The CARIBOu 2.0 system

Mathieu Benoit

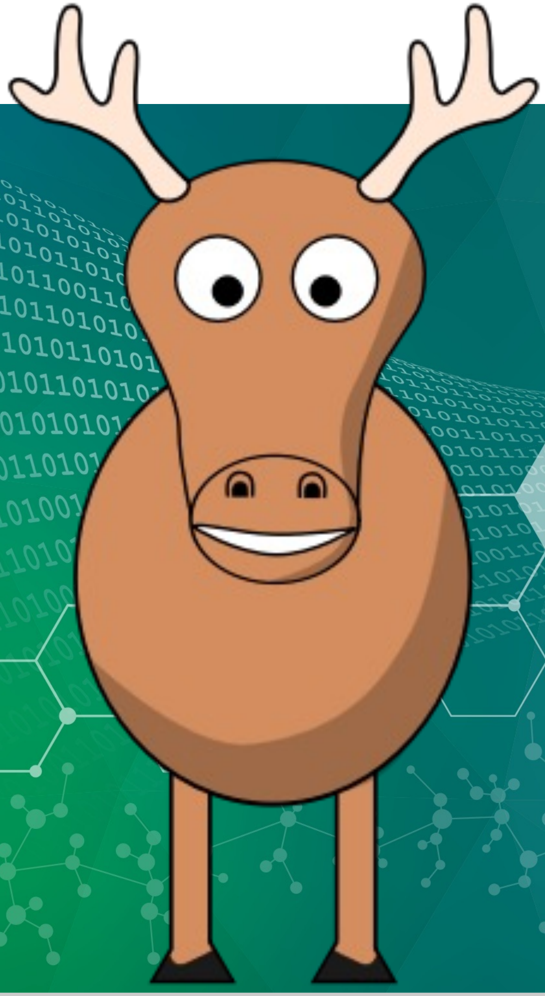
Oak Ridge National Laboratory

S. Tang, M. Begel, H. Chen, T. Liu, D. Matakias, H. Xu, E. Zhivun

Brookhaven National Laboratory

Max Pijacki, Thomas Koffas

Carleton University



ORNL is managed by UT-Battelle LLC for the US Department of Energy

Outline

- The CARIBOu 1.x system
  - Purpose and concept
  - Example of use cases
    - Software integration
    - Firmware
- The CARIBOu 2.0 system
  - Design plan summary
  - Firmware and software
- Conclusion and perspectives

The CARIBOu Framework

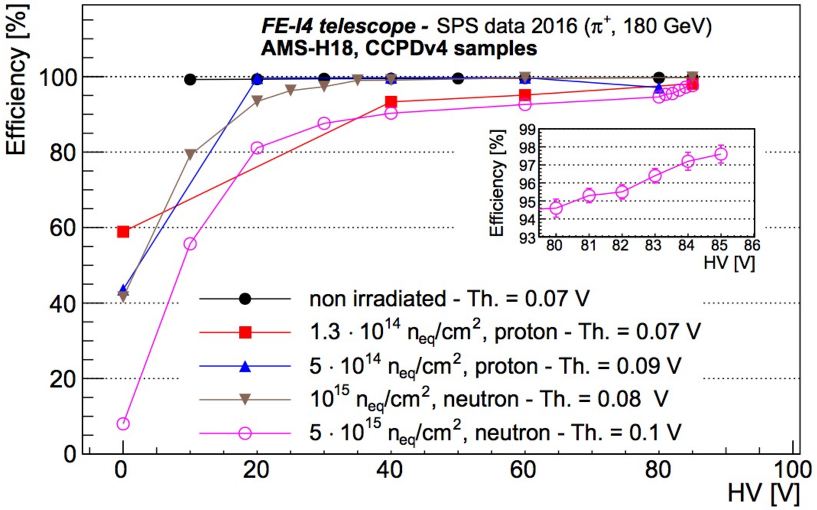
The CaRIBOu framework was originally developed in 2014, in collaboration between BNL (HW and Felix integration), UNIGe, and CERN (FW and SW design) as a versatile platform for DAQ development of our prototypes

- Many chips produced during R&D with similar needs in terms of biases, power, and data transmission. To reduce the development time, the framework is designed to provide re-usable software, firmware, and hardware that minimize integration effort and reduce time to first data acquisition
- The accompanying Peary software provides an easy, user-friendly abstraction layer to hardware and facilitates code re-use and sharing of resources (IPs, intermediate boards, etc.)
- The system was extensively used for the early CMOS sensor characterization, including the first prototype to demonstrate radiation hardness beyond 1e15 n_{eq}/cm^2
  - Collaboration expanded to ANL, Bern, CERN, Liverpool

Development of FELIX based readout system for HV-CMOS sensor testbeam, M. Benoit et al., J. Inst. Vol 14 issue 01 (2019), DOI 10.1088/1748-0221/14/01/P01013

Development of a modular test system for the silicon sensor R&D of the ATLAS Upgrade, M. Benoit et al., J. Inst. Vol 12 issue 01 (2017), DOI 10.1088/1748-0221/12/01/P01008

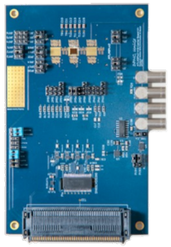
Testbeam results of irradiated ams H18 HV-CMOS pixel sensor prototypes, M. Benoit et al., J. Inst. Vol 13 (2018), arXiv:1611.02669

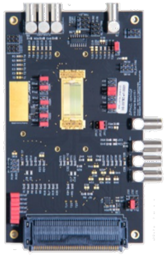


new prototypes

CLICTD

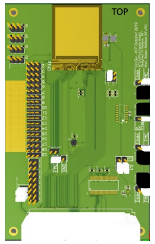
CLICpix2, C3PD


ATLASPix2

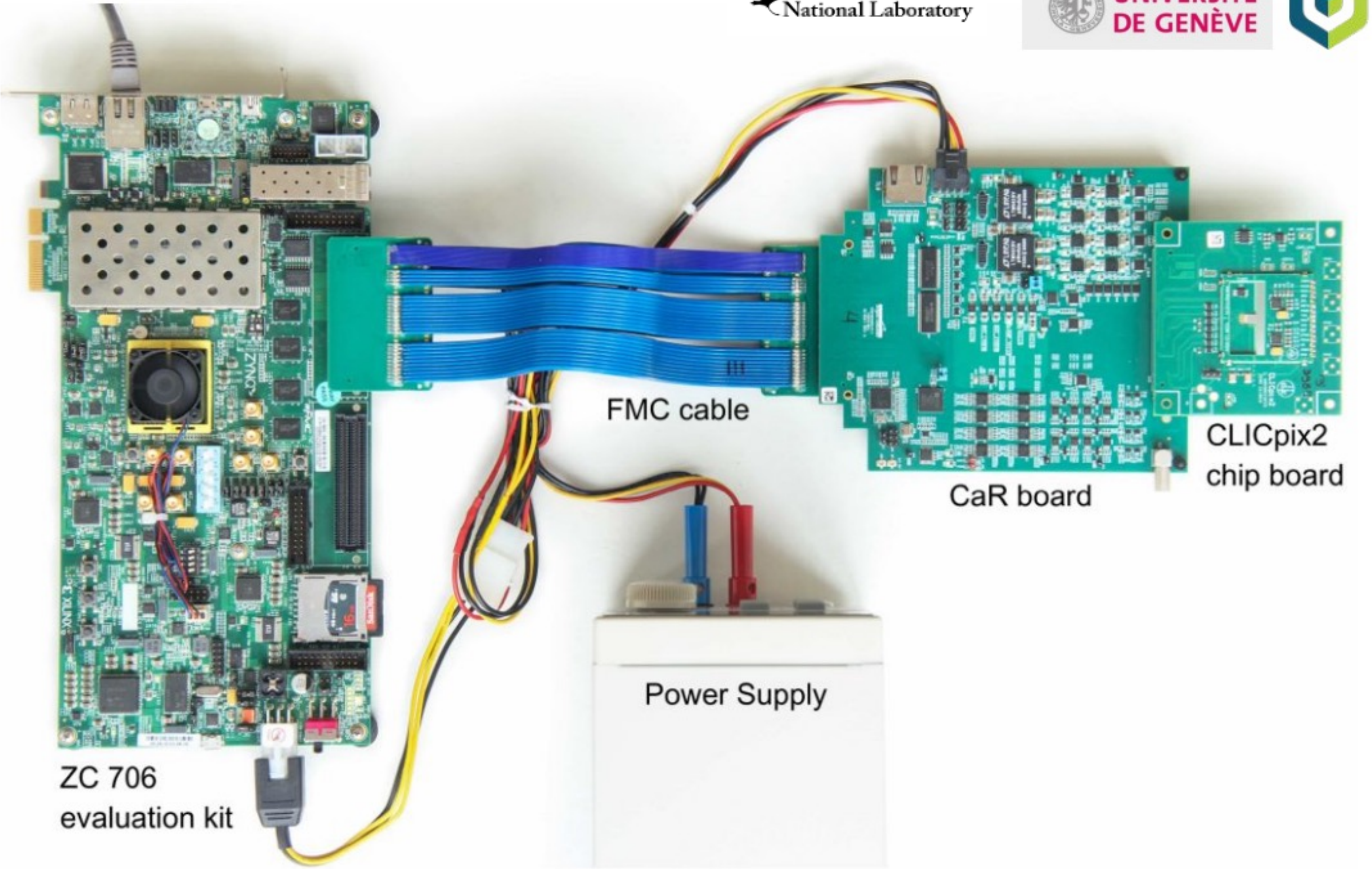

ATLASPix

H35Demo/FEI4

RD50-MPW1/2


ATLASPix3

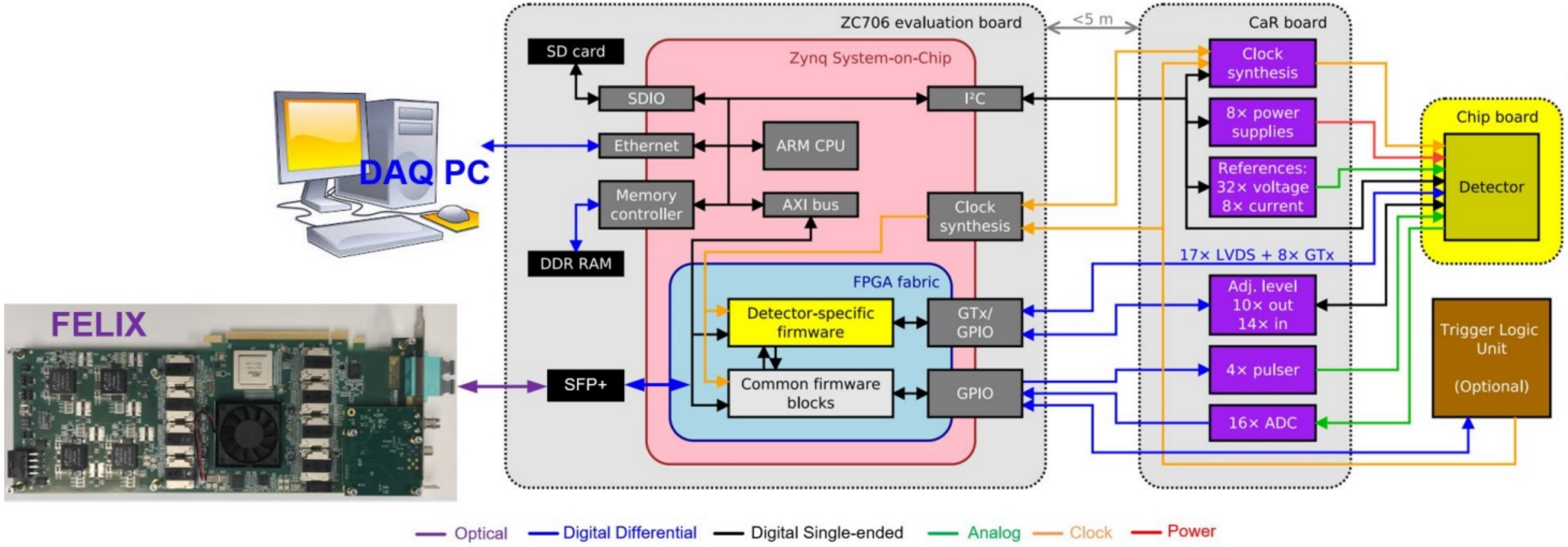
The CARIBOu system : Hardware



The current CaRIBOu DAQ system consists of:

- A Xilinx ZYNQ-7000 SoC FPGA based **ZC706** evaluation board
- A Control and Readout (**CaR**) board
- An Application-specific detector carrier board

CaRIBOu System - Block Diagram



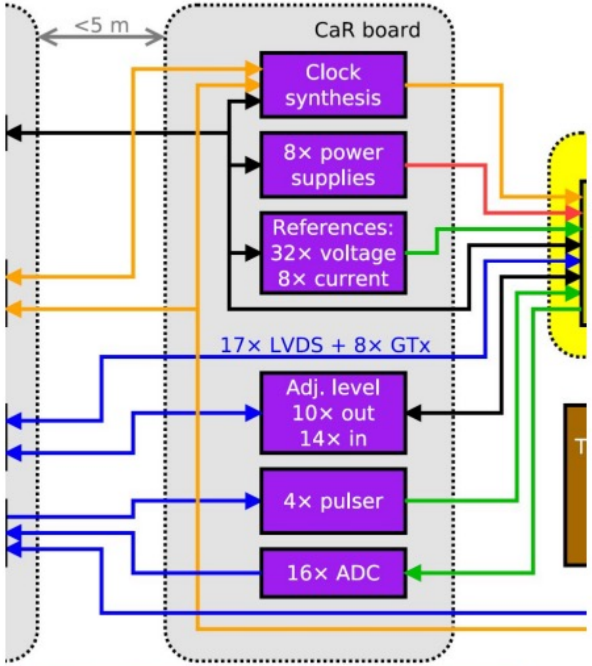
The block diagram of CaRIBOu DAQ system is shown on the left:

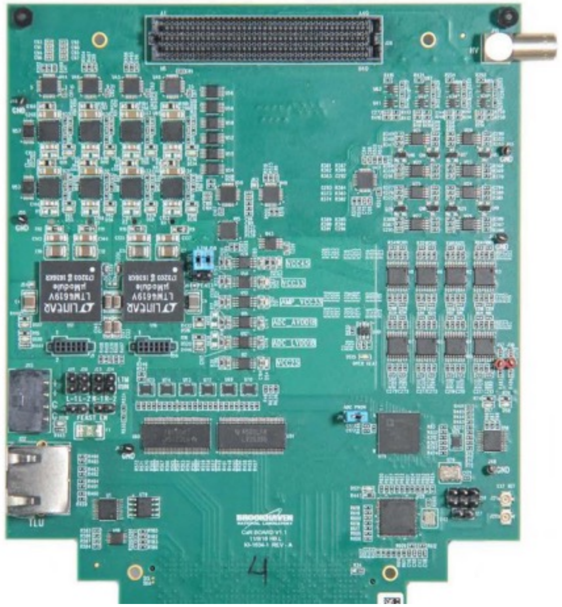
- ZC706 provides the CPU/Ethernet/SD/DDR4/MMCM/GPIO/I2C/GTX
- It also has the SFP+ optical link to **FELIX**
- FMC cable between ZC706 and CaR
- 320 pin SEAF connector for chip board

CaR Board

The CaR board has below features to support the different chips/ASICs

- 8 adjustable power supplies with monitoring
- 8-input 12-bit ADC with 50 kSamples/s (I2C interface)
- 16-input 14-bit ADC with 65 MSamples/s (parallel interface)
- 4 injection pulsers with adjustable pulse amplitude
- 32 adjustable voltage references
- 8 adjustable current references
- 10 output and 14 input single-ended links with adjustable voltage level
- 17 bidirectional LVDS links (up to 1.1 Gb/s)
- 8 full-duplex high-speed MGT links (0.8-12 Gb/s)
- programmable clock generator with external reference input
- interface for trigger and time reference (3 LVDS I/O + 1 LVDS clock)
- FMC connector to SoC platform

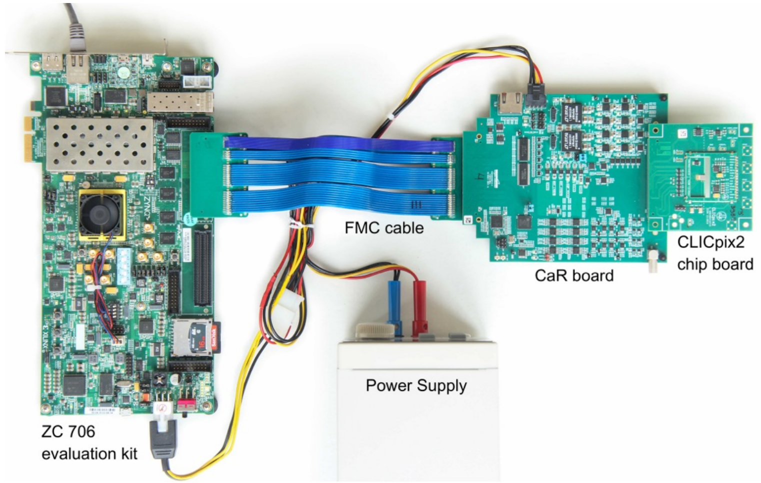




CaRIBOu (v1.x) Status

~ 50 systems in use!

- Current platform based on Xilinx ZC706 has been revised (v1.3) at BNL and distributed to users for ongoing projects
  - Software and example firmware provided to users for the new revision
  - Production organized in collaboration with CERN
- The platform is being used in multiple project for the control and readout of their pixel ASIC:
  - CLIC (CLICpix1/2,C3PD, ATLASPix(2-3),CLICTD, Fastpix)
  - RD50 (RD50Pix prototypes)
  - ATLAS (ATLASPix series, CCPD)
  - NASA/AMEGO (ASTROpix v1-2)



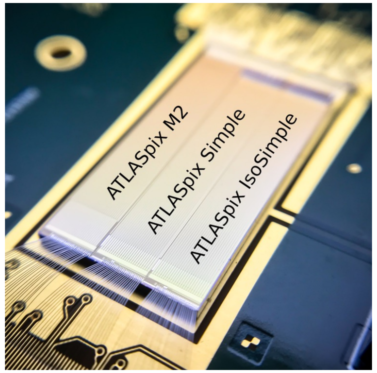
Test beam measurement of ams H35 HV-CMOS capacitively coupled pixel sensor prototypes with high-resistivity substrate, M. Benoit et al., J. Inst. Vol 13 (2018), arXiv:1712.08338

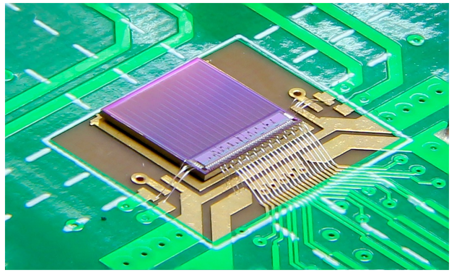
Developing the future of gamma-ray astrophysics with monolithic silicon pixels, I. Brewer et al., NIMA, 10.1016/j.nima.2021.165795

Readout system and testbeam results of the RD50-MPW2 HV-CMOS pixel chip, P. Sieberer et al, arXiv:2201.08585 [physics.ins-det]

Performance of CMOS pixel sensor prototypes in ams H35 and aH18 technology for the ATLAS ITk upgrade, M. Khien et al. NIMA (2019) DOI:10.1016/j.nima.2018.07.061

