Current and Future YARR in a Nutshell

ITk DAQ Workshop - 27.05.19

Timon Heim - LBNL

Outline

- Motivation and Goals
- Conceptual overview and design philosophy
  - Hardware controller
  - Front-End chip
  - Scans
- Data processing: decoding, histogramming, & analysis
- Next development steps

Motivation

- Example IBL: lab testing → USBPix, stave testing → RCE, operation → ROD/BOC
  - Each system had a different code base (sw + fw)
  - SW entangled with fw, not easy to simply migrate
  - Expert knowledge and maturity of sw lost or not available at next stage
  - Had a huge impact on DAQ for operation and how many developers were needed to get it running
- Want to maintain a mature sw and have experts be able to apply their knowledge over a broad spectrum of test scales

Goals

- For ITk we should strive to have software base which can be used for small scale lab tests as well full detector operation
- SW should be common to Pixels and Strips
- This results in certain requirements:
  - Agnostic to hardware/firmware
  - Somewhat agnostic to Front-End chip type
  - Scalable in both senses (small and large)

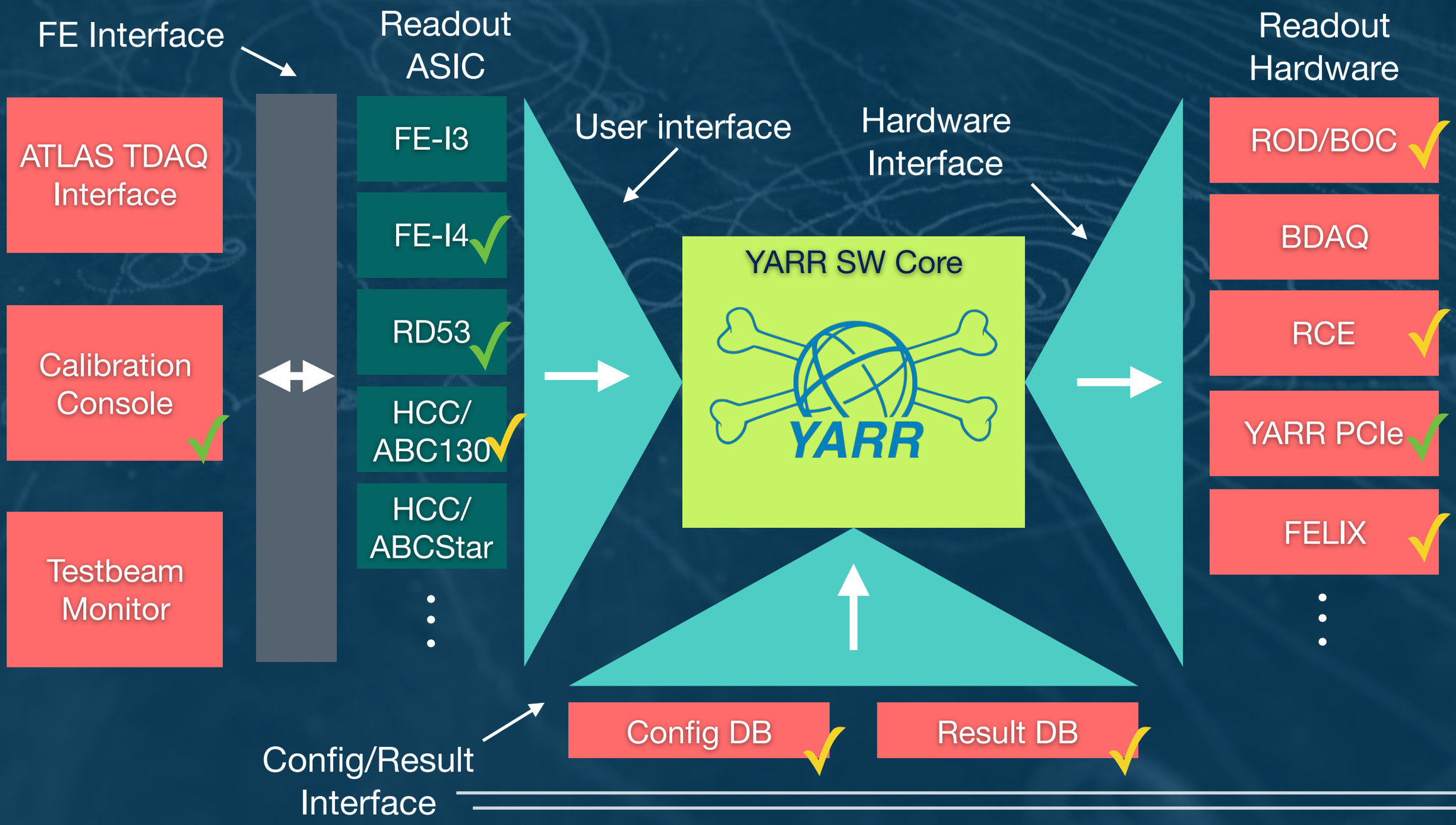
Where does YARR come from?

