None:
                with self.encoder_error:
                    self.ebpf.stmp = 0
            self.velocity = self.ebpf.stmp
            with self.ebpf.stmp > self.last_velocity + self.velocity_diff as Else:
                self.velocity = self.last_velocity + self.velocity_diff
            with Else, self.last_velocity > self.ebpf.stmp + self.velocity_diff:
                self.velocity = self.last_velocity - self.velocity_diff
            with self.enabled as Else:
                self.last_velocity = self.velocity
            with Else:
                self.last_velocity = 0

class FollowerDevice(MotorDevice):
    follow_offset = DeviceVar('q')
    doFollow = DeviceVar('B')

    def calculate_speed(self):
        super().calculate_speed()
        with (self.doFollow != 0) & (self.leader.device.enabled != 0):
            self.ebpf.stmp += (int(self.motors_factor * self.follow_factor
              * (1 << 16)) * self.leader.device.velocity) >> 16

    def program(self):
        with self.doFollow != 0:
            self.target = ((self.leader.device.encoder
                            * int(self.follow_factor * (1 << 16))) >> 16) \
                           + self.follow_offset
            with abs(self.target - self.encoder) > self.follow_lag:
                self.set_control_word = self.SWITCH_OFF
                self.doFollow = 0
                self.leader.device.set_control_word = \
                       self.leader.SWITCH_OFF
        super().program()
```

The building blocks of EBPFCat



- Terminals are the hardware actually connected to the EtherCAT bus
- Devices are hardware controlled by the terminals that logically belong together
- **Sync Groups** are units that run independently - even in different tasks so even their code can be modified on-the-fly