ASTROPIXv1

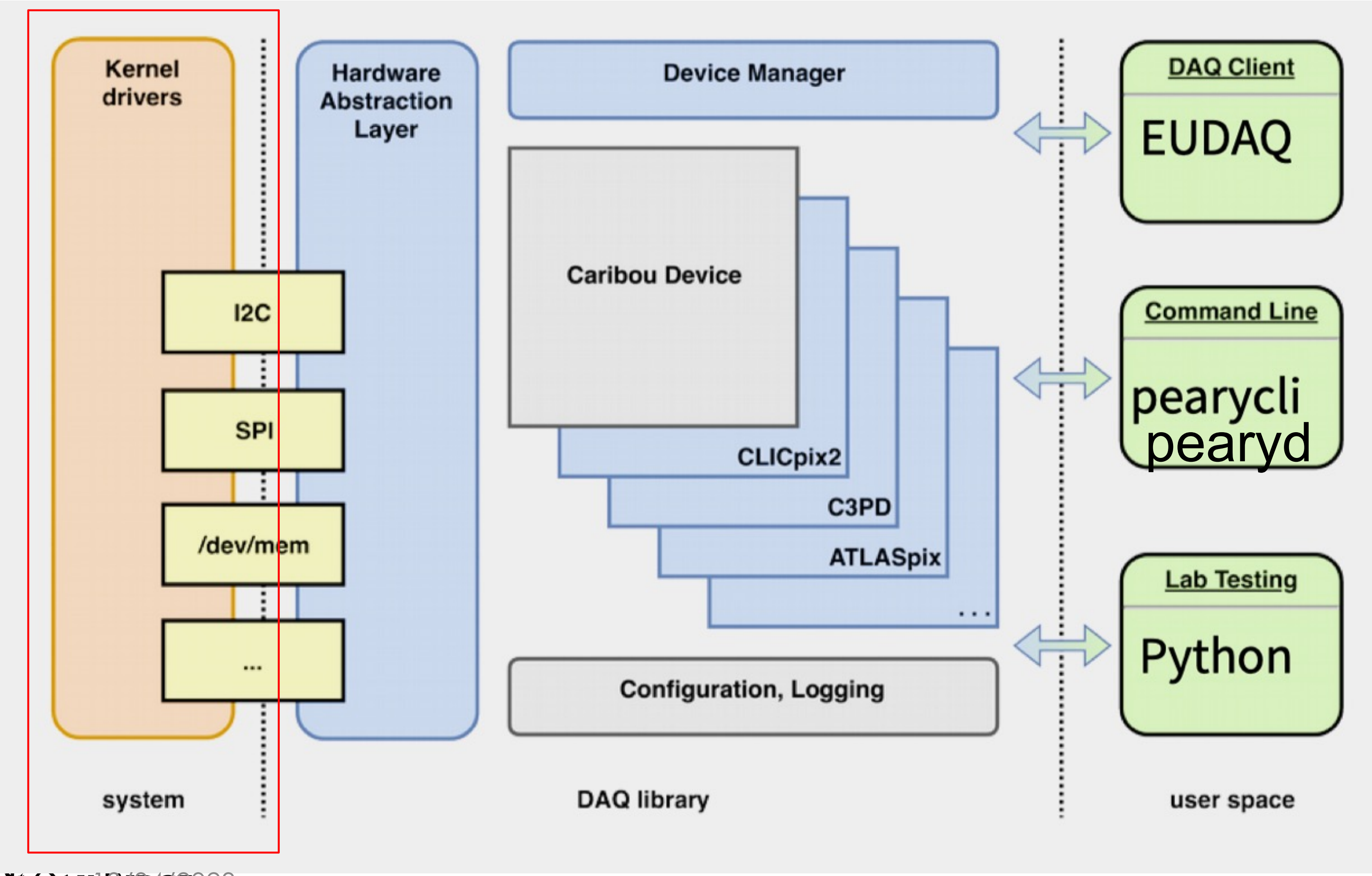




CLICTD

The Peary Framework

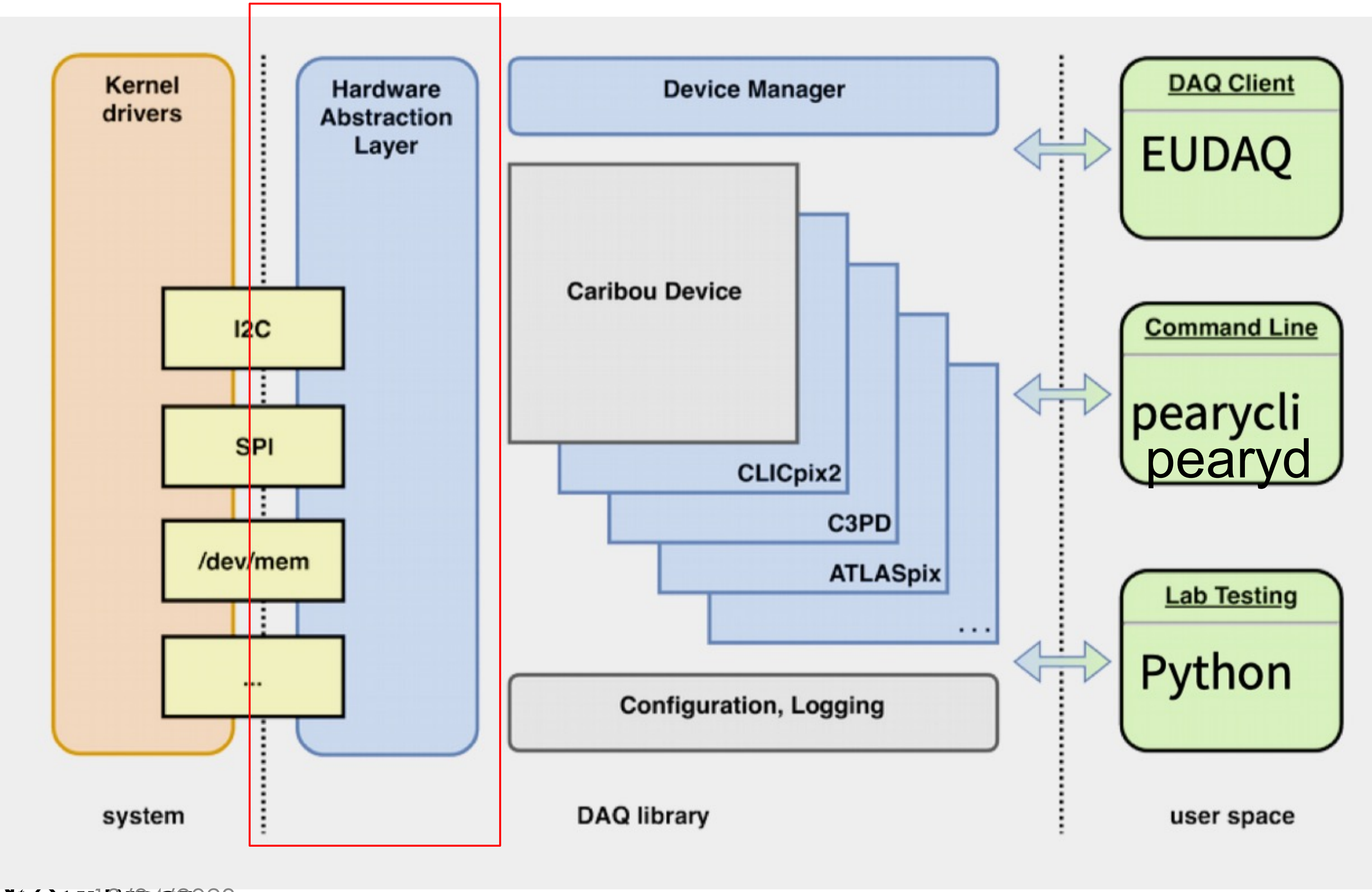
The software name derives from the smallest of the North American caribou subspecies, found in the high Arctic islands of Canada (Wikipedia).



Kernel drivers (meta-caribou)

- Interface to system
- Xilinx provided drivers for
  - I2C
  - SPI
  - Access to PL register space
  - Networking

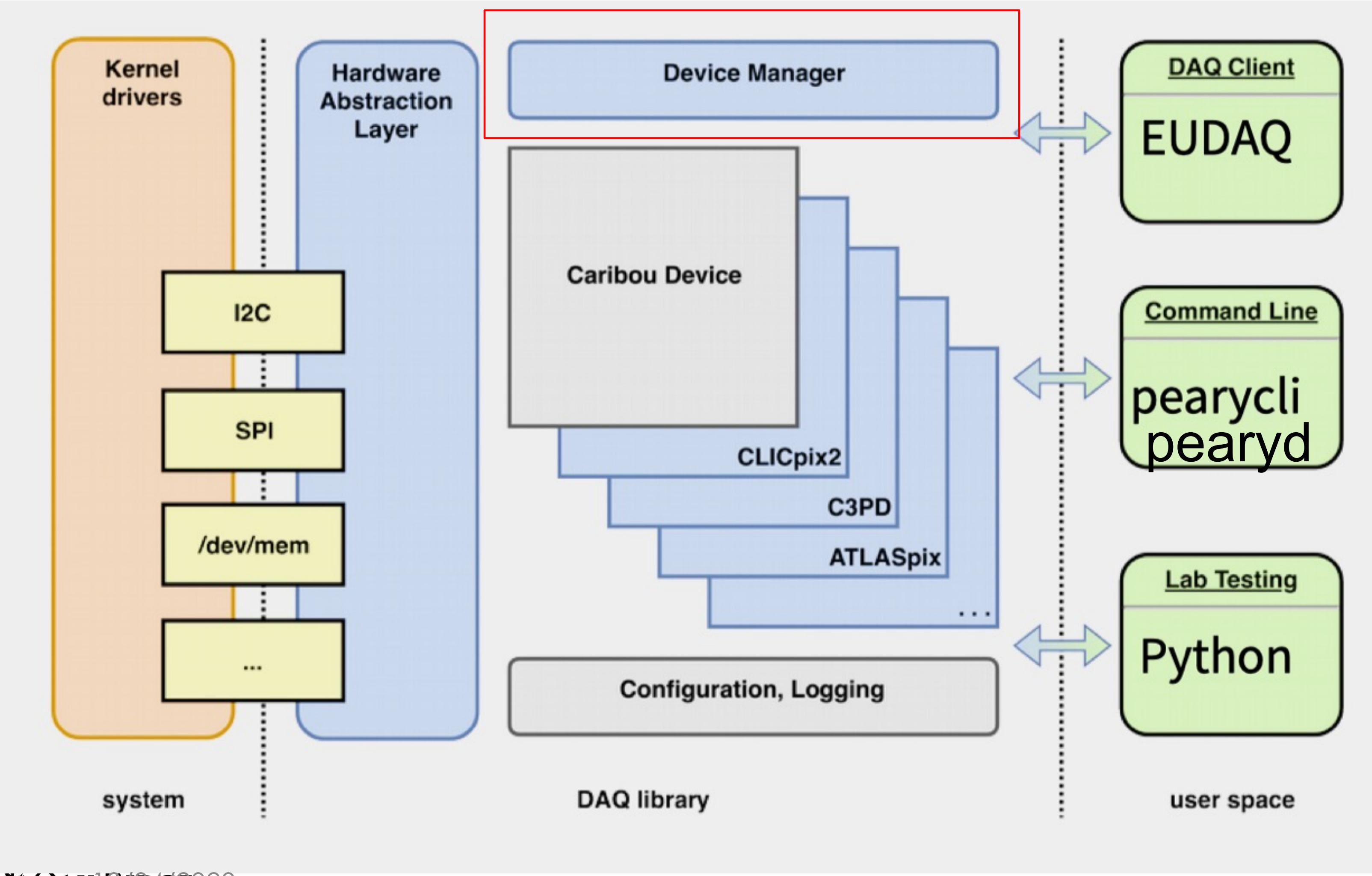
The Peary Framework



Hardware Abstraction Layer

- Implementation specific to daughter board (i.e. CaR Board)
- Library implementing "Human friendly" function to control daughter board
  - SetPowerOn/Off
  - ReadCurrent
  - ProgramClock
  - SendPulse
  - ReadADC

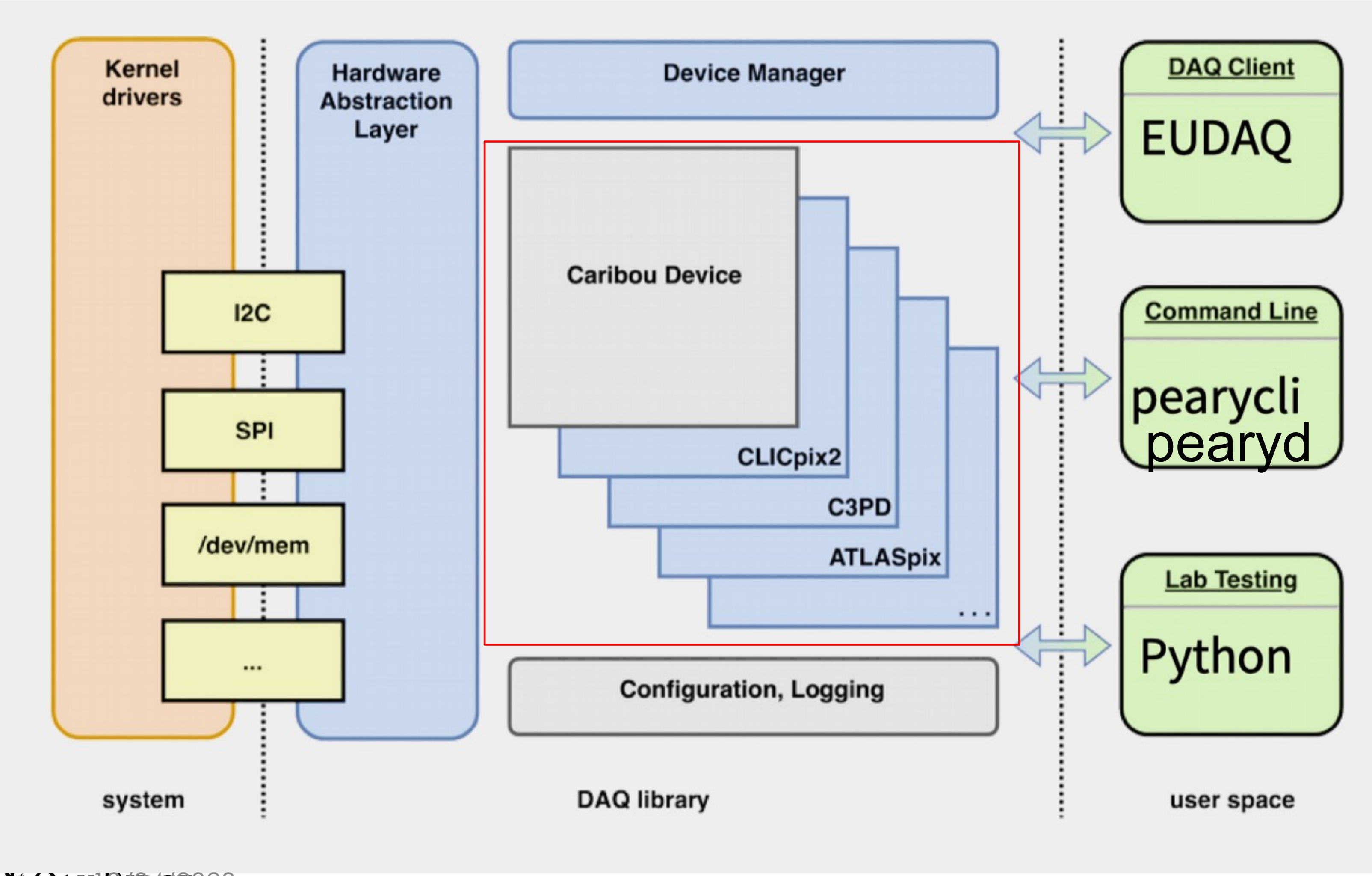
The Peary Framework



Device Manager

- Manage devices that are instantiated (ex: 1x FEI4, 1 Temperature Sensor, etc.)

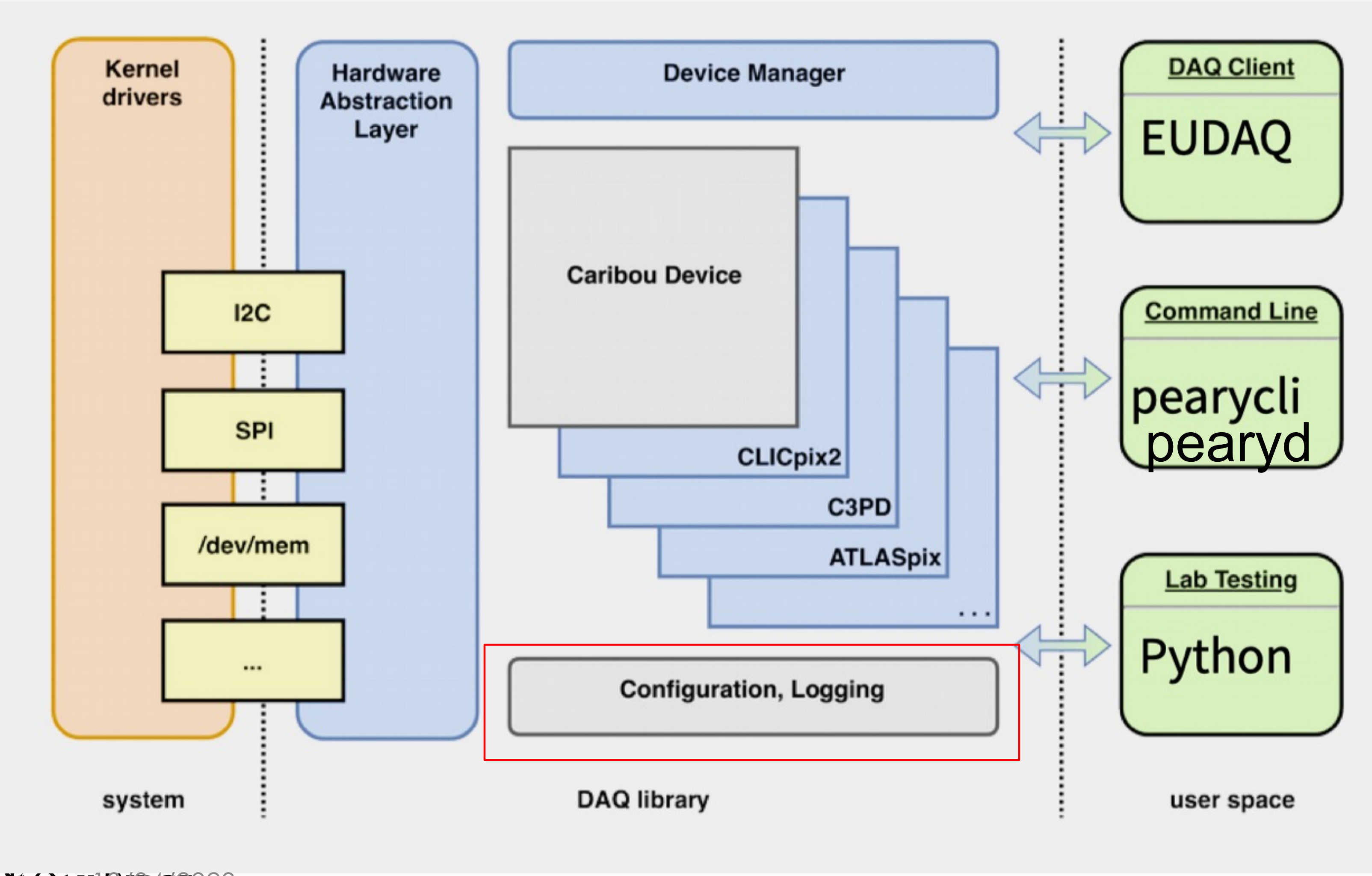
The Peary Framework



Devices

- Code controlling specific devices, implemented by users
- Interface to device specific firmware
  - Registers
  - Power, DACs
  - Readout
- Implement commands accessible via Command Line, server or API
- Contained in 1 C++ class derived from Template

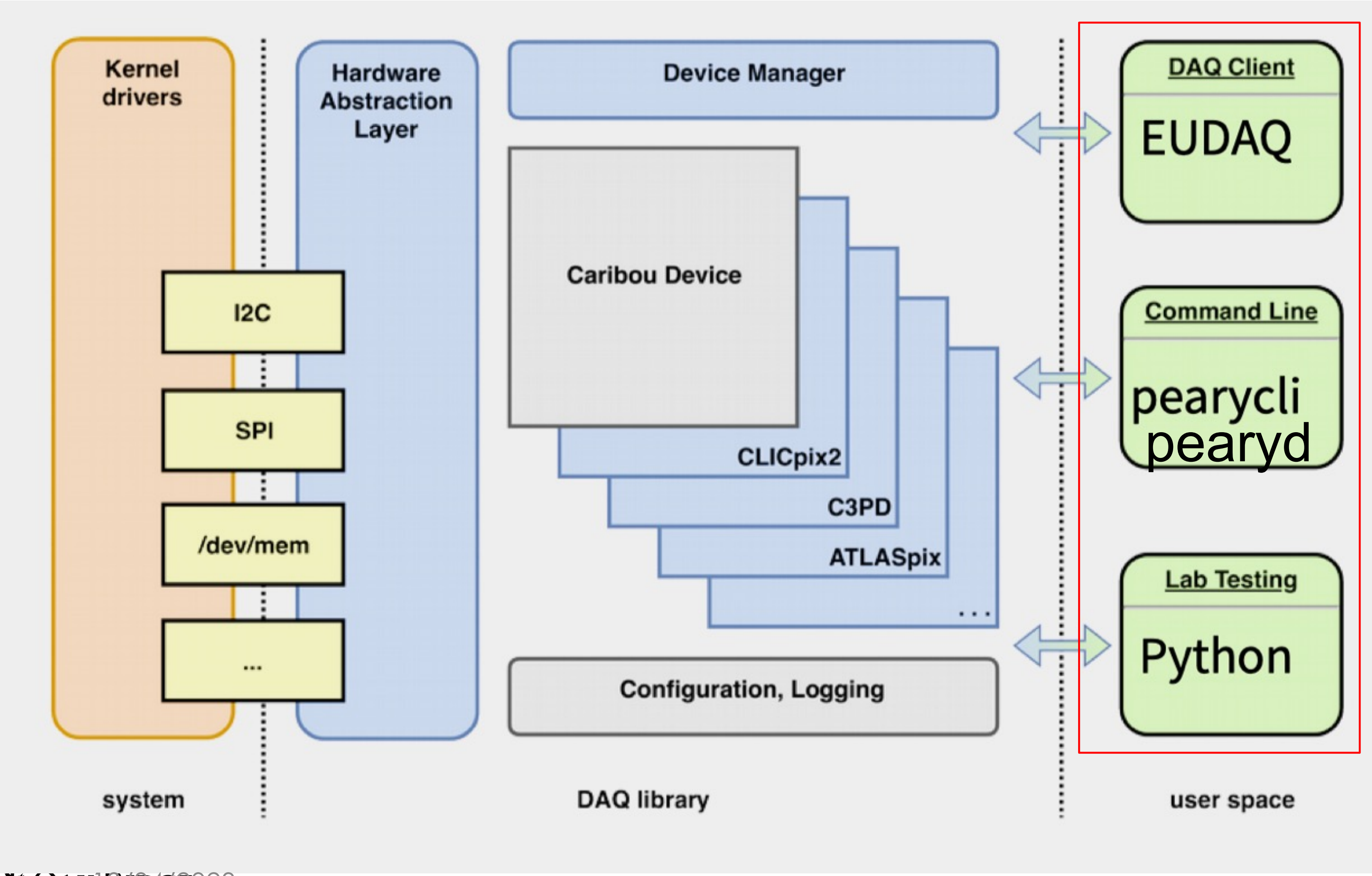
The Peary Framework



Configuration, Logging

- Log of all events in the framework
- Provide generic mechanism to load/save configuration of device

The Peary Framework



User space

- Command line interface to the framework (inside ZC706)
- TCP Client/Server CLI equivalent
- Python binding for simple scripting

Firmware

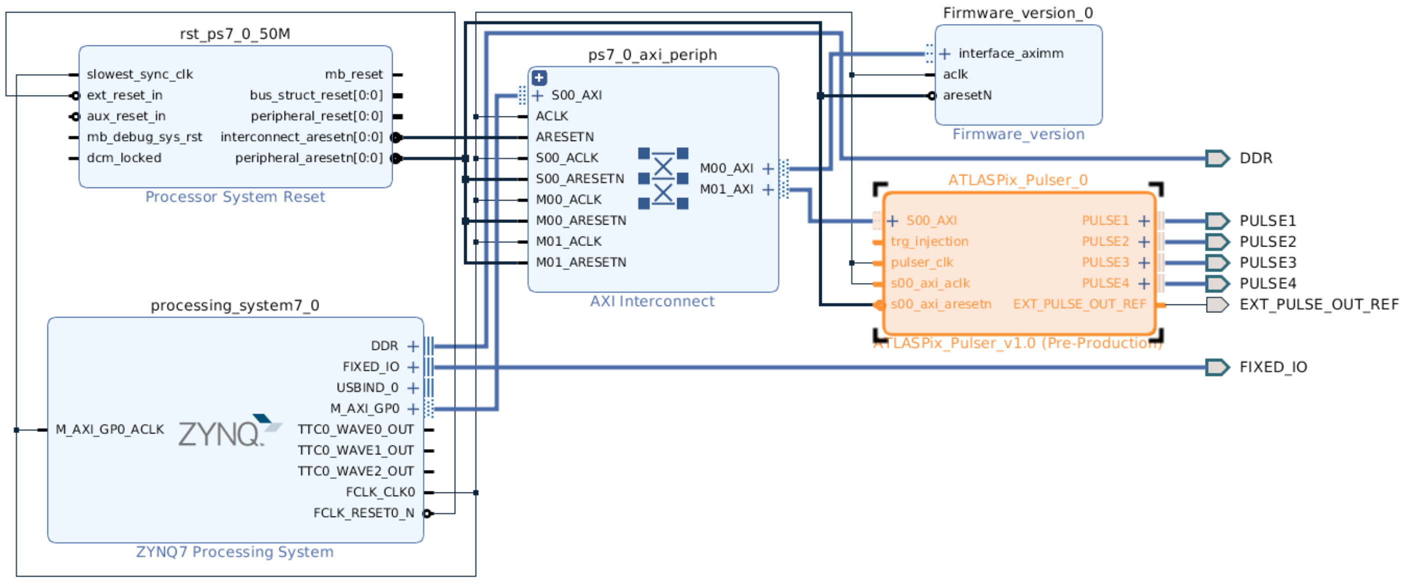
Minimal Core Image : https://gitlab.cern.ch/bnl-caribou/caribou-core