- YARR was originally designed as readout system using PCIe FPGA cards for FE-I4 (IBL)
- It tried to perform as much processing as possible in SW
- This resulted in minimum entanglement with FW, the PCIe card simply acted as a FIFO
- After some abstraction of the hardware and chip interface YARR seemed useable as a basis to be expanded and used as ITk SW
- However i.e. there are some remnant from the old days:
  - Certain things are general but still in the FE-I4 class
  - The hardware interface has certain functions which are driven by the features of the YARR-FW for PCIe (primarily naming)

Conceptual Overview

YARR SW core:

- Common sw core used in small lab systems up to full detector readout
- Improvements to core transfer to all DAQ systems
- Well defined interfaces required



Design Philosophies

- Simple firmware, smart software: the more we do in software, the less we are bound to specific features in hw/fw → hardware agnostic
- Keep it modular: the more often code and structures can be re-used, the better → Front-End chips are more alike than you might think
- Simplicity can be key: carefully balance performance and simplicity, we are bad at writing documentation and the code has to live for the next 10+ years, still don't want it to be slow of course
- Pipeline it: wherever possible data should only travel in one direction, avoid process interdependency → eases scaling

Hardware Controller

Assumptions:

- Interaction with chip can be broken down into sending and receiving of data
  - Represented by the Tx and RxCore in the interface
- Sending of commands:
  - Small to medium bandwidth
  - Primarily configuration data
  - Broadcast wherever possible
  - Some support from firmware, but not necessary for everything
- Receiving of data:
  - Max. bandwidth
  - One data stream per front-end object
  - Will enter processing chain
  - Only rudimentary decoding done in hardware
- Sending either to one or all chips*
- Receiving from all chips (demultiplexing in sw)*

*see future section

TxCore

FiFo style sending of commands

```
virtual void writeFifo(uint32_t) = 0;
virtual void releaseFifo() = 0;
virtual void setCmdEnable(uint32_t) = 0;
virtual uint32_t getCmdEnable() = 0;
virtual bool isCmdEmpty() = 0;
```

`writeFifo()` can be buffered in SW to increase package size, until `releaseFifo()` is called

Trigger interface

```
virtual void setTrigEnable(uint32_t value) = 0;
virtual uint32_t getTrigEnable() = 0;
virtual void maskTrigEnable(uint32_t value, uint32_t mask) = 0;
virtual bool isTrigDone() = 0;
virtual void setTrigConfig(enum TRIG_CONF_VALUE cfg) = 0;
virtual void setTrigFreq(double freq) = 0; // in Hz
virtual void setTrigCnt(uint32_t count) = 0;
virtual void setTrigTime(double time) = 0; // in s
virtual void setTrigWordLength(uint32_t length) = 0; // From Msb
virtual void setTrigWord(uint32_t *word, uint32_t length) = 0; // 4 words, start at Msb
virtual void toggleTrigAbort() = 0;
```

Interface to a buffer which can be send with a programmed frequency.
Can do this from either sw or fw (but better timing via fw).

https://gitlab.cern.ch/YARR/YARR/blob/master/src/libYarr/include/TxCore.h

RxCore

Reading data FiFo style

```
virtual void setRxEnable(uint32_t val) = 0;
virtual void maskRxEnable(uint32_t val, uint32_t mask) = 0;

virtual RawData* readData() = 0;
virtual void flushBuffer() {}

virtual uint32_t getDataRate() = 0;
virtual uint32_t getCurCount() = 0;
virtual bool isBridgeEmpty() = 0;
```