- Processor configured for CaR Board
- Firmware IP to read FW version
- Base script for project generation, CI

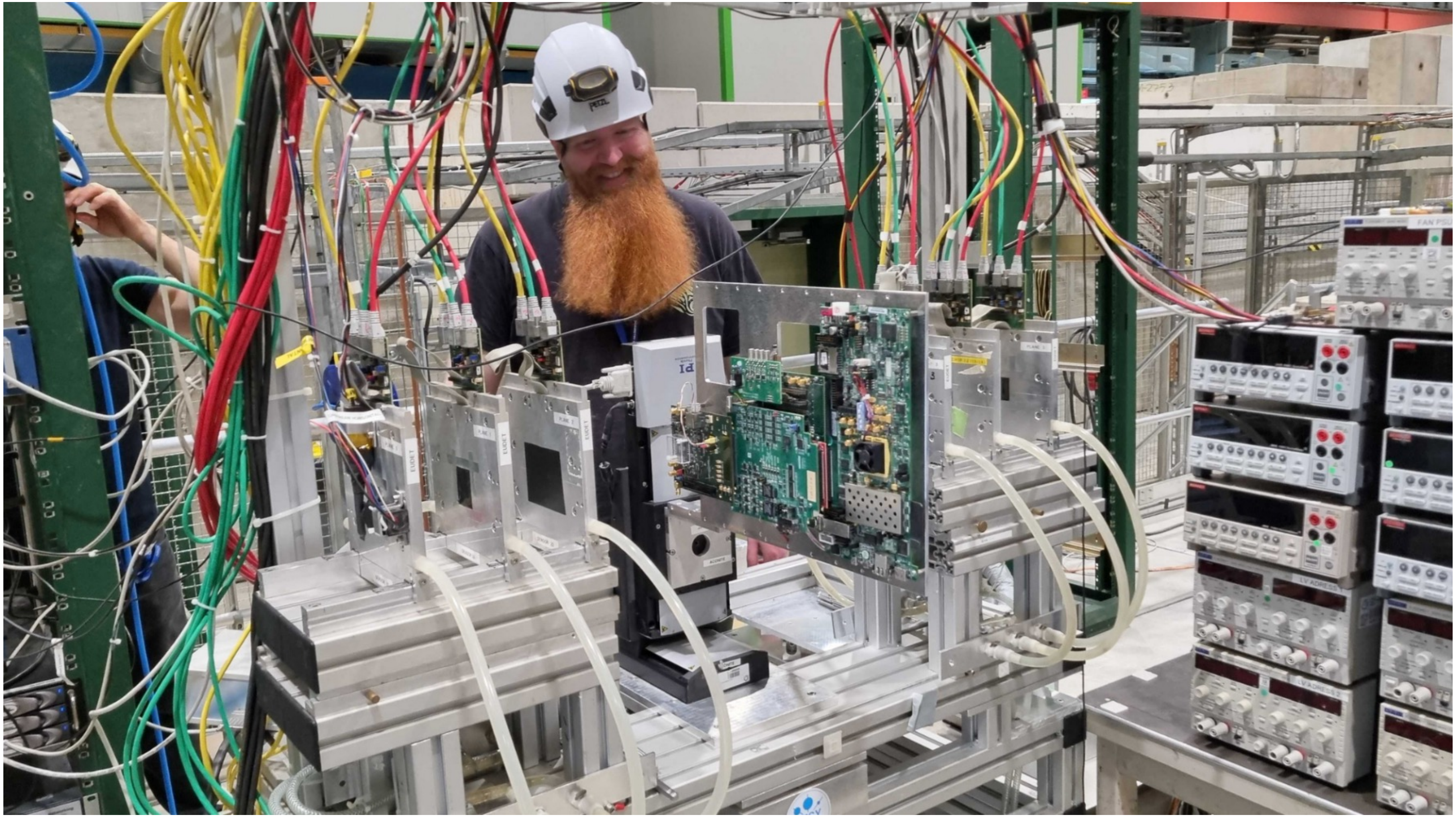
Device specific IP in separate repo, packaged using Vivado IP integrator

- can be included as sub-modules in your FW git project
- Can be integrated via Board design or in wrapper

Ex: https://gitlab.cern.ch/ATLASPix/ATLASPix3FW



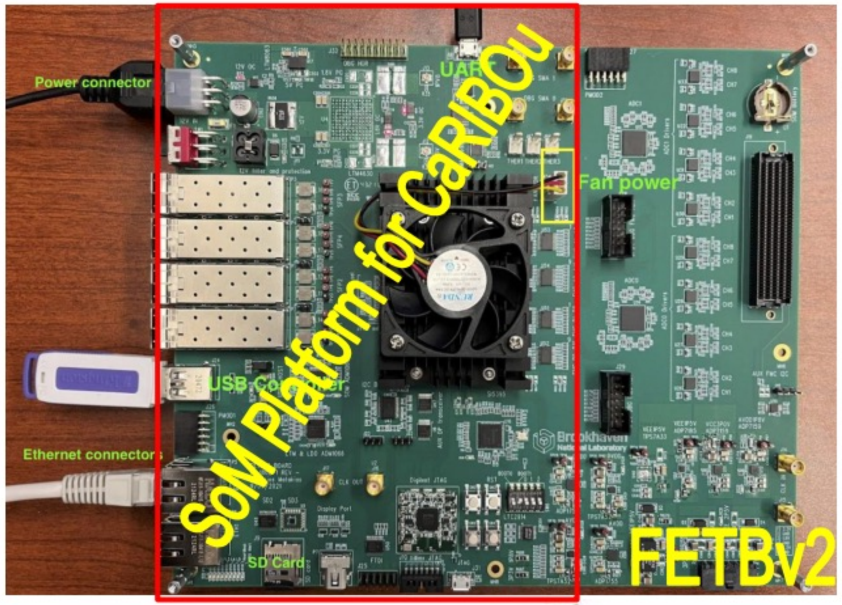
Happy users!



Upgrade of CaRIBOu System

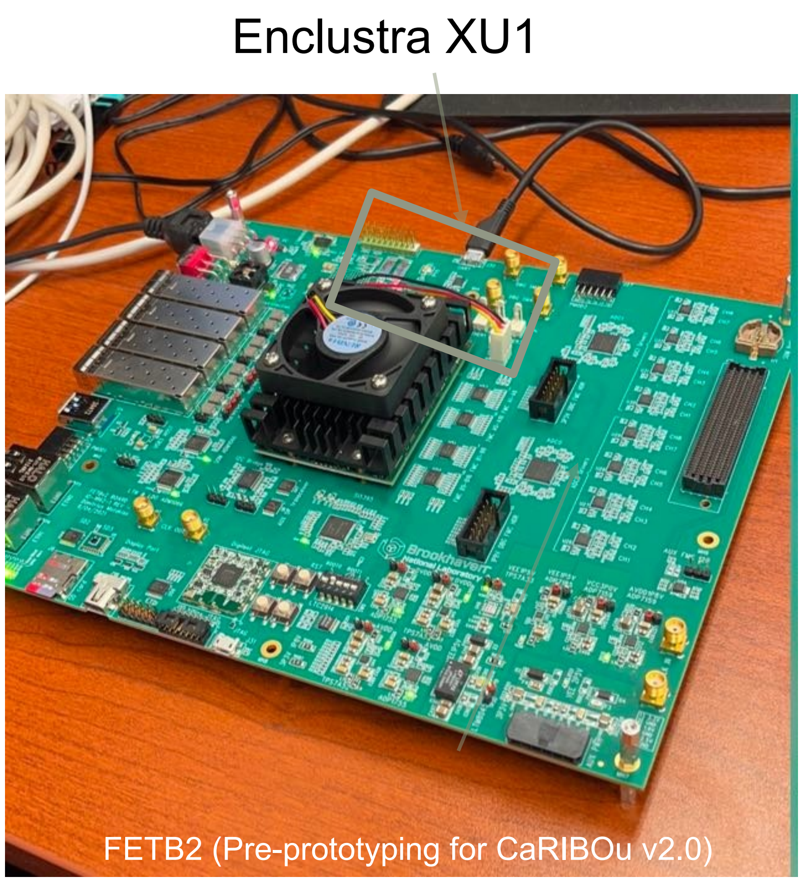
Next development will focus on reducing costs while increasing resources by taking advantage of progress in FPGAs:

- using commercial System-on-Module (SOM) to optimize the system cost for easy deployment in different experiments.
- possible utility of AI/ML
- The user community is being expanded with **collaboration** of:
  - CERN EP R&D program
  - EU H2020 Innovation Pilot program
  - in addition to the collaboration on advanced monolithic silicon sensors studies with RD50, CLIC, ALICE and NASA



CaRIBOu v2.0 plans

- In order to adapt to the new needs of the users, a new platform based on Enclustra XU1 SOM (ZynqMP) is planned :
  - Cost reduction by using minimal SOM with reduced cost with regard to ZC706
  - Increase in PL resources
  - Increased bandwidth for High-Speed IOs , for output to DAQ
  - Add USB host capabilities
  - Additional general purpose IOs
  - Adapt signals, power and voltage bias to new technologies beyond 65nm
  - 64bit processor and additional processing power
  - Add temperature monitoring for test-beam, lab characterization needs
  - Integration with modern Xilinx tools
- Preliminary results with similar platform (FETB2) designed at BNL show promising results to achieve these specifications in CaRIBOu v2.0



FETB2 (Pre-prototyping for CaRIBOu v2.0)

The RD50 Collaboration and ALPIDE 65nm have shown interests in using the future system for their developments

- Part of CERN EP R&D as part of WP 1.4 “Si Simulation and Characterization” with strong link to WP 1.1 and 1.2, hybrid and monolithic pixel detectors
- Strong requirements from pixel R&D for large bandwidth, analog signal processing, timing and more system integration
  - Realization of ADC Add-on,10 Gbps Network FW examples very well within BNL competences

CARIBOu 2.0 Proposal

The system also includes :

- Internal and external Temperature and voltage monitoring
- Si5345 Jitter cleaner for clock generation and processing
- Full support for USB3.0 (to plug HDD, keyboard etc.)
- SD Card reader for OS, data, FW
- Interface to DUT via HPC/LPC connector
- HPC FMC Implementing VITA standard for possible connection of commercial FMC Cards such as ADCs

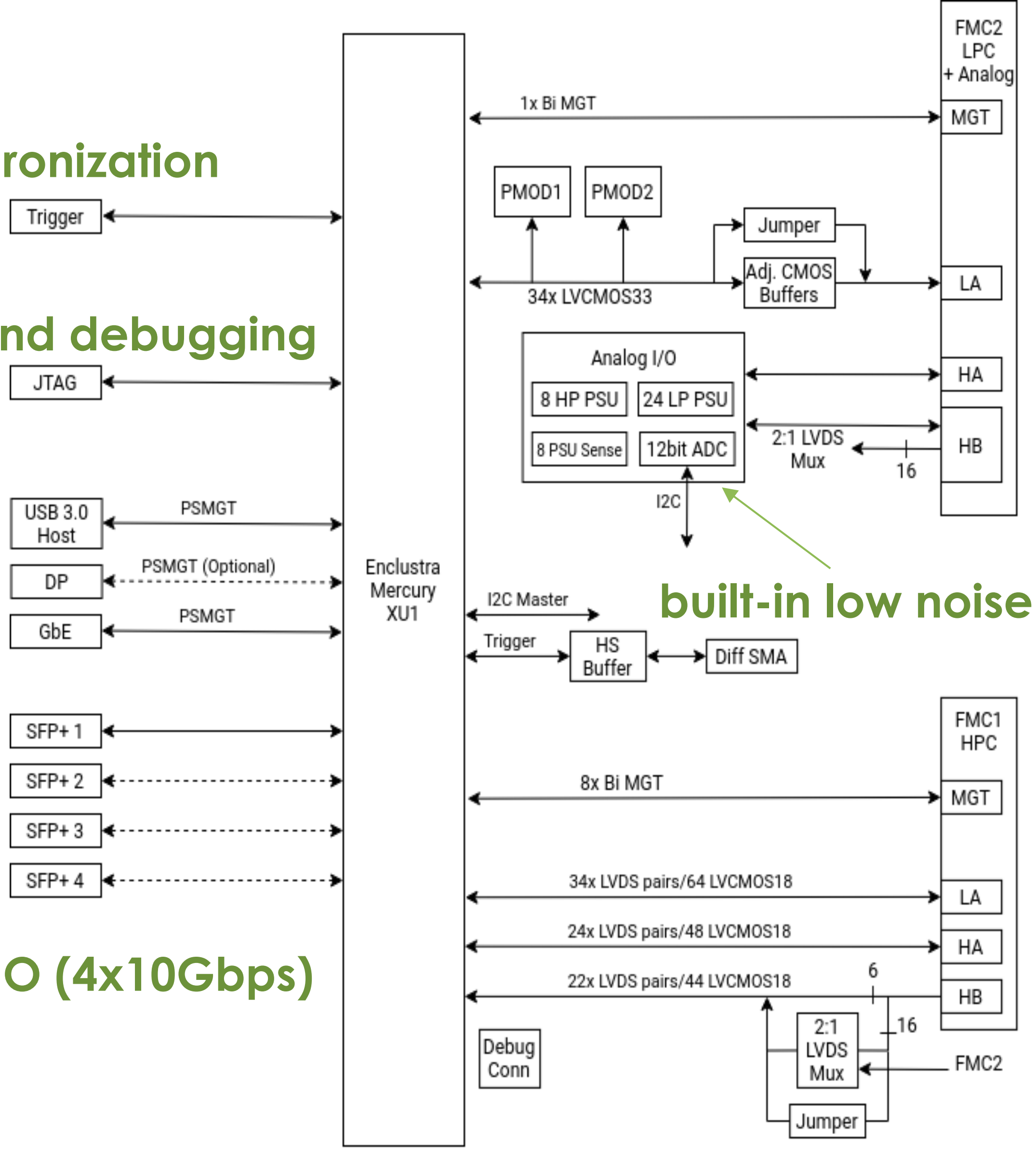
Synchronization

Programming and debugging

Slow Control/IO

Hi-Speed IO (4x10Gbps)

built-in low noise PSU



Note 1: SFP 2,3,4 circuitry to be left in schematics but omitted from layout
Note 2: n Tx LVDS = n Rx LVDS, n Tx CMOS = n Rx CMOS
Note 3: All 40 pin analog connectors have an identical pinout

CARIBOu 2.0 Power/ Supplies

8 x High power

- 2 quadrant
- -10V to +10V
- -1A to +1A
- 4mV step size
- Manually controlled (by hand) polarity

8 x Low power

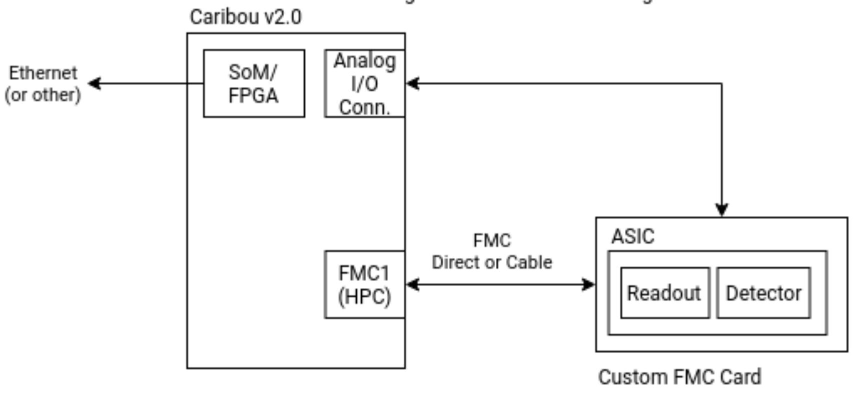
- 2 quadrant
- -10V to +10V
- -200mA to +200mA
- 10uV step size
- 250nA step size
- Values are targeted and TBC, low power still being designed

8 x Bias voltages/currents:

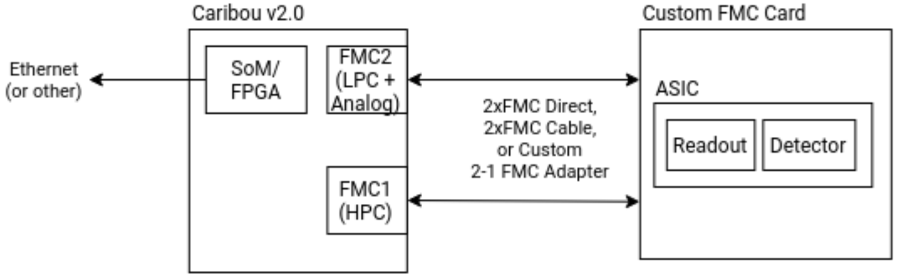
- 0V to +5V
- -5mA to +5mA
- 2mV step size
- DAC outputs, TBC, may be converted to low power supplies, only power limited

CARIBOu 2.0 usage methods

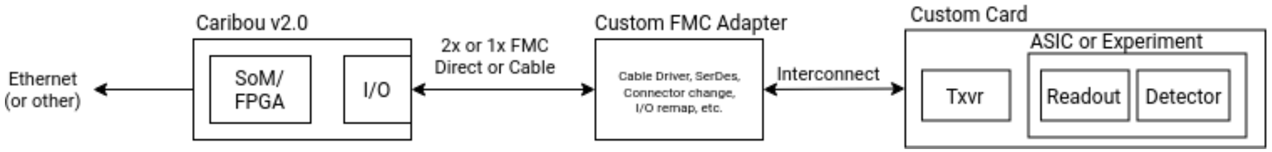
High Datarate External Analog



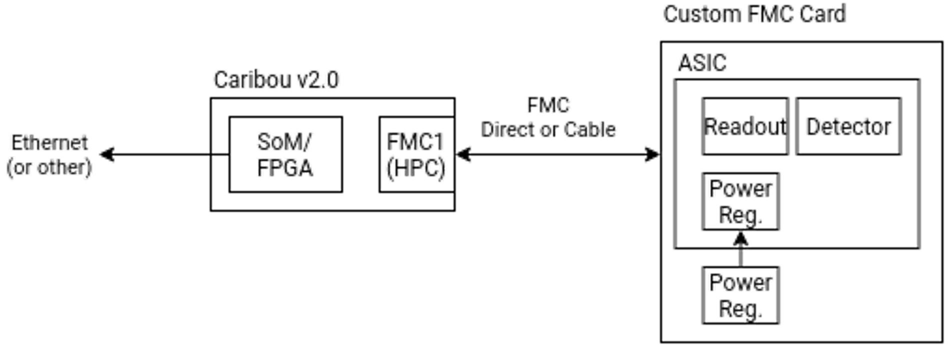
Full I/O Merge



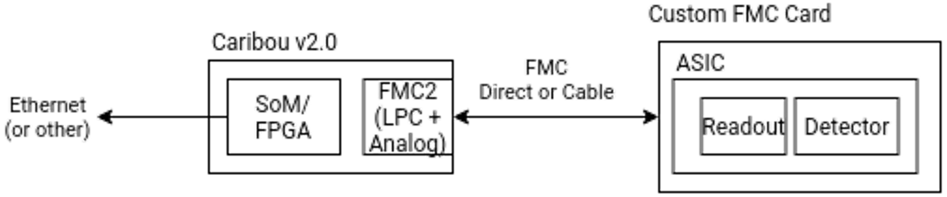
Remote/Testbeam Usage



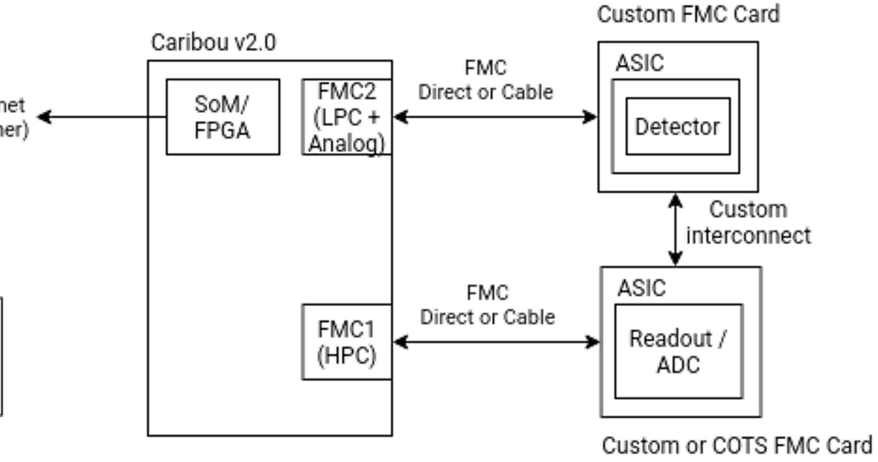
High Datarate Direct Usage



Low Datarate Direct Usage



External Readout



CARIBOu 2.0 Software

We build the OS for FETBv2/CARIBOu 2.0 with Xilinx petalinux-tools

- Layer on top of Yocto to simplify some of the process
- Take care of tools installation, go support, deploy scripts for startup etc..
- https://github.com/mathieubenoit/FETB2_15EG_OS

Slow Control/API for CARIBOu 2.0 will be implemented by CERN/DESY/ORNL in Python/C++ and run inside the PL