SW will read until FiFo is empty

Raw data object

```
class RawData {
    public:
        RawData(uint32_t arg_adr, uint32_t *arg_buf, unsigned arg_words);
        ~RawData();

        uint32_t adr;
        uint32_t *buf;
        unsigned words;
        LoopStatus stat;
};
```

If data give as pointer, does not copy data.

https://gitlab.cern.ch/YARR/YARR/blob/master/src/libYarr/include/RxCore.h
https://gitlab.cern.ch/YARR/YARR/blob/master/src/libYarr/include/RawData.h

Front-End Chip Implementation

- Chip only needs to implement a config file interface and basic configuration routines
- Advanced functions determined by scan needs which are not generic
- There will be one object for each chip
- A virtual copy of the chip config is saved within the object
- Wherever possible register should be referred to by object and not by string (can’t avoid this fully)

https://gitlab.cern.ch/YARR/YARR/blob/master/src/libYarr/include/FrontEnd.h

Scans

- Scans typically do the following:
  - Configure all activated chips
  - Run a loop actions as nested structure:
    - Loop over parameter
      - Activate portion of pixels
        - Inject & Trigger O(100) times
        - Read data
- This is facilitated by the scan engine
- Most loop actions are custom for each chip, there are some more general though

Scans II

- All data from one innermost loop iteration is packaged and meta data describing the current loop state is added
- These data packages are then run through the processing chain
- Most scans are fully described a-priori, except tunings:
  - Tunings require a parameter change which depends on the analysis outcome
  - FeedbackLoops facilitate the interface to the analysis and allow the analysis to change parameters
  - Usually use a “hot or cold” scheme, where the analysis only determines the direction and the LoopAction applies the correct parameter change (LoopAction is in charge of tuning Algorithm)

An Example Scan

Nesting direction: outermost → innermost

```
"loops": [
  {
    "config": {
      "enable_lcap": true,
      "enable_scap": true,
      "mask": 65537,
      "max": 16,
      "min": 0,
      "step": 1
    },
    "loopAction": "Fei4MaskLoop"
  },
  {
    "config": {
      "max": 4,
      "min": 0,
      "mode": 1,
      "step": 1
    },
    "loopAction": "Fei4DcLoop"
  },
  {
    "config": {
      "count": 100,
      "delay": 30,
      "extTrigger": false,
      "frequency": 1000,
      "noInject": false,
      "time": 0
    },
    "loopAction": "Fei4TriggerLoop"
  },
  {
    "loopAction": "StdDataLoop"
  }
],
"name": "DigitalScan",
```

LoopAction config

LoopAction name

Fei4MaskLoop:
Enables an n-th of all pixel, where n=max.

Fei4DcLoop:
Enables an n double columns, where n is defined by max/mode.

Fei4TriggerLoop:
Injects and triggers at specified frequency for specified number of times.

StdDataLoop:
Collect data during triggering.