- Client/Server configuration possible, but system can be standalone
- For more computing intensive activities (Readout, processing, ML/AI etc., IO to ROOT files etc), other language like Go, C++ can provide executable that run natively in ARM processor

Conclusion and perspective

The CARIBOu 2.0 platform is currently under design, based on our experience with the FETB2, that already integrates an Enclustra XU1 FPGA

- The goal of the system is to “avoid re-inventing the wheel”
- The system is supported by a community of users, which facilitate debugging, code and experience sharing
- Important software and firmware developed in common between users, avoiding work duplication and favoring more robust code
- The system is versatile, from a simple standalone tabletop measurements to large bandwidth system interconnecting with a DAQ like FELIX, it allows to naturally progress towards a final design
- With proper design of FW/SW/HW, this system and the work done can be carried up to the final system/RDO

Backup

The FETBv2 Firmware

- JESD Interface for ADC with readout at full throughput
  - Synchronization between ADCs
  - Common clocking via Si5345 Chip

- ADC decoding and data sorting

- ADC Data stream to AXI-Stream
  - Achieve bandwidth of ~1.4GB/s to RAM -> 2xADC @ 40MHz

- Pulser IP with AXI-programmable MMCM for phase scan with 0.01° resolution

SW in Go

Goals of Go

- Provide a simple language in terms of syntax
  - Syntax similar to python, C, Java
- Provide an easy method to deal with modern multi-processor architecture to maximize its usage
  - The language provides mechanism to simply handle concurrency and parallelism issues
- Provide the speed of a compiled language, and the feeling and ease of use (libraries etc.) of a compiled language
  - Large set of standard libraries covering the needs of a modern software
- Provide ease of deployment **via the use of static linking**

Go language in a nutshell

- The go language is an object-oriented language, but not in the common sense
  - No classes, inheritance, polymorphism
  - **Types can be attributed methods**
  - Object-Oriented concept via **Interfaces**
  - Functions are objects
- The language is strongly typed
  - Conversion between types always explicit
- The language is not sensitive to indentation

The go environment

The go compiler can be downloaded and installed for any platform and architecture from www.golang.org

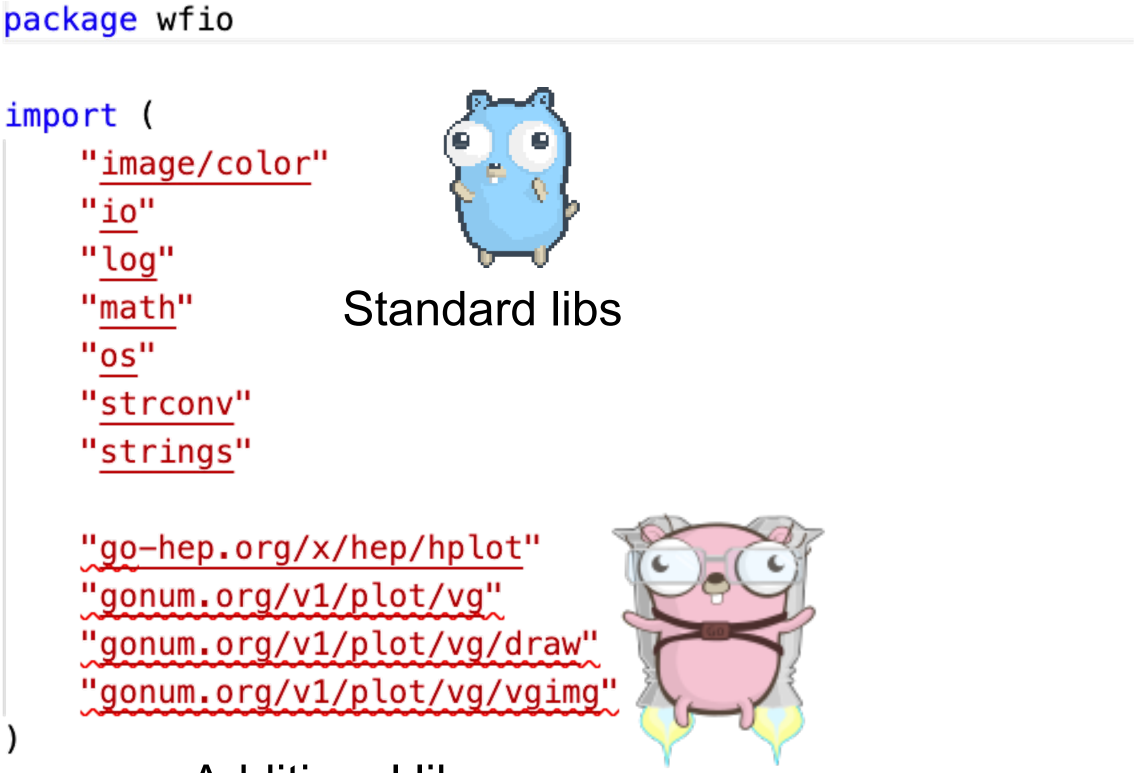
- The compiler comes with all the tools required to develop in the go langage : Compiler, formatter, linter, code checker, profiler , tracer, tester, coverage and version control integration
  - It is recommended to use an IDE such as Visual code to make most use of the tools
- The go language is statically linked, so the executable produced **are standalone and contain all the code needed to run**
- The compiler can **produce executable for any platform** that run the go compiler (for example Linux ARM64)

Go libraries

The go standard library provides a wide array of tools to produce quickly modern code, not usually available in standard lib in other languages :

- Stats, math functions, logging and error handling
- http, tcp server client handling
- Cipher, cryptography etc …
- File readers and writer (JSON,YAML,HDF etc…)
- Image handling etc..

Libraries can be used simply by adding their URL to the code, the go compiler will fetch and compile them for you



```go
package wfio

import (
	"image/color"
	"io"
	"log"
	"math"
	"os"
	"strconv"
	"strings"

	"go-hep.org/x/hep/hplot"
	"gonum.org/v1/plot/vg"
	"gonum.org/v1/plot/vg/draw"
	"gonum.org/v1/plot/vg/vgimg"
)
```

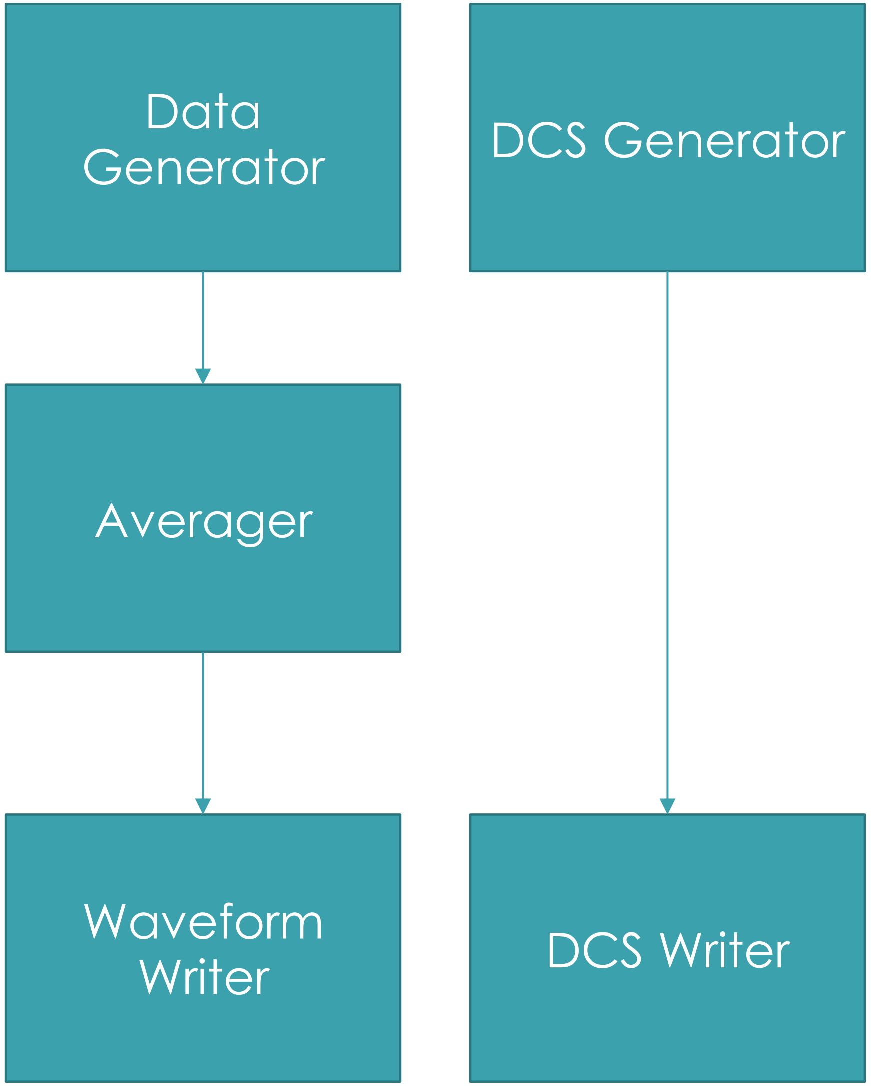
Standard libs

Additional libs

Example : DAQ

In this example, we show how to use buffered channel to maximize the throughput of the system by using all threads to average waveform and write them to disk (in ROOT format!)

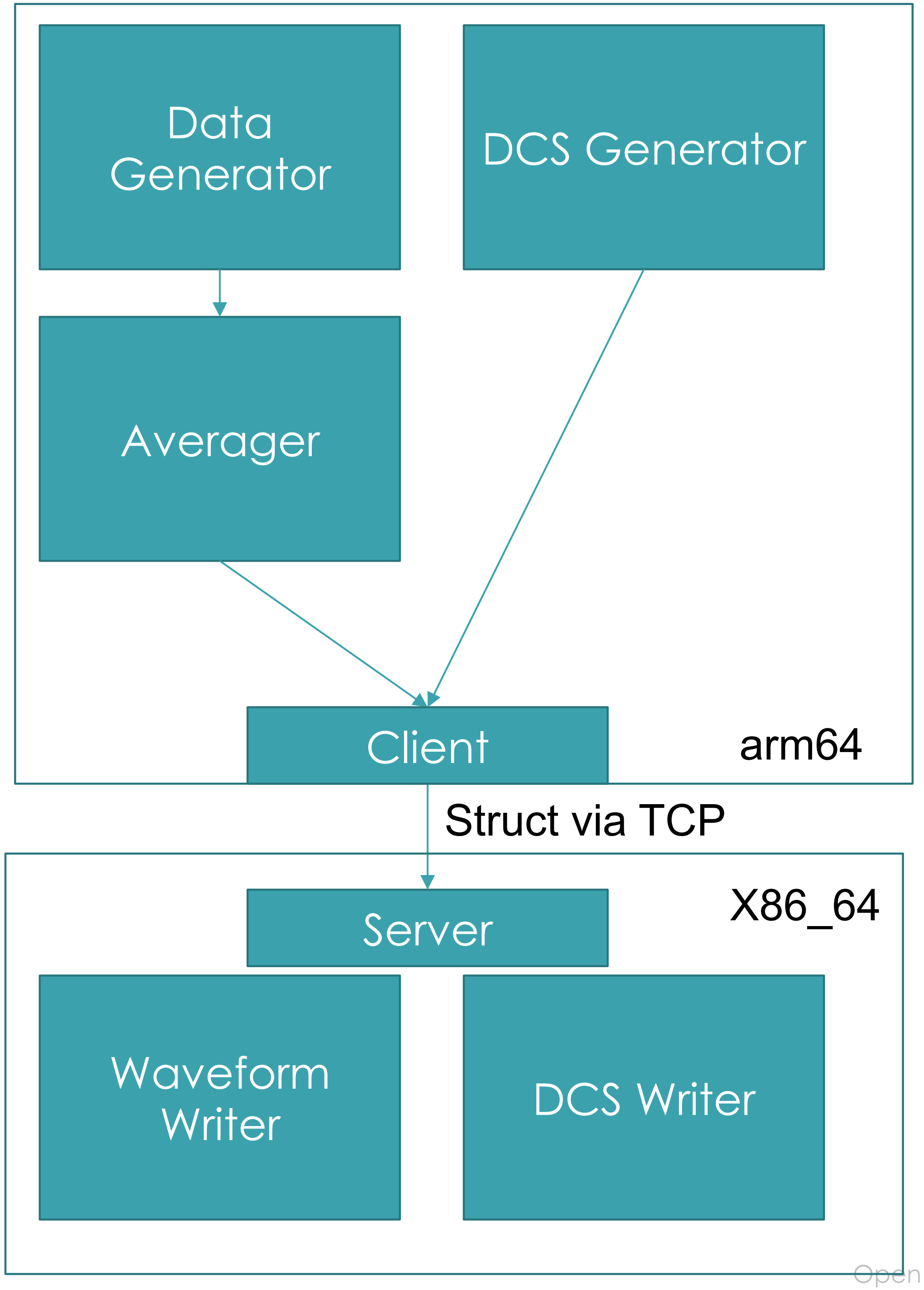
In a second step, we show how a few line of code can change this application into a heterogenous client server application



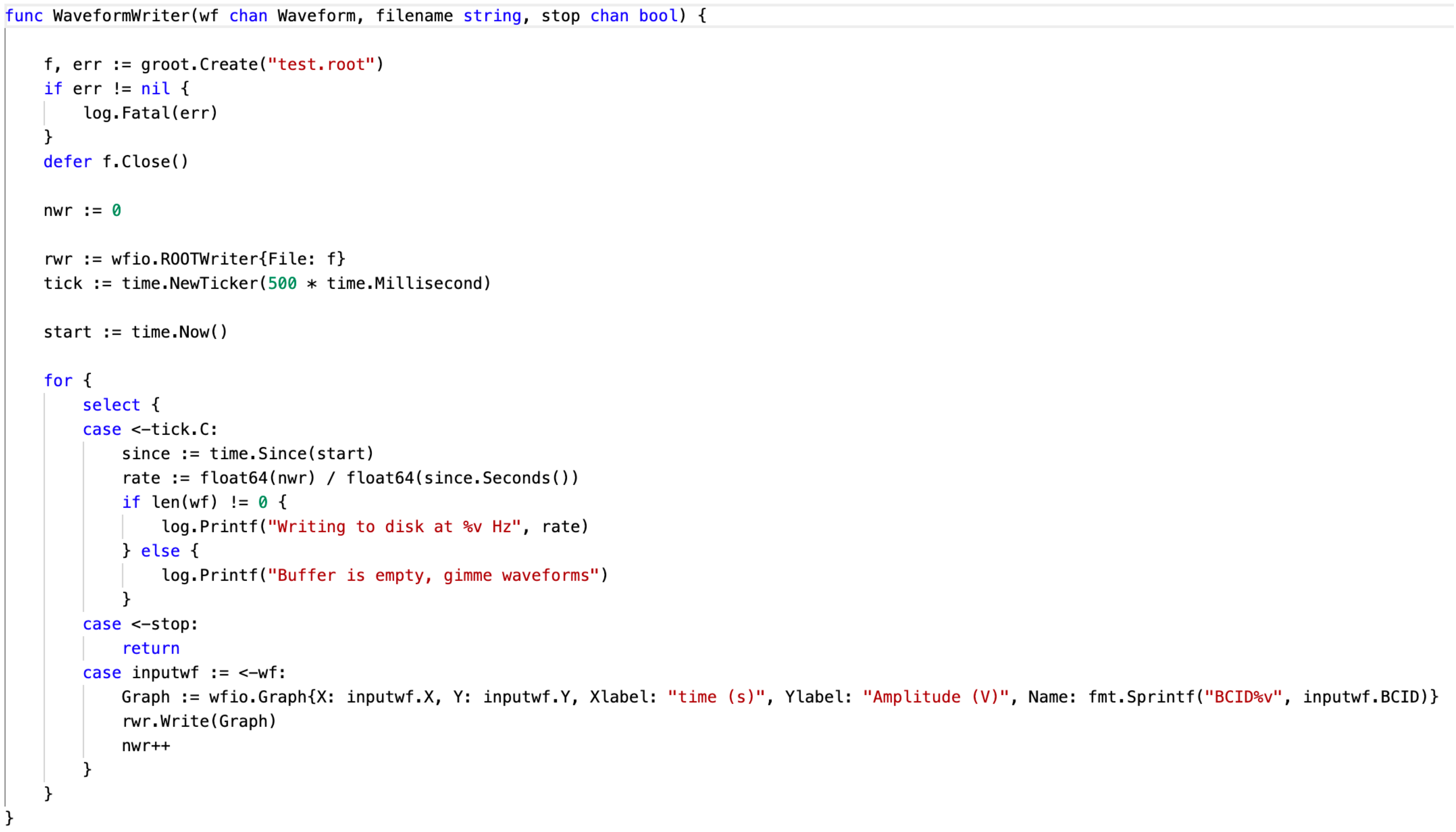
Example : DAQ

In this example, we show how to use buffered channel to maximize the throughput of the system by using all threads to average waveform and write them to disk (in ROOT format!)

In a second step, we show how a few line of code can change this application into a heterogenous client server application, using cross-compilation features of the go compiler



Writing to ROOT file



```go
func WaveformWriter(wf chan Waveform, filename string, stop chan bool) {

    f, err := groot.Create("test.root")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    nwr := 0

    rwr := wfio.ROOTWriter{File: f}
    tick := time.NewTicker(500 * time.Millisecond)

    start := time.Now()

    for {
        select {
        case <-tick.C:
            since := time.Since(start)
            rate := float64(nwr) / float64(since.Seconds())
            if len(wf) != 0 {
                log.Printf("Writing to disk at %v Hz", rate)
            } else {
                log.Printf("Buffer is empty, gimme waveforms")
            }
        case <-stop:
            return
        case inputwf := <-wf:
            Graph := wfio.Graph{X: inputwf.X, Y: inputwf.Y, Xlabel: "time (s)", Ylabel: "Amplitude (V)", Name: fmt.Sprintf("BCID%v", inputwf.BCID)}
            rwr.Write(Graph)
            nwr++
        }
    }
}
```

FETBv2 DCS Monitoring

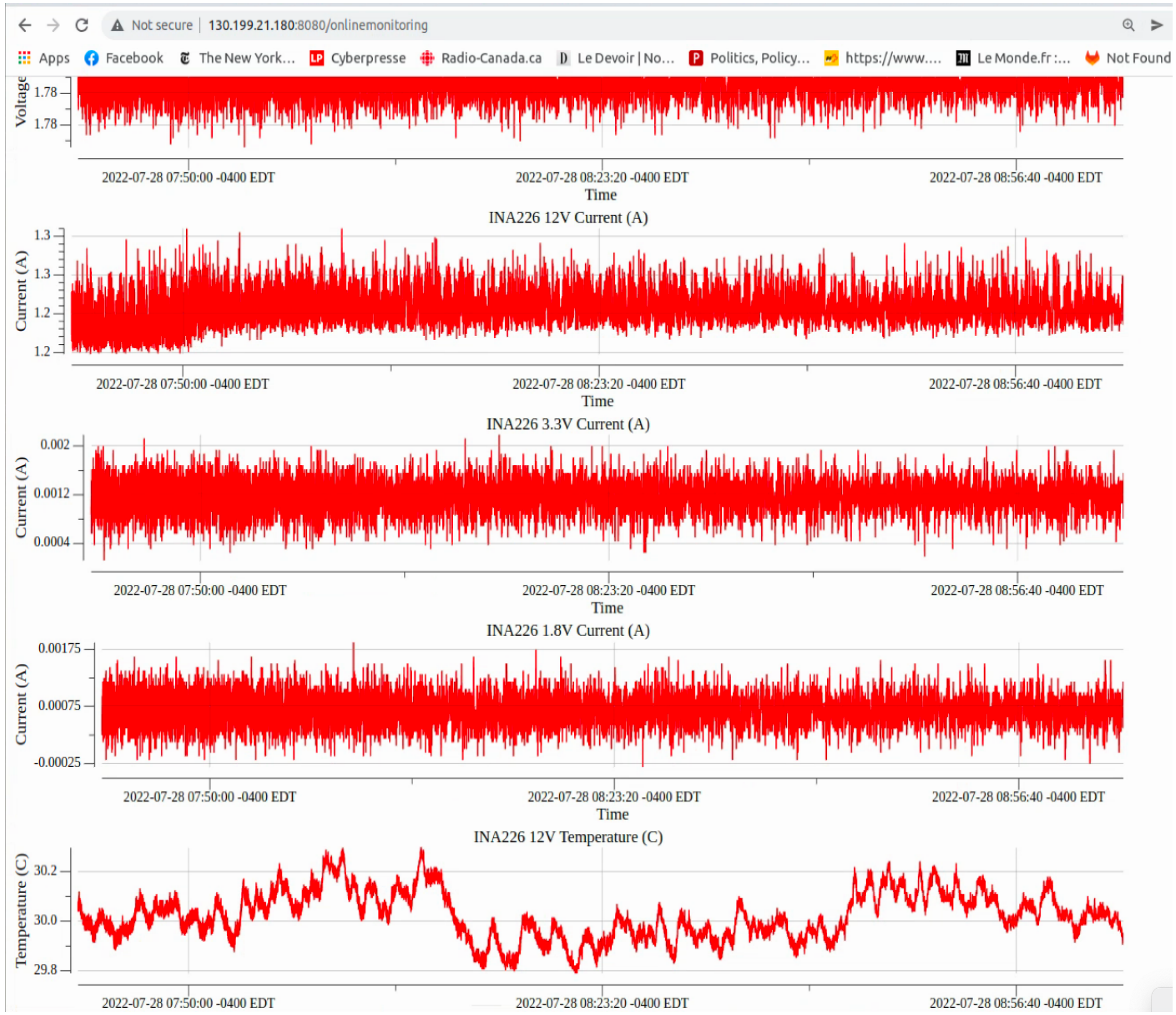
DCS Consist of measurements from two sources :

- Internal to FETB2 (read via I2C)
  - INA228, MAX Temperature chip (including 3 external NTC for ALFE2, etc.)
  - Power and Voltages from INA228
- External (Read via SCPI over Ethernet)
  - Keysight power supplies

The DCS runs in the FETB2 and act as an HTTP server that display the live results of measurements

- A log of all event is stored in a ROOT Tree/File

DCS


Not secure | 130.199.21.180:8080/onlinemonitoring
Apps
Facebook
The New York...
Cyberpresse
Radio-Canada.ca
Le Devoir | No...
Politics, Policy...
https://www....
Le Monde.fr :...
Not Found
Voltage
1.78
1.78
2022-07-28 07:50:00 -0400 EDT
2022-07-28 08:23:20 -0400 EDT
2022-07-28 08:56:40 -0400 EDT
Time
INA226 12V Current (A)
Current (A)
1.3
1.3
1.2
1.2
2022-07-28 07:50:00 -0400 EDT
2022-07-28 08:23:20 -0400 EDT
2022-07-28 08:56:40 -0400 EDT
Time
INA226 3.3V Current (A)
Current (A)
0.002
0.0012
0.0004
2022-07-28 07:50:00 -0400 EDT
2022-07-28 08:23:20 -0400 EDT
2022-07-28 08:56:40 -0400 EDT
Time
INA226 1.8V Current (A)
Current (A)
0.00175
0.00075
-0.00025
2022-07-28 07:50:00 -0400 EDT
2022-07-28 08:23:20 -0400 EDT
2022-07-28 08:56:40 -0400 EDT
Time
INA226 12V Temperature (C)
Temperature (C)
30.2
30.0
29.8
2022-07-28 07:50:00 -0400 EDT
2022-07-28 08:23:20 -0400 EDT
2022-07-28 08:56:40 -0400 EDT

FETB2 Software

Functionalities of the SW implemented via tools

- Noise Scan
- Linearity Scan
- Peaking Time Scan
- Baseline Scan
- etc.

The underlying libraries allow the scripting of the main board functionalities

- Pulser controller
- DAQ Controller (steer data acquisition)
- MMCM Controller
- Slow Control Register controller

The various controllers are integrated into a top controller exposing the required function to manipulate the FETB2 State : FETB2 Controller

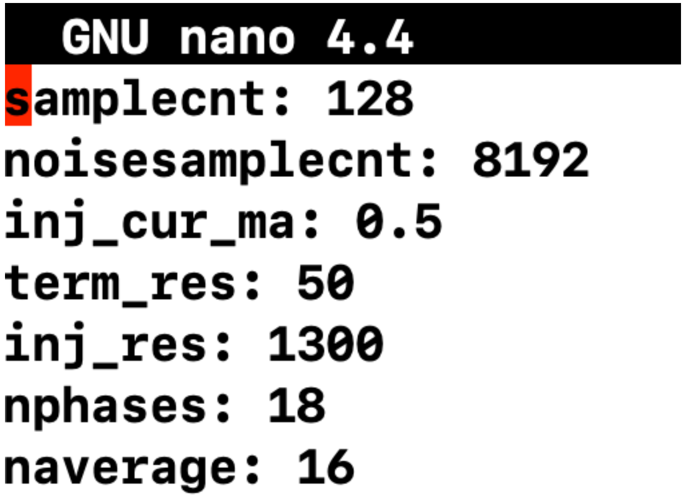
Scans

Each type of scan are based on a common template in 4 steps :

- Initialization of the FETB2 controller
- Scanning step : Acquisition of the required data
- Analysis step : Perform analysis of acquired data
- Writing Results : Write results and raw data to a ROOT file

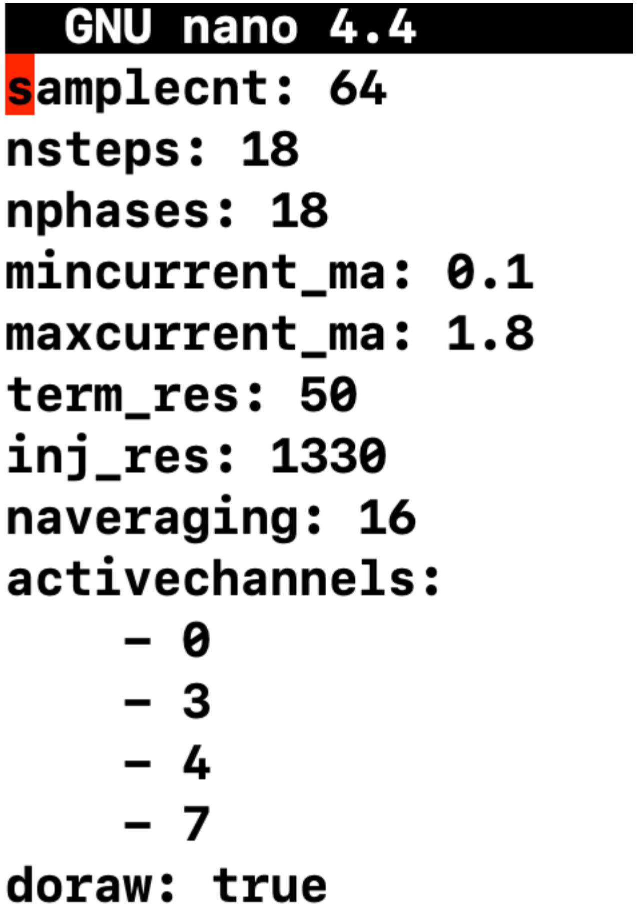
The parameters of each scans are described in a YAML file that can easily be edited to change scan parameters

Noise Scan



```
  GNU nano 4.4
samplecnt: 128
noisesamplecnt: 8192
inj_cur_ma: 0.5
term_res: 50
inj_res: 1300
nphases: 18
naverage: 16
```

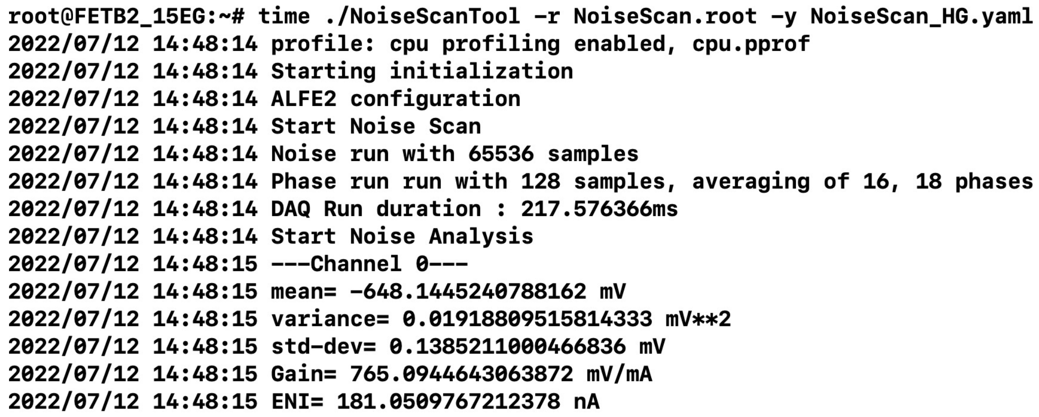
Linearity Scan



```
  GNU nano 4.4
samplecnt: 64
nsteps: 18
nphases: 18
mincurrent_ma: 0.1
maxcurrent_ma: 1.8
term_res: 50
inj_res: 1330
naveraging: 16
activechannels:
    - 0
    - 3
    - 4
    - 7
doraw: true
```

Noise Scan

- Perform a baseline measurement with a large amount of sample (ie: 65536)
- Perform a pulse injection and measure pulse using a delay scan
- Extract gain from delay scan, noise and baseline from noise waveform
- Compute ENI, FFT
- Write raw data, FFT(optional),Analysis results to ROOT file, CSV

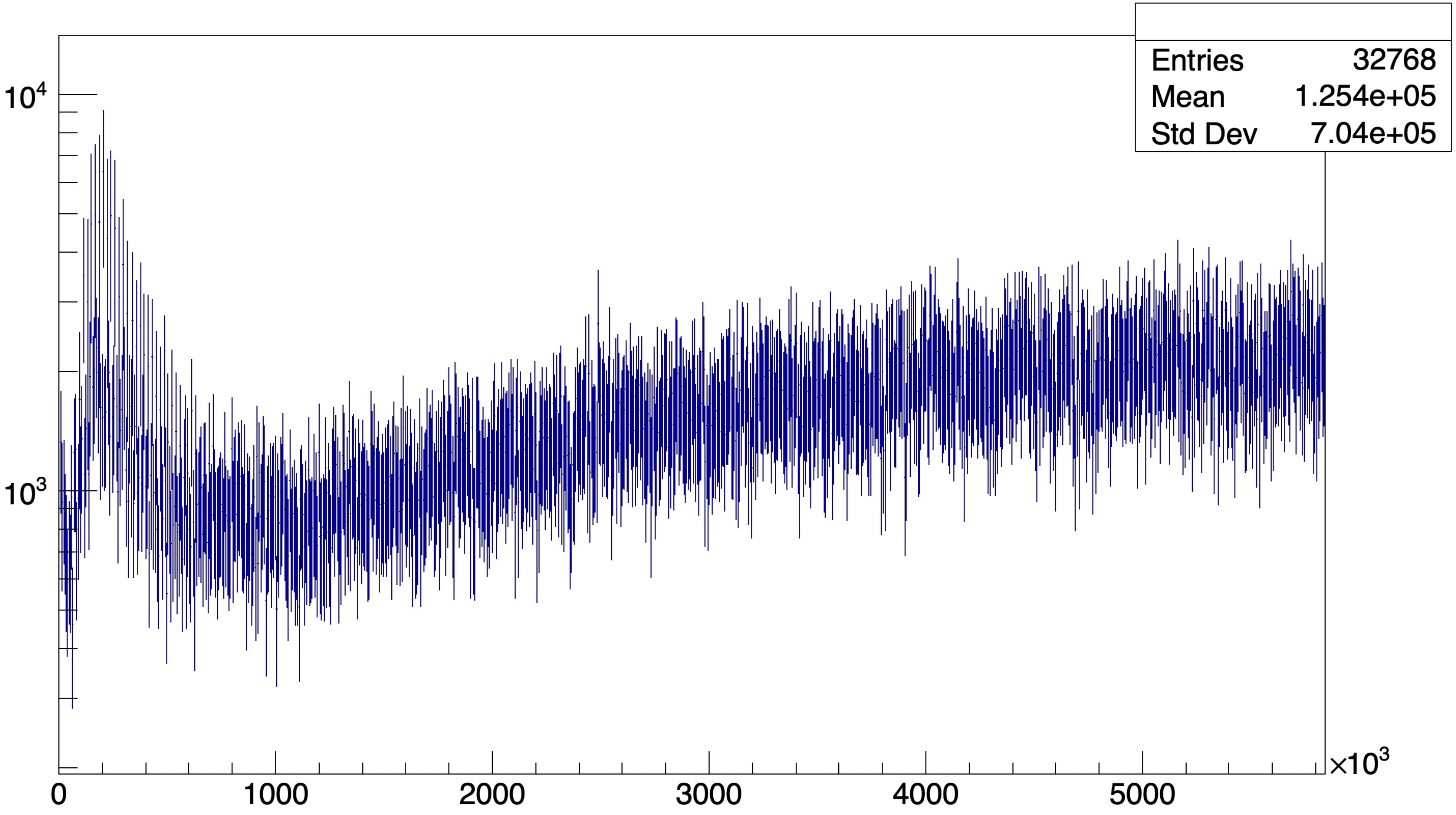


```
root@FETB2_15EG:~# time ./NoiseScanTool -r NoiseScan.root -y NoiseScan_HG.yaml
2022/07/12 14:48:14 profile: cpu profiling enabled, cpu.pprof
2022/07/12 14:48:14 Starting initialization
2022/07/12 14:48:14 ALFE2 configuration
2022/07/12 14:48:14 Start Noise Scan
2022/07/12 14:48:14 Noise run with 65536 samples
2022/07/12 14:48:14 Phase run run with 128 samples, averaging of 16, 18 phases
2022/07/12 14:48:14 DAQ Run duration : 217.576366ms
2022/07/12 14:48:14 Start Noise Analysis
2022/07/12 14:48:15 ---Channel 0---
2022/07/12 14:48:15 mean= -648.1445240788162 mV
2022/07/12 14:48:15 variance= 0.01918809515814333 mV**2
2022/07/12 14:48:15 std-dev= 0.1385211000466836 mV
2022/07/12 14:48:15 Gain= 765.0944643063872 mV/mA
2022/07/12 14:48:15 ENI= 181.0509767212378 nA
```

Execution time (18 phases, 16 averaging, 65536 samples) 1.7s

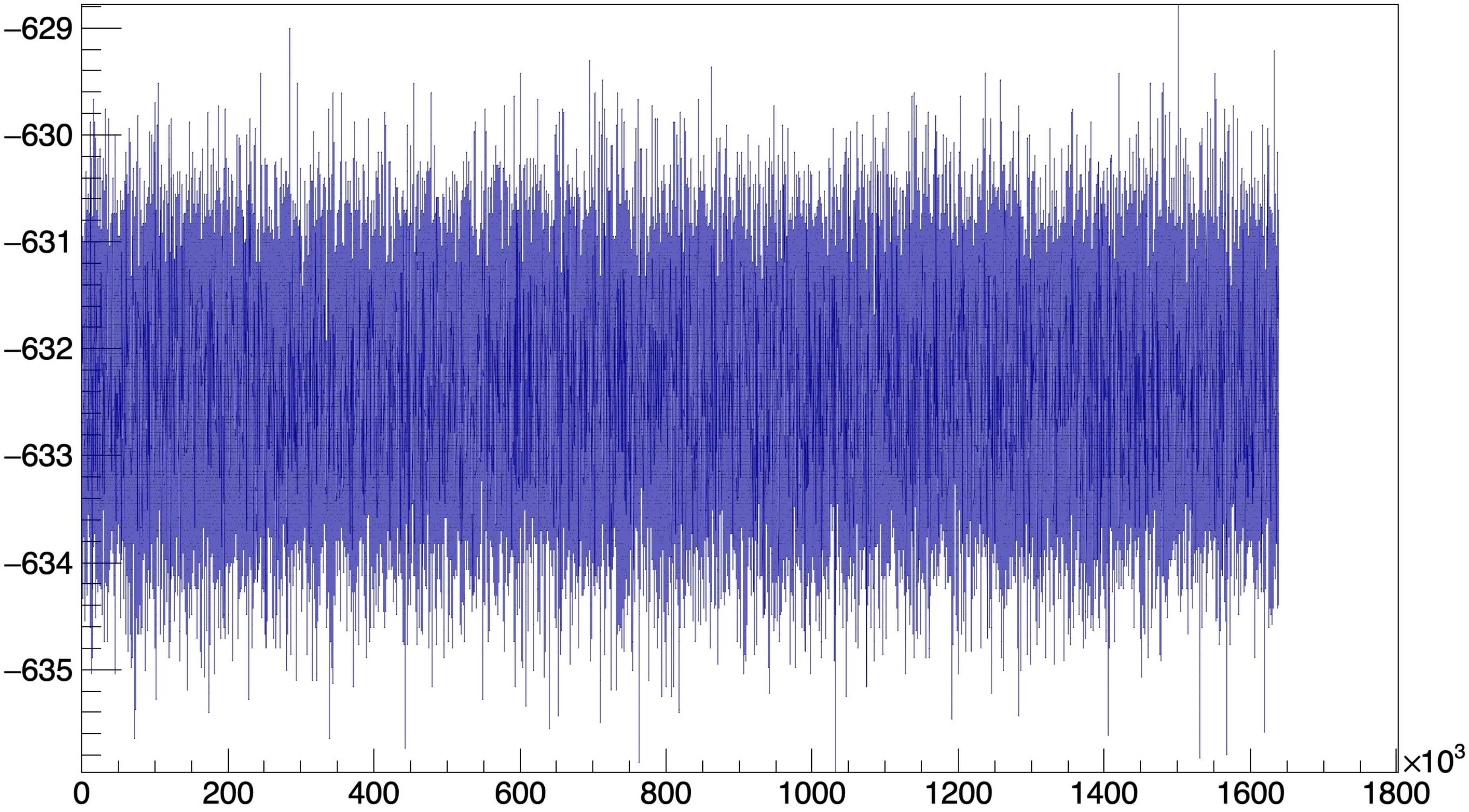
- 500ms for analysis
- 970ms to write data to disk
- 800ms for optional FFT

FFT

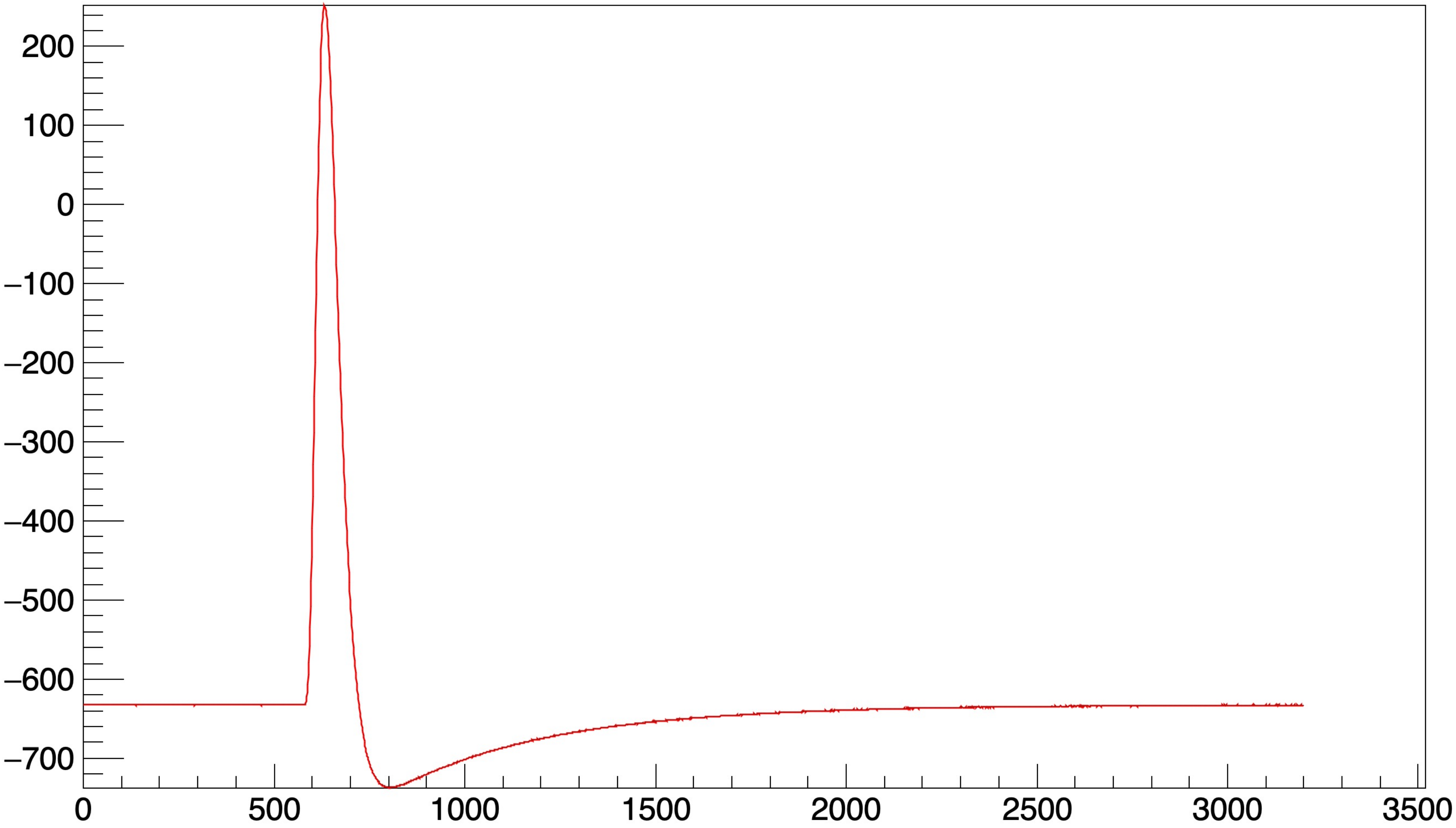


OAK RIDGE National Laboratory

Noise


−629
−630
−631
−632
−633
−634
−635
0
200
400
600
800
1000
1200
1400
1600
1800
×10^3