Data Processing

Monolithic

Parallelised

Tuning Feedback

Scan Engine

Configure

Scan Loop

Read Data

Data Processor

Pick up Raw Data

Process and build Event

Histogrammer

Pick up Event

Histogrammer

Publish Histogram

Analyser

Pick up Histogram

Analysis

Publish Result

One thread per FE

Raw Data

Events

Events

Events

Histogram

Histogram

Histogram

Result

Result

Result

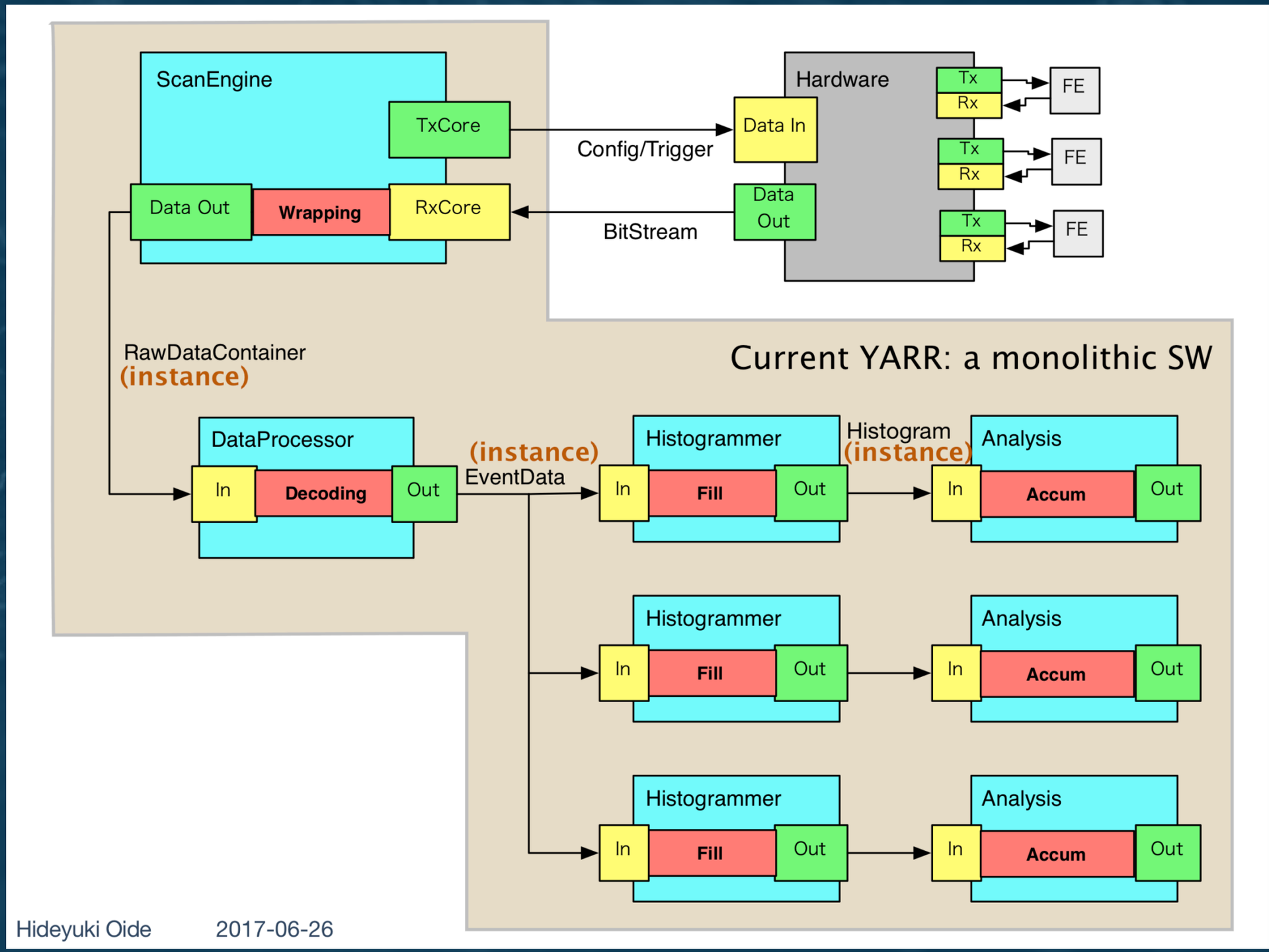
Bookkeeper

Algorithm

Thread

Datacontainer

Data Pipeline



scanConsole

- scanConsole is the main tool which will be used for calibration

```
[theim@Epsilon src]$ bin/scanConsole -h
#####################################
# Welcome to the YARR Scan Console! #
#####################################
-> Parsing command line parameters ...
Help:
 -h: Shows this.
 -s <scan_type> : Scan config
 -c <cfg1.json> [<cfg2.json> ...]: Provide connectivity configuration, can take multiple arguments.
 -r <ctrl.json> Provide controller configuration.
 -t <target_charge> [<tot_target>] : Set target values for threshold/charge (and tot).
 -p: Enable plotting of results.
 -o <dir> : Output directory. (Default ./data/)
 -m <int> : 0 = pixel masking disabled, 1 = start with fresh pixel mask, default = pixel masking enabled
 -k: Report known items (Scans, Hardware etc.)
```

Important command line arguments:

- -c : the connectivity tell scanConsole which chip is connected where and also points to the right chip config
- -r : the controller config tell scanConsole which controller to use and how to configure it
- -s : the scan config contains all necessary information to construct the scan

configs

example_rd53a_setup.json

```
{
    "chipType" : "RD53A",
    "chips" : [
        {
            "config" : "configs/rd53a_test.json",
            "tx" : 0,
            "rx" : 0,
            "enable" : 1,
            "locked" : 0
        },
        {
            "config" : "configs/rd53a_test_1.json",
            "tx" : 1,
            "rx" : 1,
            "enable" : 0,
            "locked" : 0
        }
    ]
}
```

specCfg.json

```
{
    "ctrlCfg" : {
        "type": "spec",
        "cfg" : {
            "specNum" : 0,
            "spiConfig" : 541200,
            "autoZero" : {
                "word" : 1549575846,
                "interval" : 500
            },
            "cmdPeriod" : 6.25e-9
        }
    }
}
```

configs II

std_digitalscan.json

```
{
  "scan": {
    "analysis": {
      "0": {
        "algorithm": "OccupancyAnalysis",
        "config": {
          "createMask": true
        }
      },
      "1": {
        "algorithm": "L1Analysis"
      },
      "n_count": 2
    },
    "histogrammer": {
      "0": {
        "algorithm": "OccupancyMap",
        "config": {}
      },
      "1": {
        "algorithm": "TotMap",
        "config": {}
      },
      "2": {
```

```
    "loops": [
      {
        "config": {
          "max": 64,
          "min": 0,
          "step": 1
        },
        "loopAction": "Rd53aMaskLoop"
      },
      {
        "config": {
          "max": 50,
          "min": 0,
          "step": 1,
          "nSteps": 25
        },
        "loopAction": "Rd53aCoreColLoop"
      },
      {
        "config": {
          "count": 100,
          "delay": 56,
          "extTrigger": false,
          "frequency": 18000,
          "noInject": false,
          "time": 0,
          "edgeMode": true
        },
        "loopAction": "Rd53aTriggerLoop"
      },
      {
        "loopAction": "StdDataLoop"
      }
    ],
    "name": "DigitalScan",
    "prescan": {
        "InjEnDig": 1,
        "InjAnaMode": 0,
        "LatencyConfig": 58,
        "GlobalPulseRt": 16384,
        "SyncVth": 500,
        "LinVth" : 500,
        "DiffVth1": 500
    }
  }
}
```