Pulse

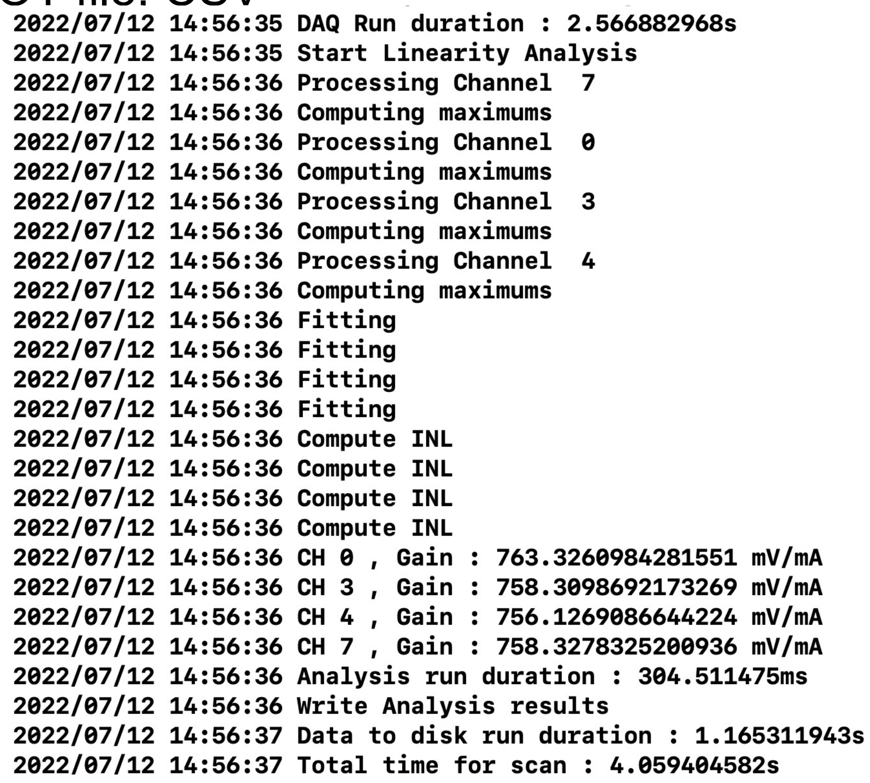

200
100
0
−100
−200
−300
−400
−500
−600
−700
0
500
1000
1500
2000
2500
3000
3500

Linearity Scan

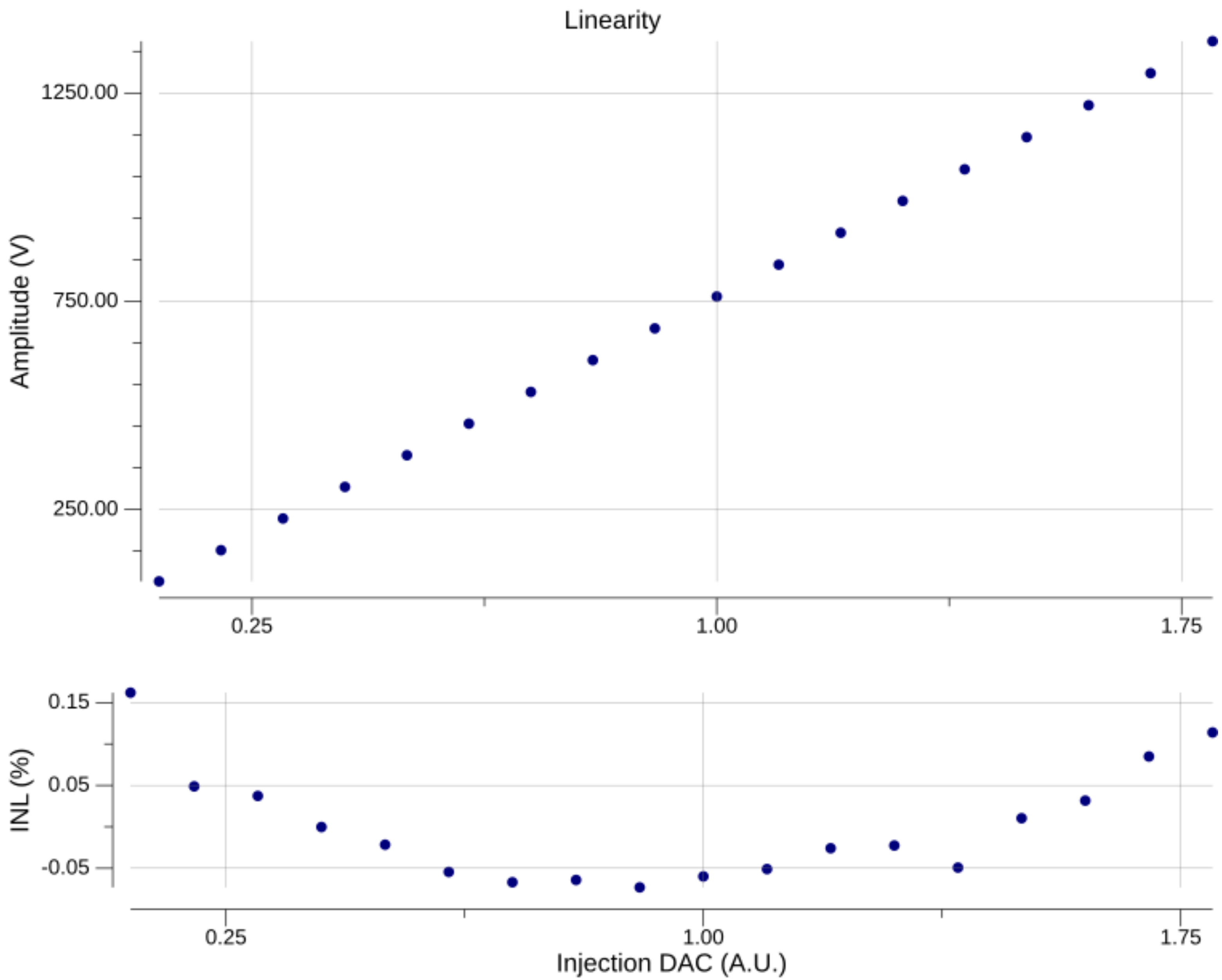
- Perform a pulse injection and measure pulse using a delay scan for each injection in the range
- Extract pulse baseline and amplitude, do linearity fit and extract INL
- Write raw data.Analysis results to ROOT file. CSV

Execution time (18 phases, 16 averaging, 18 steps) 4s

- 2.5s data acquisition
- 300ms analysis
- 1.15s to write all data to disk



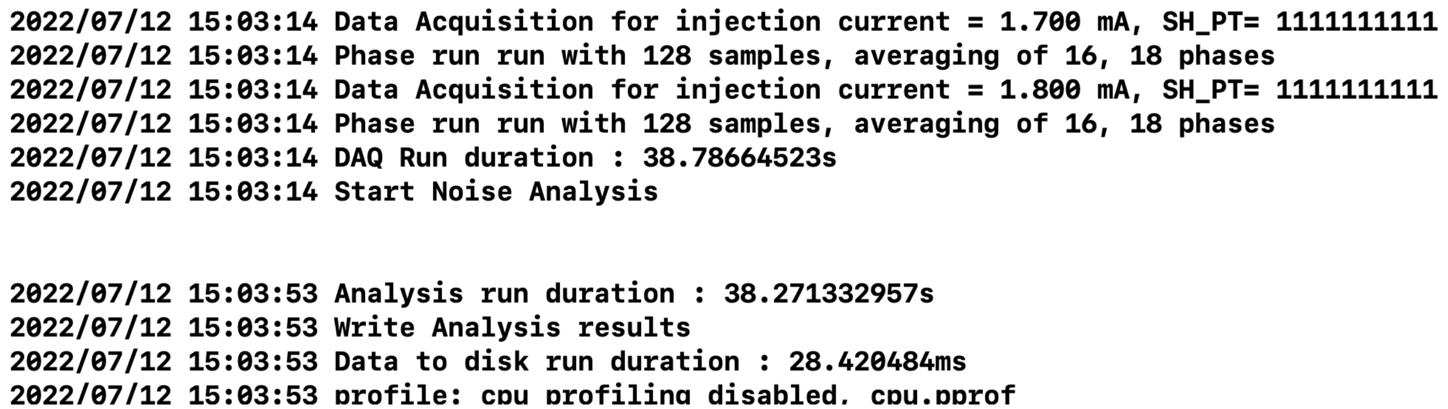
```
2022/07/12 14:56:35 DAQ Run duration : 2.566882968s
2022/07/12 14:56:35 Start Linearity Analysis
2022/07/12 14:56:36 Processing Channel  7
2022/07/12 14:56:36 Computing maximums
2022/07/12 14:56:36 Processing Channel  0
2022/07/12 14:56:36 Computing maximums
2022/07/12 14:56:36 Processing Channel  3
2022/07/12 14:56:36 Computing maximums
2022/07/12 14:56:36 Processing Channel  4
2022/07/12 14:56:36 Computing maximums
2022/07/12 14:56:36 Fitting
2022/07/12 14:56:36 Fitting
2022/07/12 14:56:36 Fitting
2022/07/12 14:56:36 Fitting
2022/07/12 14:56:36 Compute INL
2022/07/12 14:56:36 Compute INL
2022/07/12 14:56:36 Compute INL
2022/07/12 14:56:36 Compute INL
2022/07/12 14:56:36 CH 0 , Gain : 763.3260984281551 mV/mA
2022/07/12 14:56:36 CH 3 , Gain : 758.3098692173269 mV/mA
2022/07/12 14:56:36 CH 4 , Gain : 756.1269086644224 mV/mA
2022/07/12 14:56:36 CH 7 , Gain : 758.3278325200936 mV/mA
2022/07/12 14:56:36 Analysis run duration : 304.511475ms
2022/07/12 14:56:36 Write Analysis results
2022/07/12 14:56:37 Data to disk run duration : 1.165311943s
2022/07/12 14:56:37 Total time for scan : 4.059404582s
```


Linearity
Amplitude (V)
1250.00
750.00
250.00
0.25
1.00
1.75
INL (%)
0.15
0.05
-0.05
0.25
1.00
1.75
Injection DAC (A.U.)

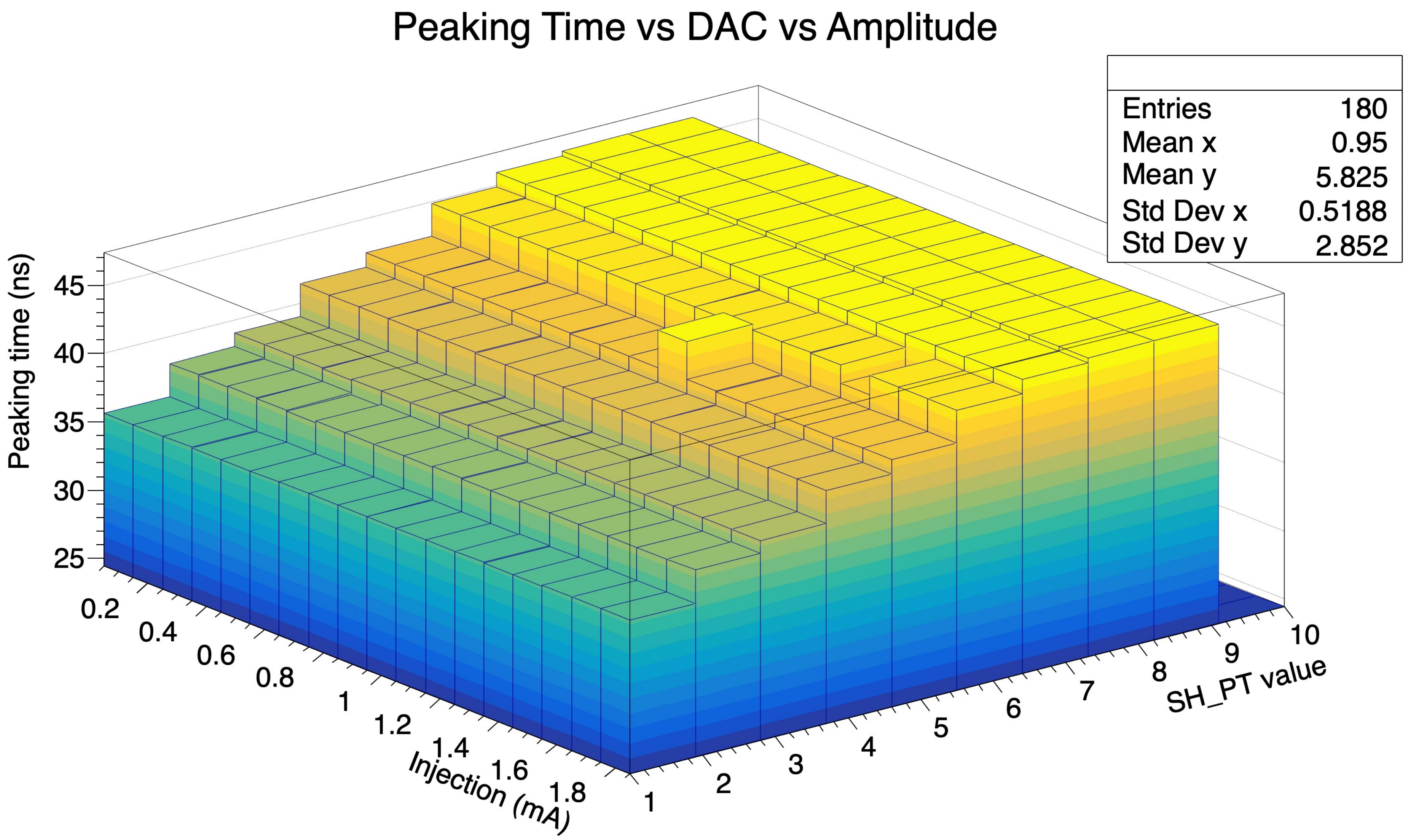
Peaking Time Scan

- Perform a pulse injection and measure pulse using a delay scan for each injection in the range , for each register value in range
- Extract pulse baseline and amplitude and peaking time
- Write raw data.Analysis results to ROOT file, CSV

Execution time (18 phases, 16 averaging, 18 steps, 10 DAC values) 76s



```
2022/07/12 15:03:14 Data Acquisition for injection current = 1.700 mA, SH_PT= 1111111111
2022/07/12 15:03:14 Phase run run with 128 samples, averaging of 16, 18 phases
2022/07/12 15:03:14 Data Acquisition for injection current = 1.800 mA, SH_PT= 1111111111
2022/07/12 15:03:14 Phase run run with 128 samples, averaging of 16, 18 phases
2022/07/12 15:03:14 DAQ Run duration : 38.78664523s
2022/07/12 15:03:14 Start Noise Analysis


2022/07/12 15:03:53 Analysis run duration : 38.271332957s
2022/07/12 15:03:53 Write Analysis results
2022/07/12 15:03:53 Data to disk run duration : 28.420484ms
2022/07/12 15:03:53 profile: cpu profiling disabled, cpu.pprof
```


Peaking Time vs DAC vs Amplitude
Entries 180
Mean x 0.95
Mean y 5.825
Std Dev x 0.5188
Std Dev y 2.852
Peaking time (ns)
45
40
35
30
25
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
Injection (mA)
1
2
3
4
5
6
7
8
9
10
SH_PT value

Ressources to learn Go

Basic, interactive online introduction :

- https://tour.golang.org/welcome/1

Interesting video that go a bit more in depth :

https://research.swtch.com/gotour

https://vimeo.com/53221560

https://www.youtube.com/watch?v=f6kdp27TYZs

Installing go : https://golang.org/doc/install

Code examples

- Various simple wrappers for interfaces such as I2C, SPI, AXI, SCPI memory access, etc.
  - https://gitlab.cern.ch/bnl-omega-go
- Small tool to configure Si5345
  - https://gitlab.cern.ch/bnl-omega-go/si5345tool
- Small tool for INA228
  - https://gitlab.cern.ch/bnl-omega-go/ina228tool

Autocorrelation

$$r_k = \frac{\sum_{i=1}^{N-k}(Y_i - \bar{Y})(Y_{i+k} - \bar{Y})}{\sum_{i=1}^{N}(Y_i - \bar{Y})^2}$$

In the autocorrelation formula, there is a nested loop that can take a lot of computing time for long waveform.

To make use of all CPU on a machine**, we launch a goroutine for each coefficient k, sent the result via a channel, and gather the result to communicate to the writer .**

Notice , no mention of mutex, semaphore etc… Here, channel are used to handle concurrency. The Data handle the thread synchronization, instead of thread handling data synchronization in usual languages

https://gitlab.cern.ch/BNL-ATLAS/larphase2/analysistools/gowfanalysis/-/tree/master/autocorrelation

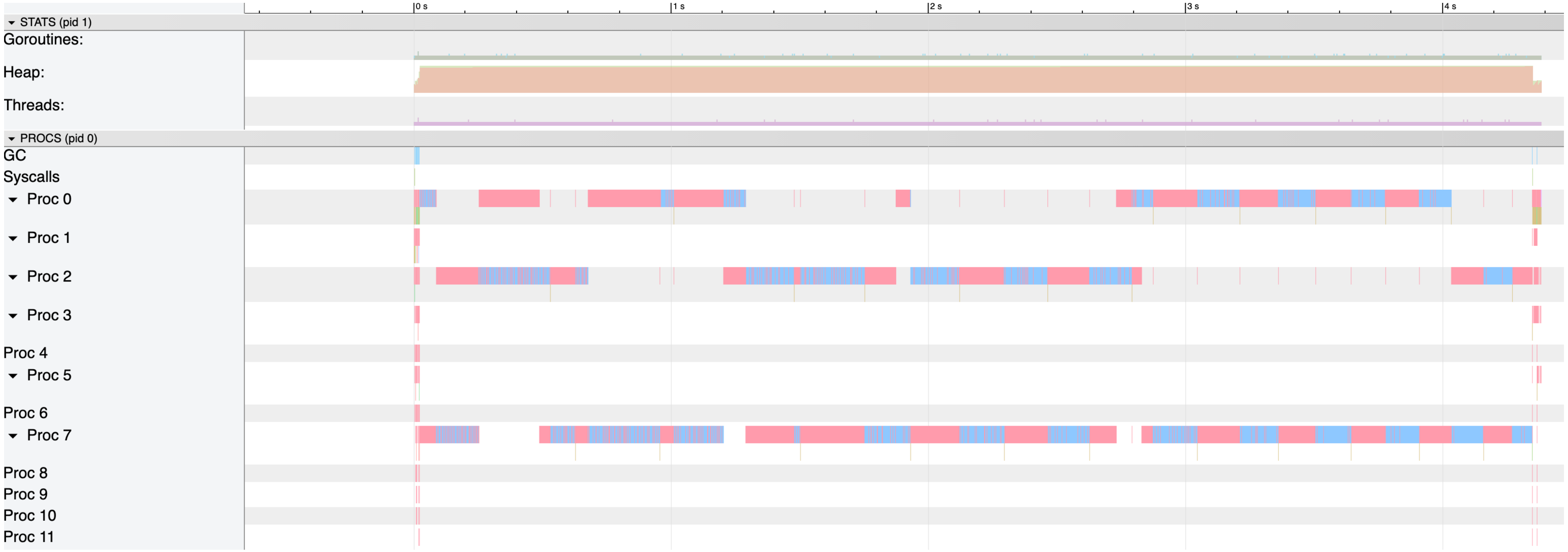
Interfaces

Using the Reader and Writer interfaces, we can implement many different reader and writer, as long as they expect the same bytes in input or output, this way we implement :

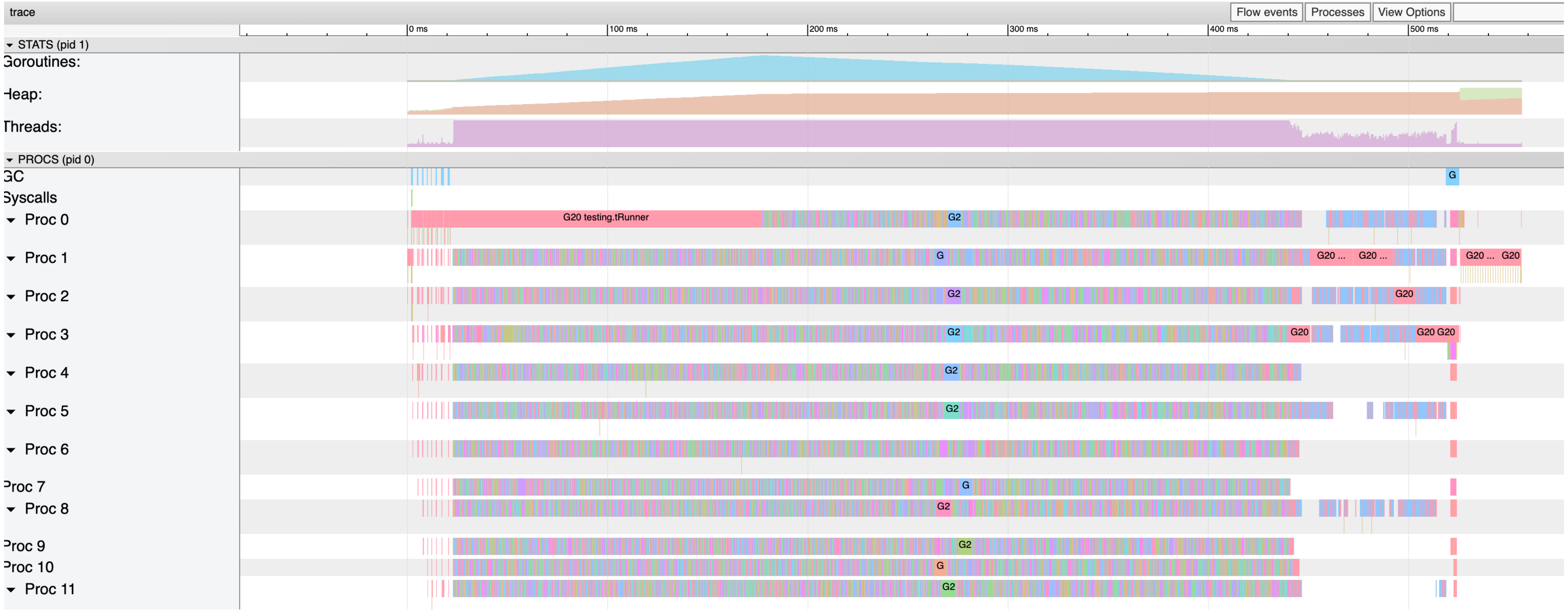
- TXT reader, get waveform from txt file
- Random reader, get random generated signal
- Sine Reader, generate a sine signal
- CSV reader, read a CSV using standard library

- TXT file writer
- CSV writer (using standard library)
- Cipher writer (encrypting data !)
- PlotWriter , sending data to a plot in PNG

Trace in single thread mode

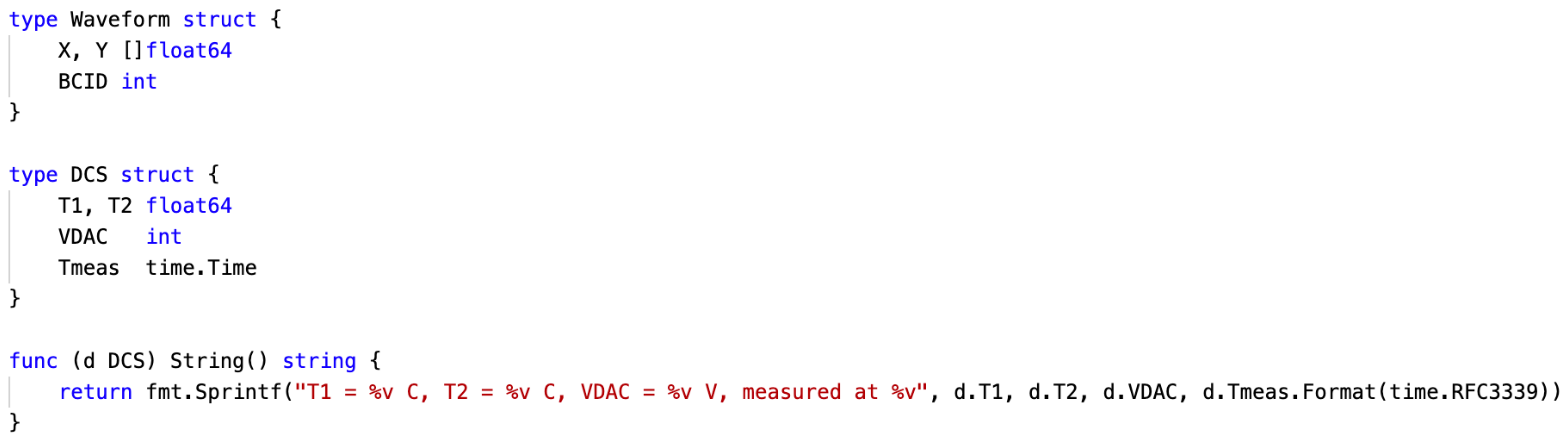


Trace in multithreaded mode



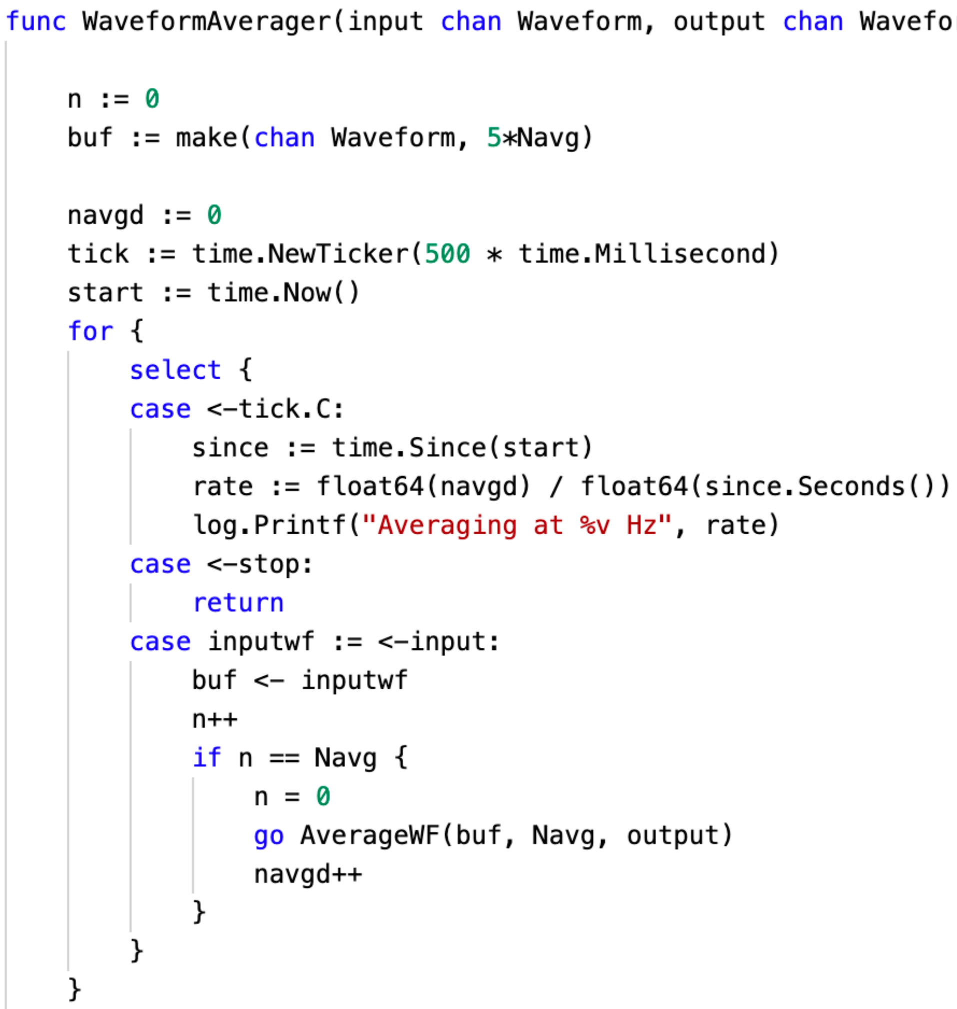
OAK RIDGE National Laboratory

Data struct for IO



```go
type Waveform struct {
    X, Y []float64
    BCID int
}

type DCS struct {
    T1, T2 float64
    VDAC   int
    Tmeas  time.Time
}

func (d DCS) String() string {
    return fmt.Sprintf("T1 = %v C, T2 = %v C, VDAC = %v V, measured at %v", d.T1, d.T2, d.VDAC, d.Tmeas.Format(time.RFC3339))
}
```

Averaging on all CPU efficiently



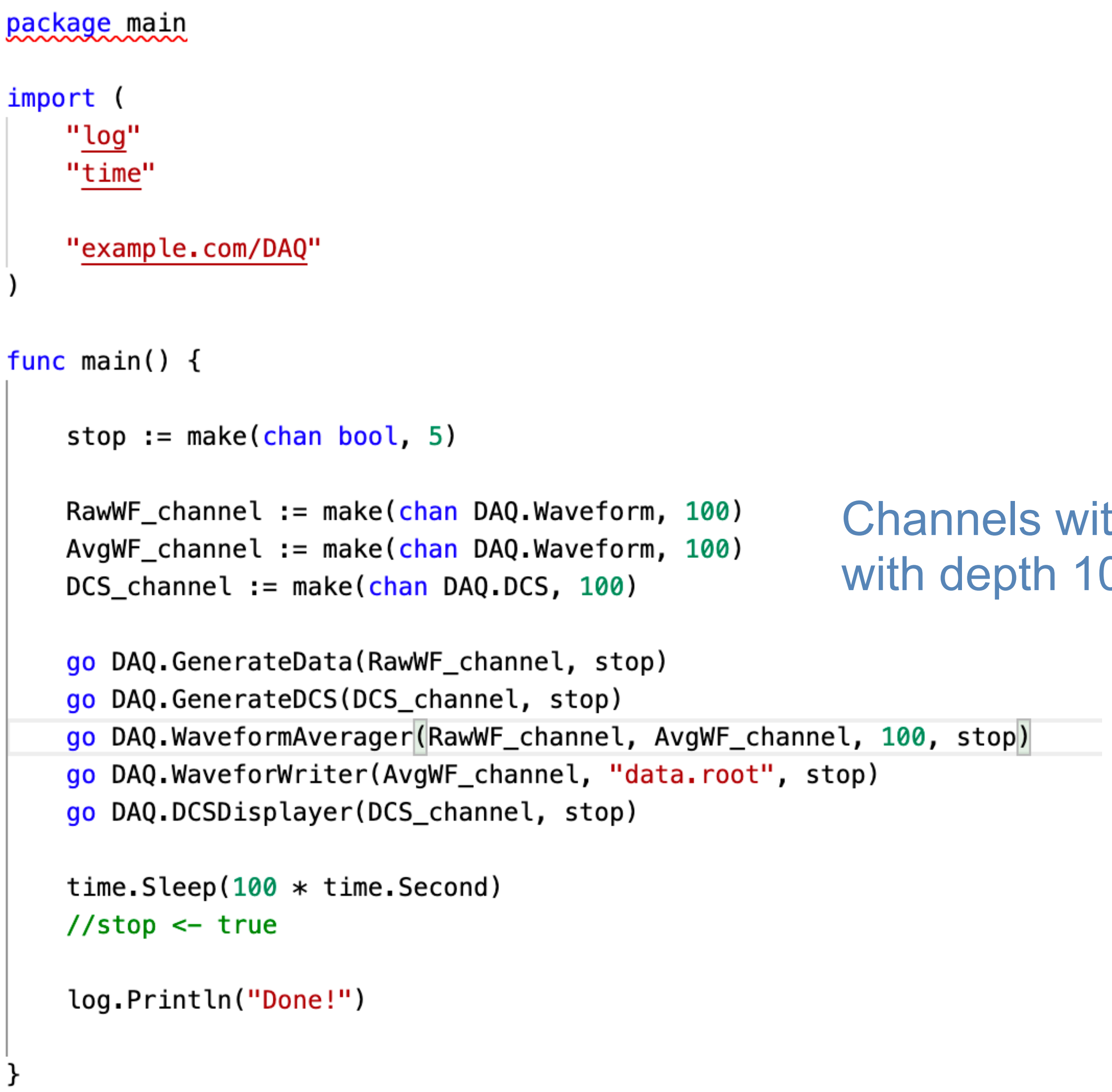
```go
func WaveformAverager(input chan Waveform, output chan Waveform, Navg int, stop chan bool) {

    n := 0
    buf := make(chan Waveform, 5*Navg)

    navgd := 0
    tick := time.NewTicker(500 * time.Millisecond)
    start := time.Now()
    for {
        select {
        case <-tick.C:
            since := time.Since(start)
            rate := float64(navgd) / float64(since.Seconds())
            log.Printf("Averaging at %v Hz", rate)
        case <-stop:
            return
        case inputwf := <-input:
            buf <- inputwf
            n++
            if n == Navg {
                n = 0
                go AverageWF(buf, Navg, output)
                navgd++
            }
        }
    }
```

Tick generate an event on a channel tick.C every 500 ms

Display rate every 500ms

If we have put enough waveform in the buffer, launch an Average goroutine to digest it

Assembling the code

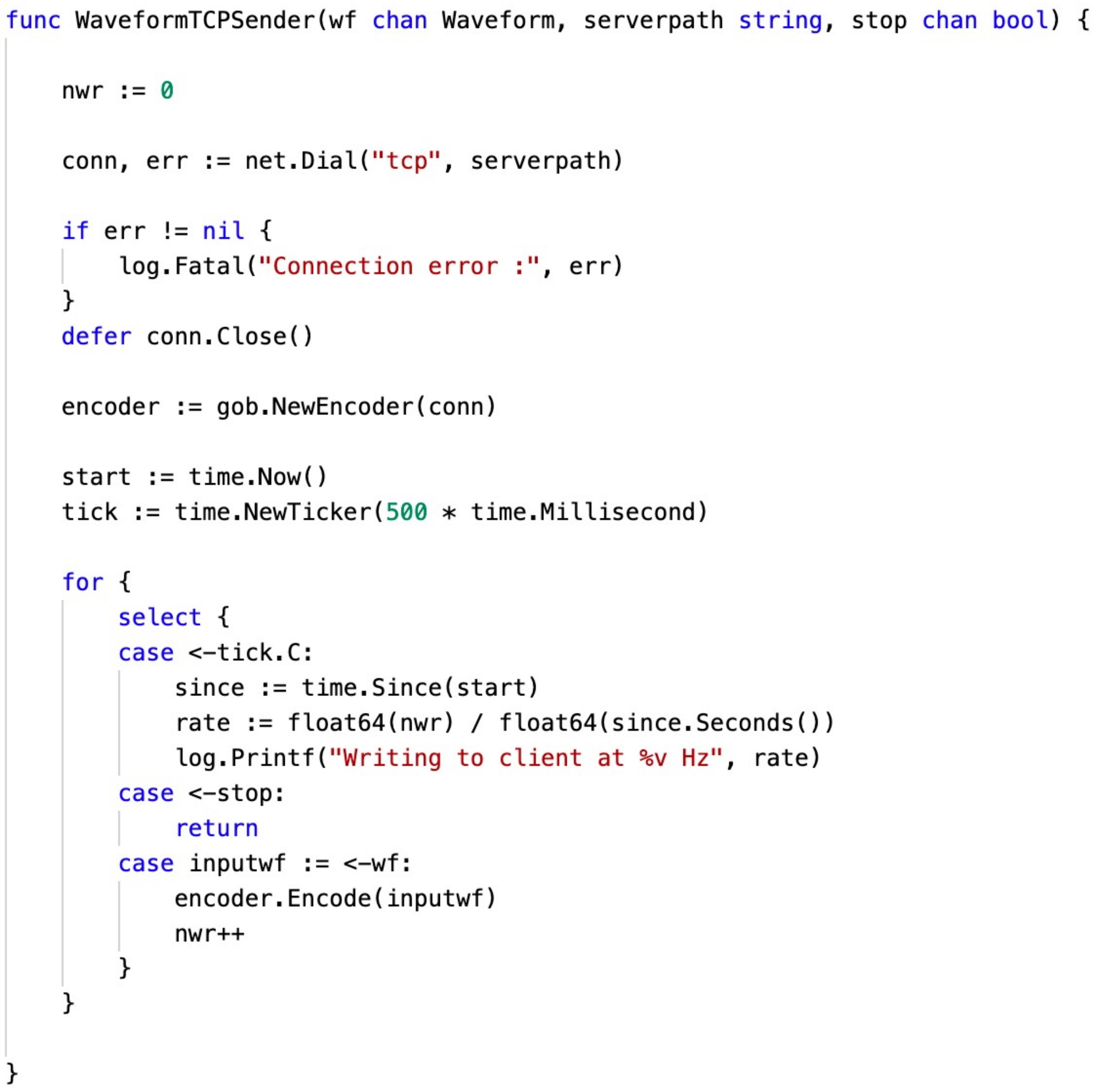


```go
package main

import (
	"log"
	"time"

	"example.com/DAQ"
)

func main() {

	stop := make(chan bool, 5)

	RawWF_channel := make(chan DAQ.Waveform, 100)
	AvgWF_channel := make(chan DAQ.Waveform, 100)
	DCS_channel := make(chan DAQ.DCS, 100)

	go DAQ.GenerateData(RawWF_channel, stop)
	go DAQ.GenerateDCS(DCS_channel, stop)
	go DAQ.WaveformAverager(RawWF_channel, AvgWF_channel, 100, stop)
	go DAQ.WaveforWriter(AvgWF_channel, "data.root", stop)
	go DAQ.DCSDisplayer(DCS_channel, stop)

	time.Sleep(100 * time.Second)
	//stop <- true

	log.Println("Done!")

}
```

Channels with buffers with depth 100

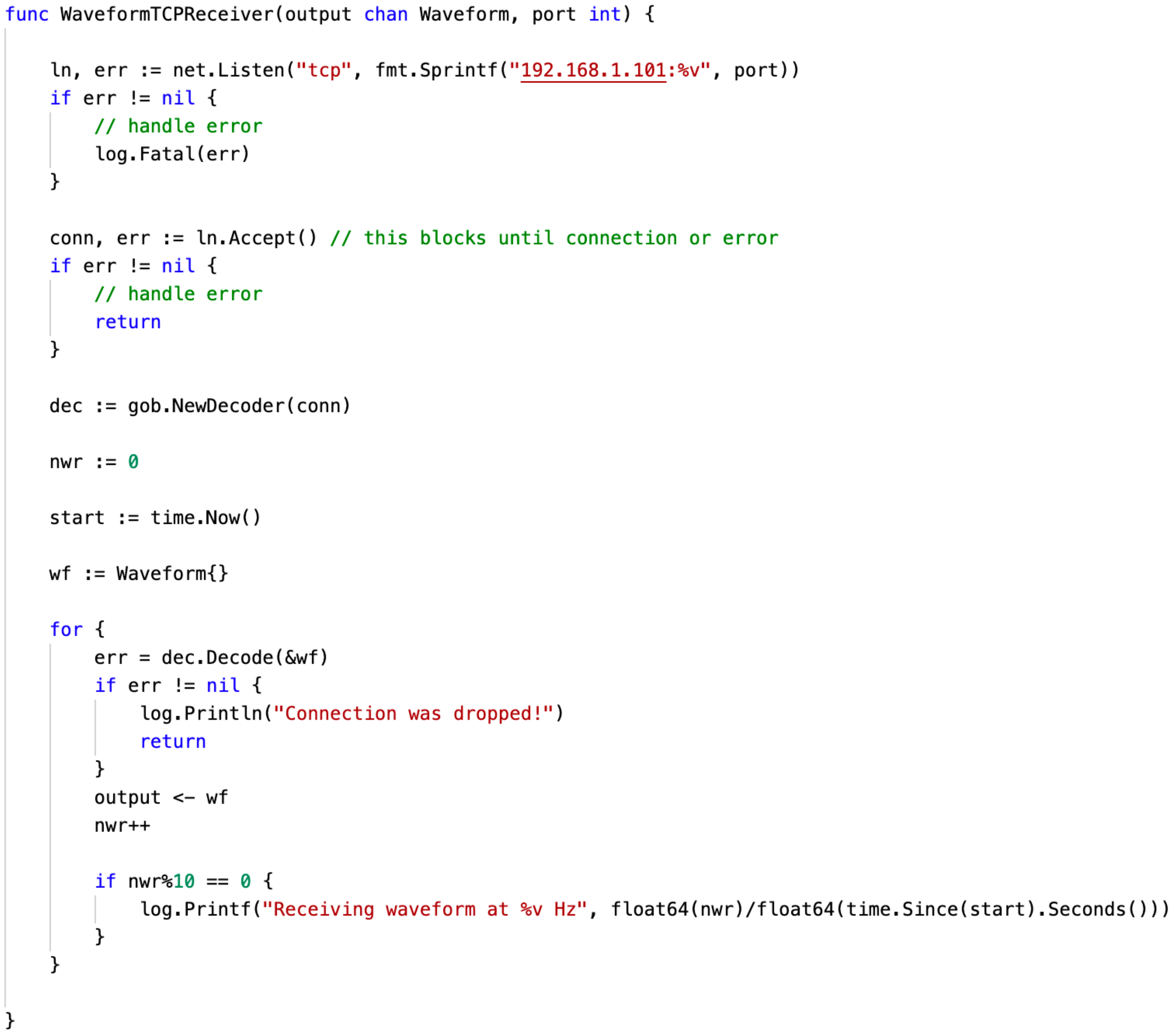
Go routines for each steps

Sending struct over network



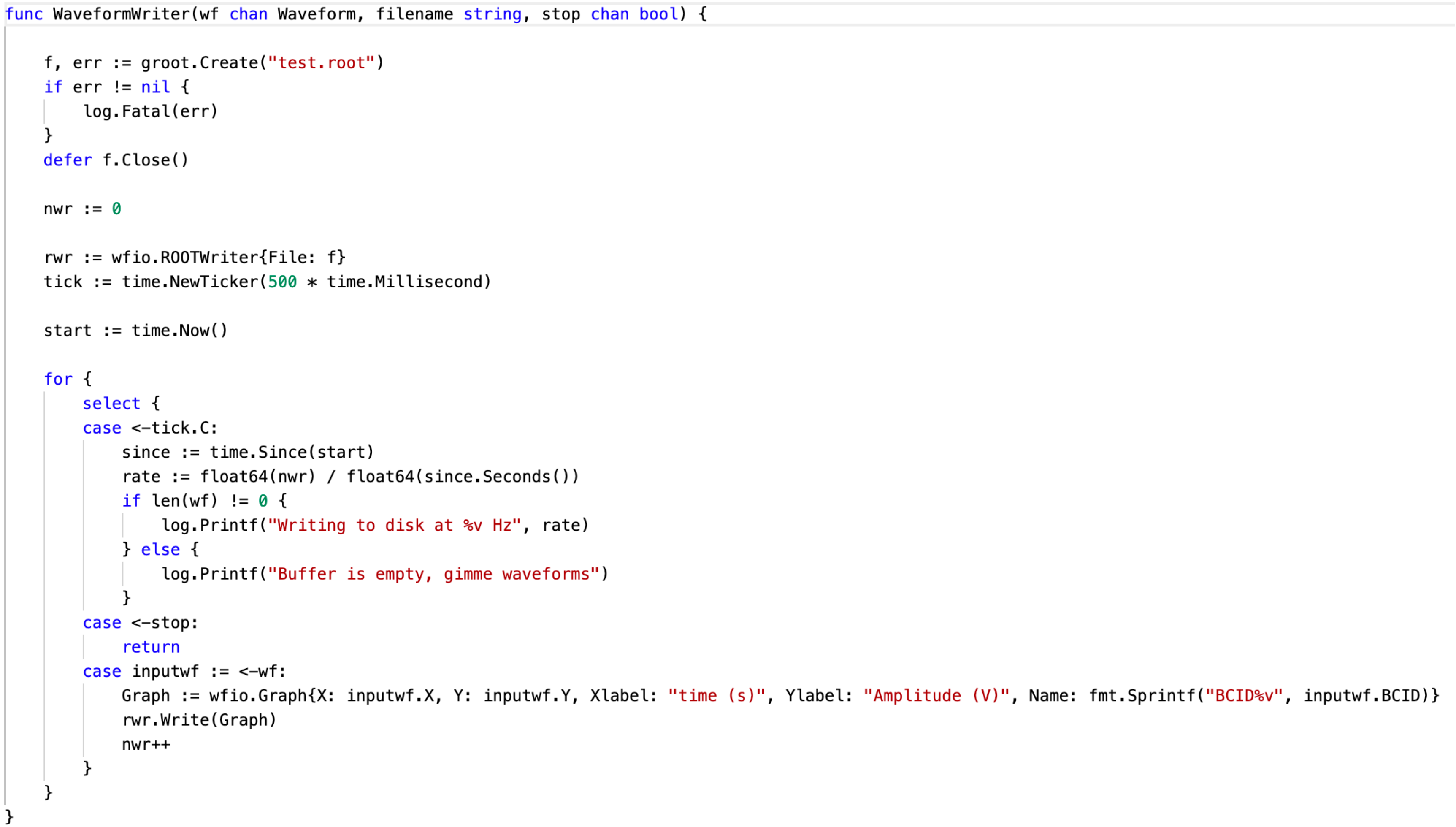
```go
func WaveformTCPSender(wf chan Waveform, serverpath string, stop chan bool) {

    nwr := 0

    conn, err := net.Dial("tcp", serverpath)

    if err != nil {
        log.Fatal("Connection error :", err)
    }
    defer conn.Close()

    encoder := gob.NewEncoder(conn)

    start := time.Now()
    tick := time.NewTicker(500 * time.Millisecond)

    for {
        select {
        case <-tick.C:
            since := time.Since(start)
            rate := float64(nwr) / float64(since.Seconds())
            log.Printf("Writing to client at %v Hz", rate)
        case <-stop:
            return
        case inputwf := <-wf:
            encoder.Encode(inputwf)
            nwr++
        }
    }

}
```

Receiving struct over network



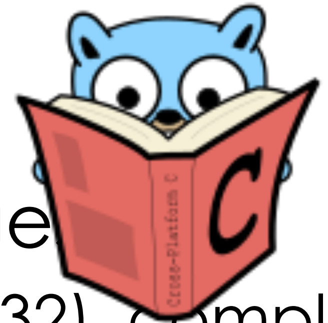
```go
func WaveformTCPReceiver(output chan Waveform, port int) {

    ln, err := net.Listen("tcp", fmt.Sprintf("192.168.1.101:%v", port))
    if err != nil {
        // handle error
        log.Fatal(err)
    }

    conn, err := ln.Accept() // this blocks until connection or error
    if err != nil {
        // handle error
        return
    }

    dec := gob.NewDecoder(conn)

    nwr := 0

    start := time.Now()

    wf := Waveform{}

    for {
        err = dec.Decode(&wf)
        if err != nil {
            log.Println("Connection was dropped!")
            return
        }
        output <- wf
        nwr++

        if nwr%10 == 0 {
            log.Printf("Receiving waveform at %v Hz", float64(nwr)/float64(time.Since(start).Seconds()))
        }
    }

}
```