Work In Progress

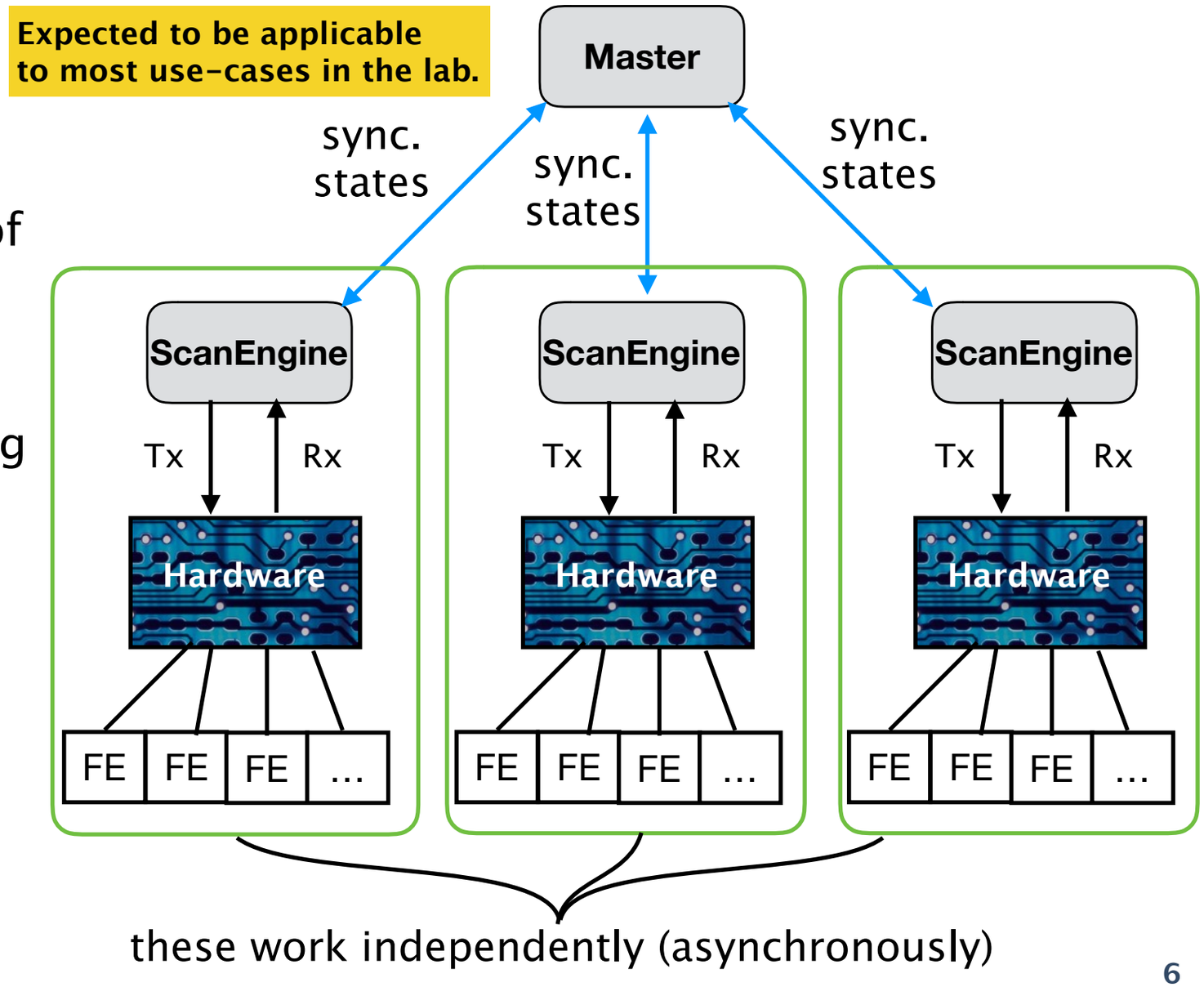
Scaling It Up

- How to scale this up?
  - Break pipeline into pieces distributed over multiple machines
  - Have multiple scan engines delivering data to a central or multiple central data processing servers
- Requires:
  - Orchestration of scan engines and data processors, distribution of configuration to all sub-processors (RPC)
  - Serialisation of data in between processes (IPC)

https://indico.cern.ch/event/609081/contributions/2636091/attachments/1483038/2300644/ItkWeek_SW_20170626.pdf

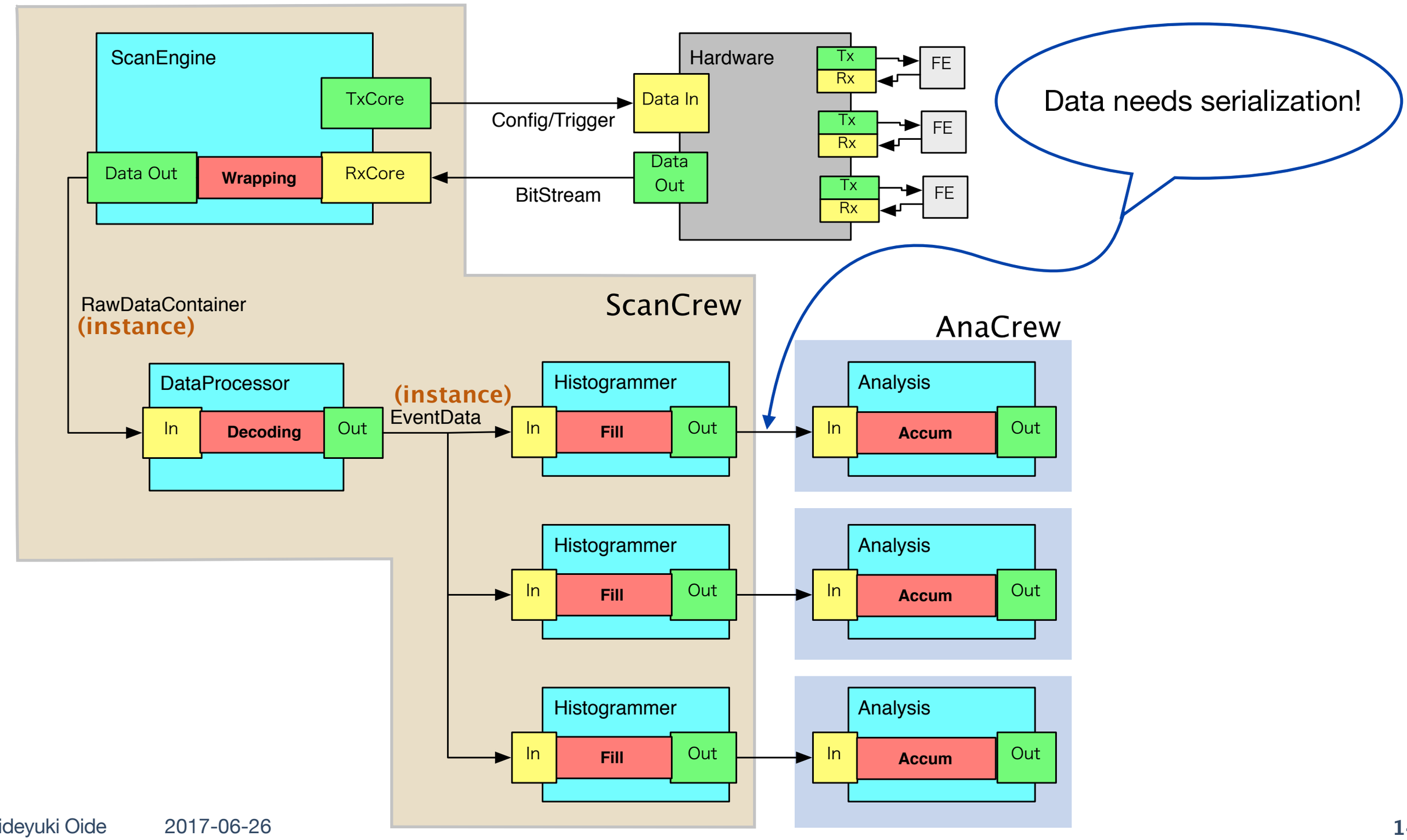
The current scan operation model being assumed