Writing to ROOT file

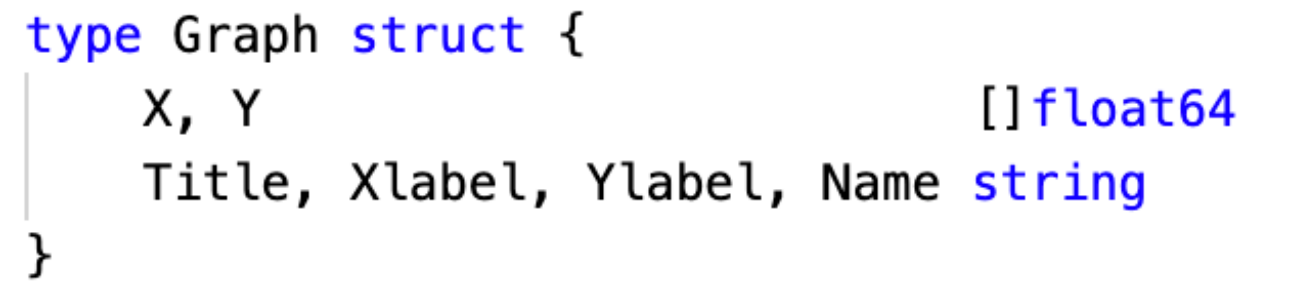


```go
func WaveformWriter(wf chan Waveform, filename string, stop chan bool) {

    f, err := groot.Create("test.root")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    nwr := 0

    rwr := wfio.ROOTWriter{File: f}
    tick := time.NewTicker(500 * time.Millisecond)

    start := time.Now()

    for {
        select {
        case <-tick.C:
            since := time.Since(start)
            rate := float64(nwr) / float64(since.Seconds())
            if len(wf) != 0 {
                log.Printf("Writing to disk at %v Hz", rate)
            } else {
                log.Printf("Buffer is empty, gimme waveforms")
            }
        case <-stop:
            return
        case inputwf := <-wf:
            Graph := wfio.Graph{X: inputwf.X, Y: inputwf.Y, Xlabel: "time (s)", Ylabel: "Amplitude (V)", Name: fmt.Sprintf("BCID%v", inputwf.BCID)}
            rwr.Write(Graph)
            nwr++
        }
    }
}
```

Go concepts type and variables

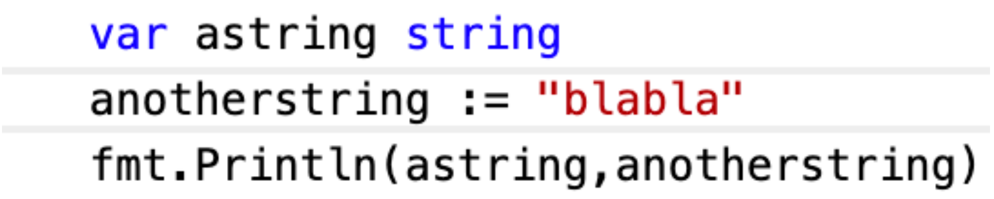


- Basic types are similar to other language
  - int, float, bool, string, byte (uint8), rune (uint32), complex
- Composite types are also provided
  - Struct, slice (array), map, **channel, error**
- Types always have a default value
  - 0 for int, 0.0 for float, nil for composite objects like error
- Vc

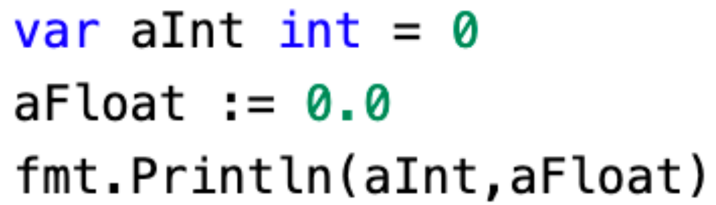


```go
type Graph struct {
	X, Y                        []float64
	Title, Xlabel, Ylabel, Name string
}
```

nplicitly

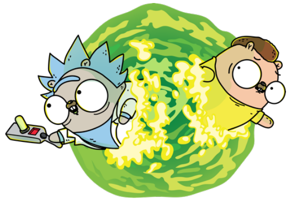


```go
var astring string
anotherstring := "blabla"
fmt.Println(astring,anotherstring)
```

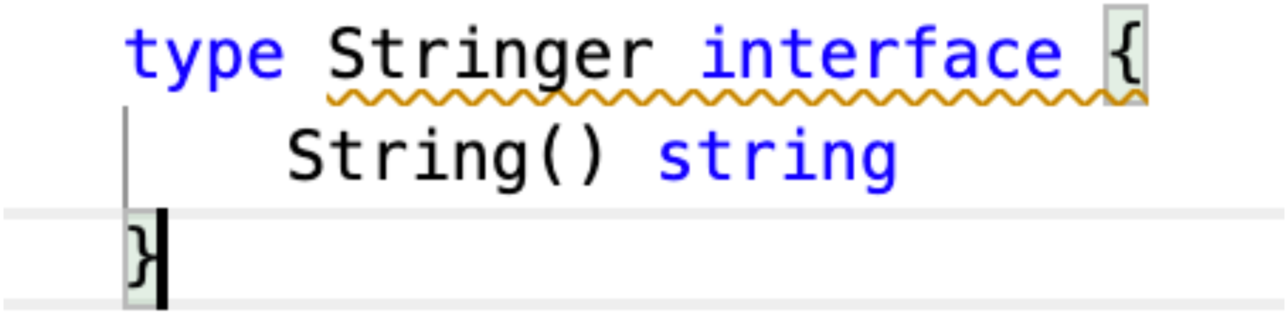


```go
var aInt int = 0
aFloat := 0.0
fmt.Println(aInt,aFloat)
```

Types interface and functions



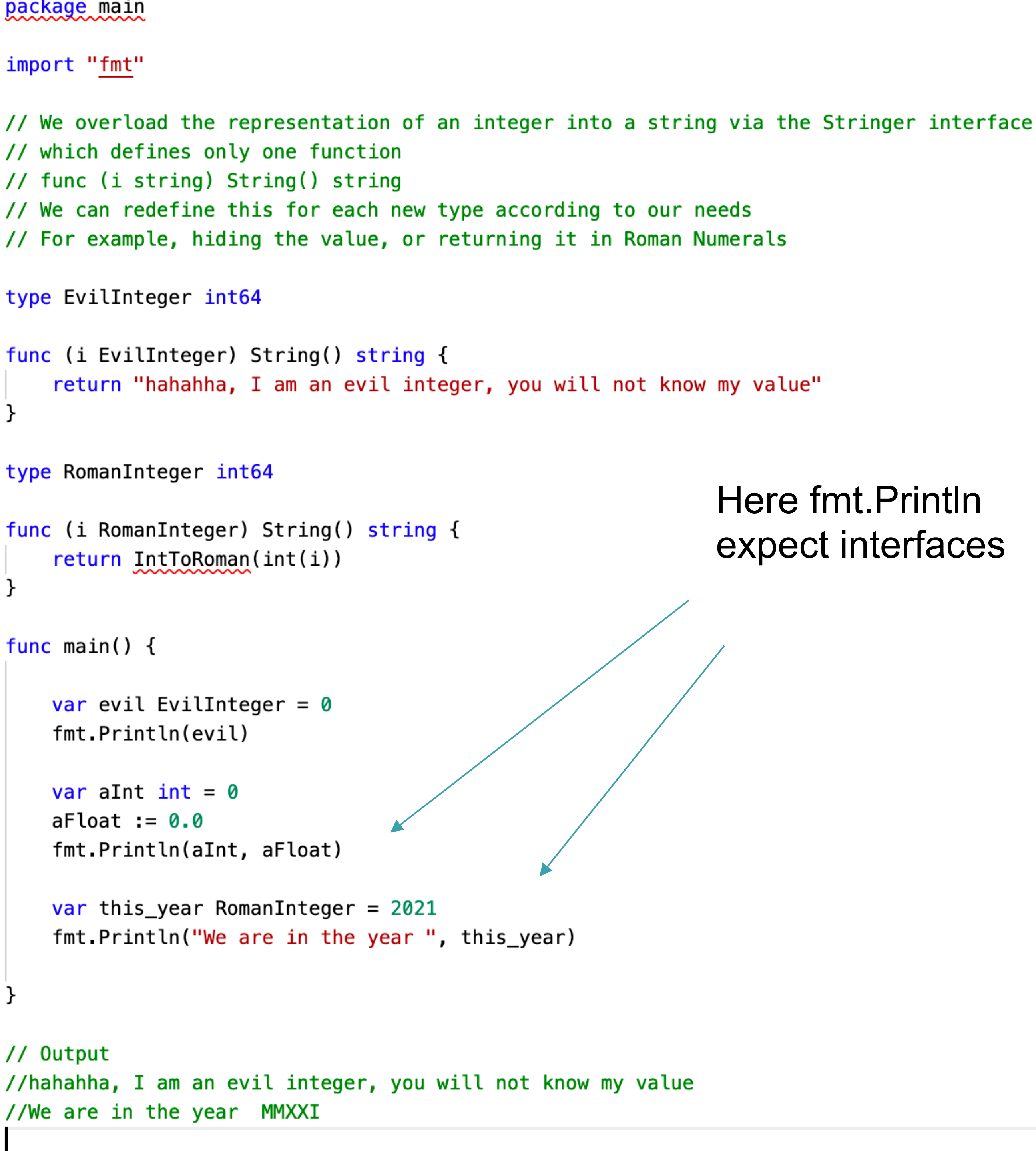
In go, any types can be attributed functions, some special functions, called Interfaces, define specific type of behavior for your type , for example the Stringer Interface :



```
type Stringer interface {
	String() string
}
```

In go, any type that implement the function String() that return a string, implements the stringer interface , which is itself a special kind of type.
Interfaces are an extremely simple, yet extremely powerful concept in go

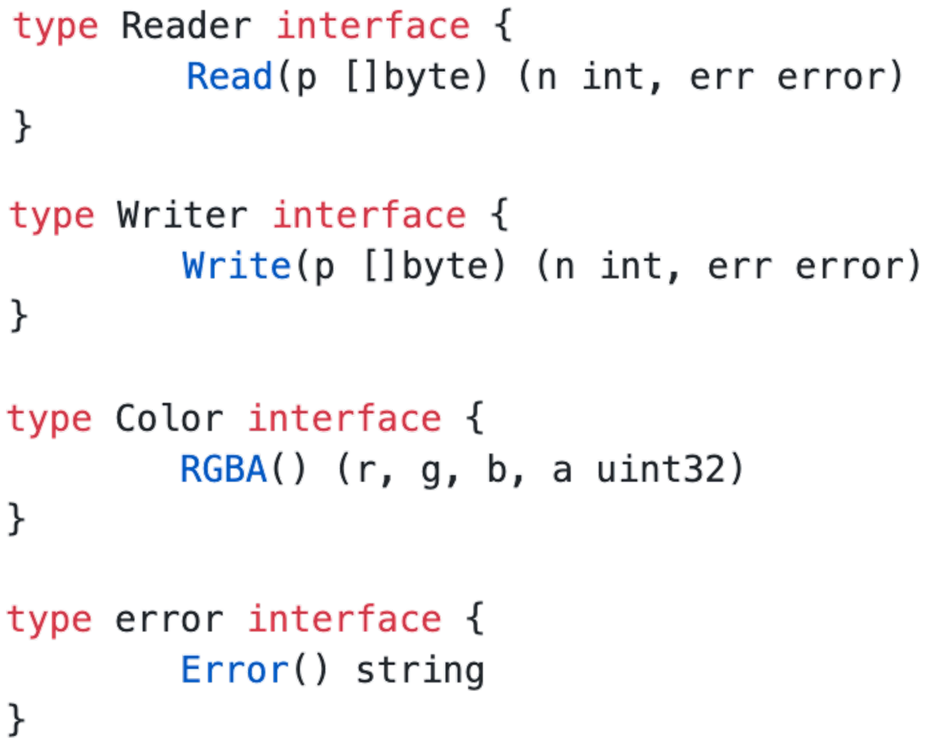
Example



```go
package main

import "fmt"

// We overload the representation of an integer into a string via the Stringer interface
// which defines only one function
// func (i string) String() string
// We can redefine this for each new type according to our needs
// For example, hiding the value, or returning it in Roman Numerals

type EvilInteger int64

func (i EvilInteger) String() string {
    return "hahahha, I am an evil integer, you will not know my value"
}

type RomanInteger int64

func (i RomanInteger) String() string {
    return IntToRoman(int(i))
}

func main() {

    var evil EvilInteger = 0
    fmt.Println(evil)

    var aInt int = 0
    aFloat := 0.0
    fmt.Println(aInt, aFloat)

    var this_year RomanInteger = 2021
    fmt.Println("We are in the year ", this_year)

}

// Output
//hahahha, I am an evil integer, you will not know my value
//We are in the year  MMXXI
```

Here fmt.Println expect interfaces

Other common interfaces



```
type Reader interface {
        Read(p []byte) (n int, err error)
}

type Writer interface {
        Write(p []byte) (n int, err error)
}

type Color interface {
        RGBA() (r, g, b, a uint32)
}

type error interface {
        Error() string
}
```

Something that read n bytes into p

Something that write p

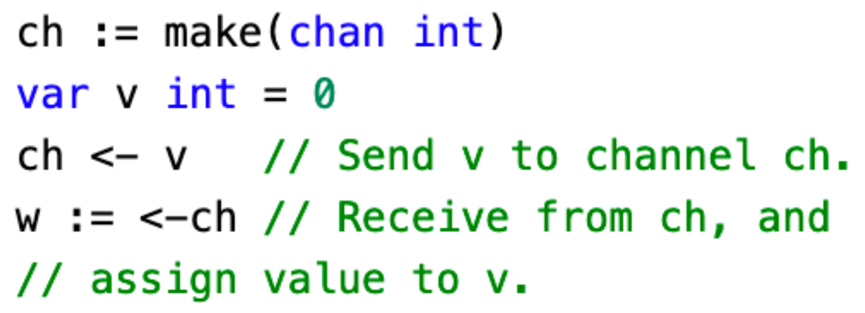
Something that has a color

A type of error that occured

https://gist.github.com/asukakenji/ac8a05644a2e98f1d5ea8c299541fce9

Channels

Channels are a typed conduit through which you can send and receive values with the channe

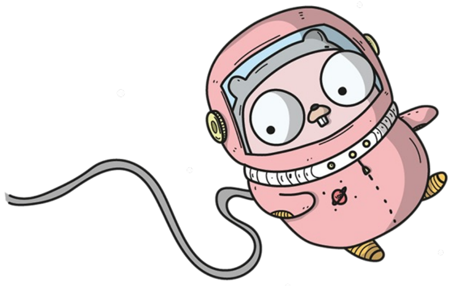


```
ch := make(chan int)
var v int = 0
ch <- v    // Send v to channel ch.
w := <-ch // Receive from ch, and
// assign value to v.
```

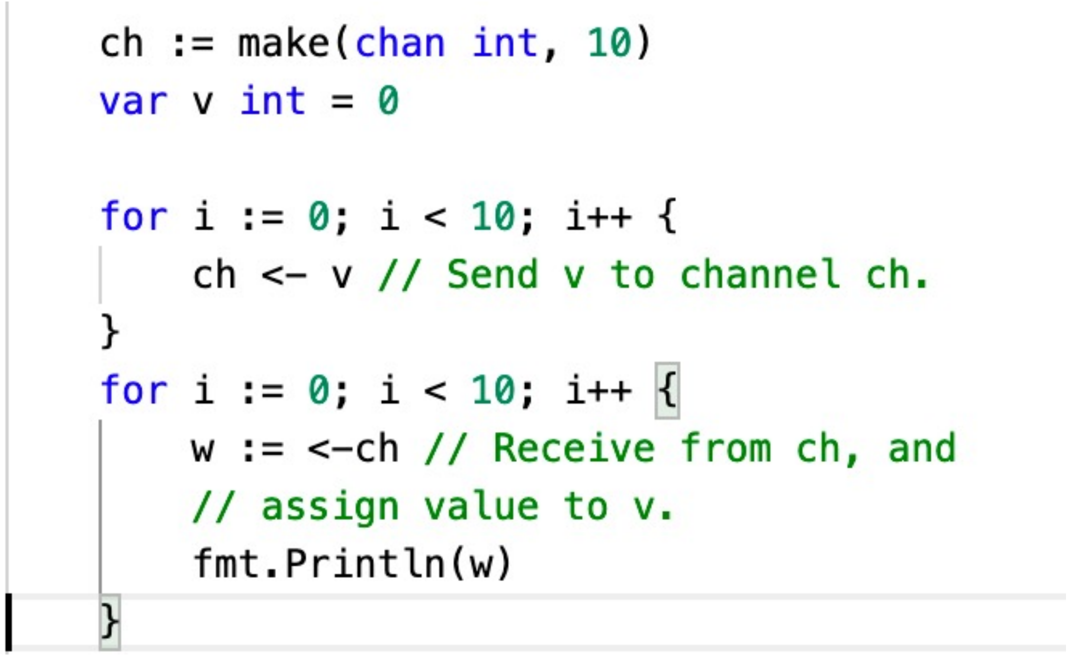
Reading from a channel is a blocking operation, the process will stop until data is available on the channel

Channels can be passed to functions, and returned by them

Channels



Channels can also be buffered :

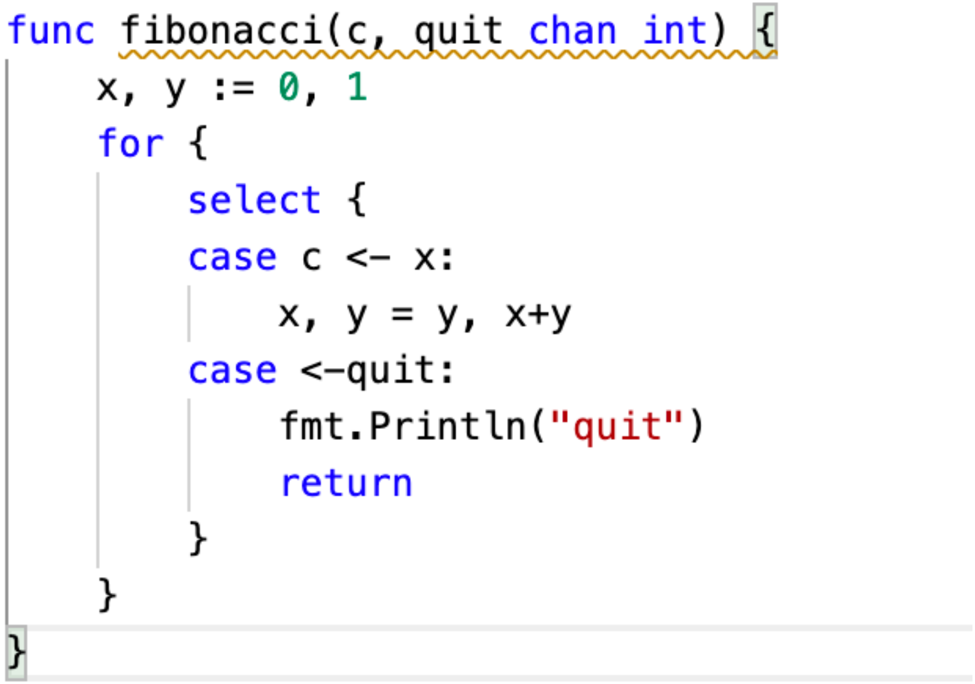


```go
ch := make(chan int, 10)
var v int = 0

for i := 0; i < 10; i++ {
    ch <- v // Send v to channel ch.
}
for i := 0; i < 10; i++ {
    w := <-ch // Receive from ch, and
    // assign value to v.
    fmt.Println(w)
}
```

Channels are a key concept to implement concurrency and parallelism in go

Select statements

A select statement is used to have the current process wait for communication operation from ch

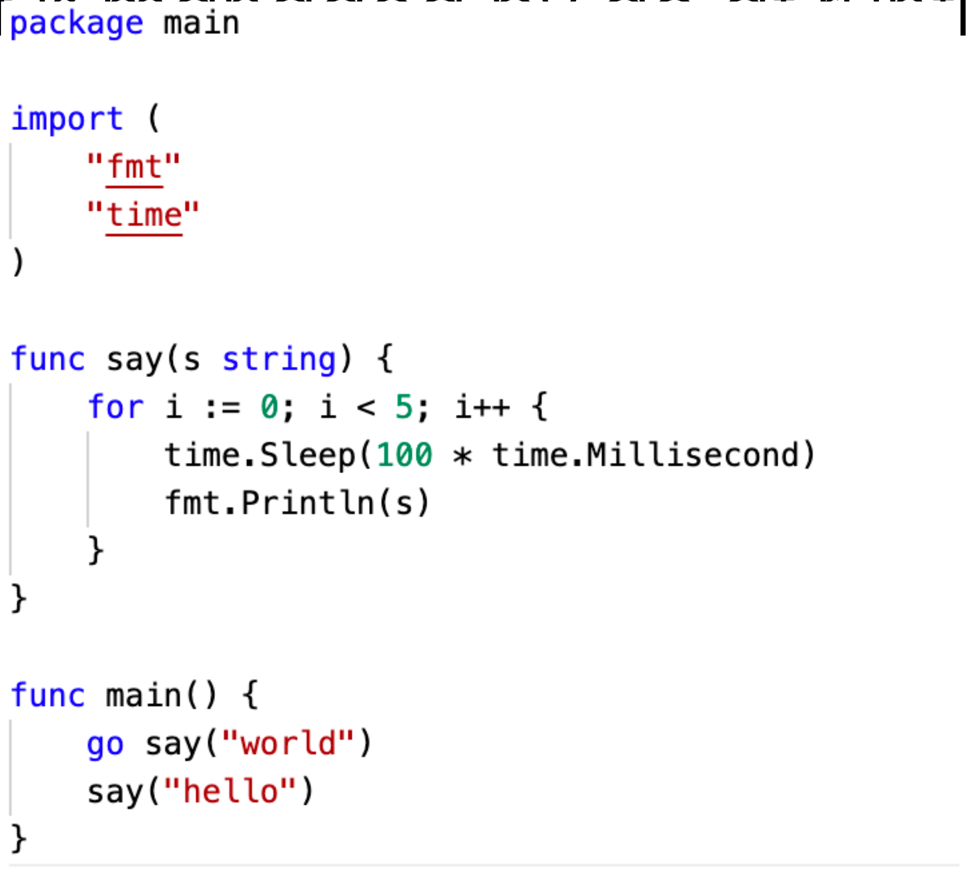


```go
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}
```

The process wait until of of the channel send data, if multiple have data, the execution is random

Goroutines

The final building block of golang are goroutines. A Goroutine is a light-weight thread that is managed by go at runtime



```go
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```