- For performing a scan, for each hardware, there is a software process which exclusively governs the control of configuration and trigger (the one that `TxCore()` and `RxCore()` are equipped). This part of the software module is referred to as “Scan Engine”.
- Each scan engine is agnostic to the presence of the other hardwares.
- Each hardware board works in parallel between the start and the end of the scan, but they do the same scan task.
- The organization of multiple scan engines is synchronization of the configuration and states (in terms of slow control) via **high-level messaging**.
- The data processing (histogramming and analysis) may be performed locally, or delegated to the specialized computing farm allocated in the downstream. (Arbitrarieness of the arrangement should be ensured.)

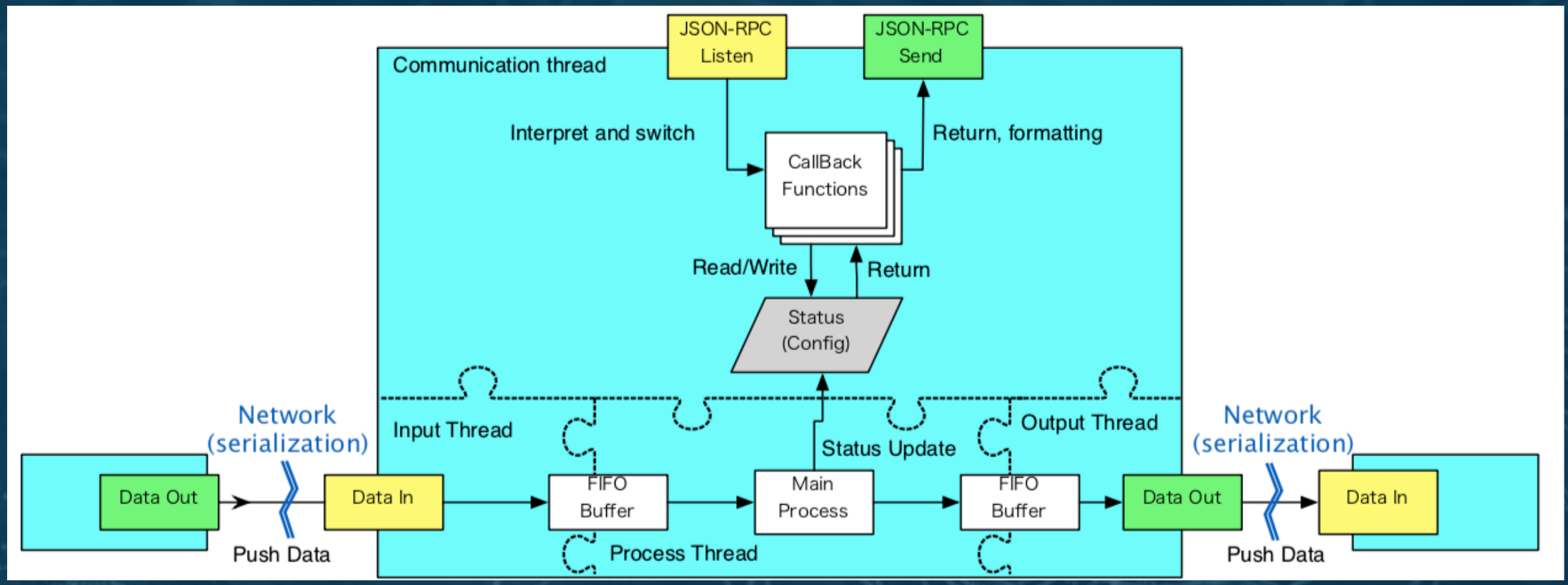


Data flow design

- Wish to have flexibility of grouping of the function modules within a process.
- Object data need to support serialization.



Generic Data Processor

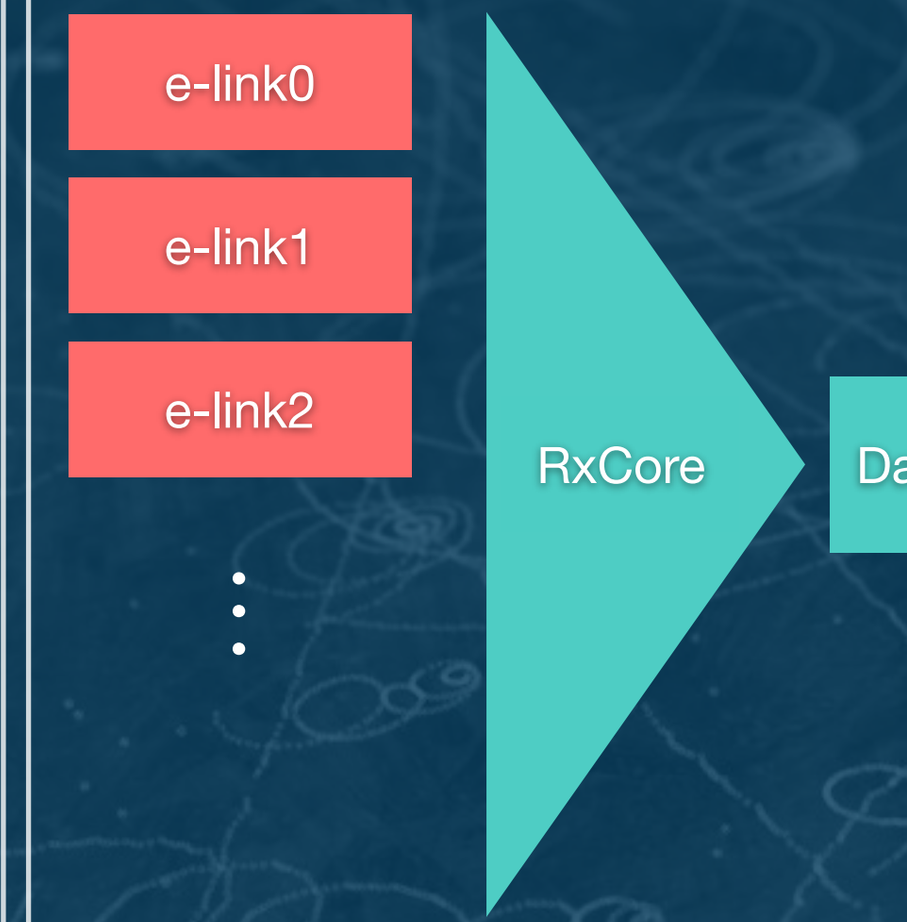


- Already heavily rely on nlohmann::json, convenient format also for serialisation
- Use msgpack to serialise json object
- Where json is too costly in terms of memory or bandwidth, usually already have handy RawData format
- By performing some optimisation to nlohmann::json could even be used as a histogram container (see recent work from Matthias)

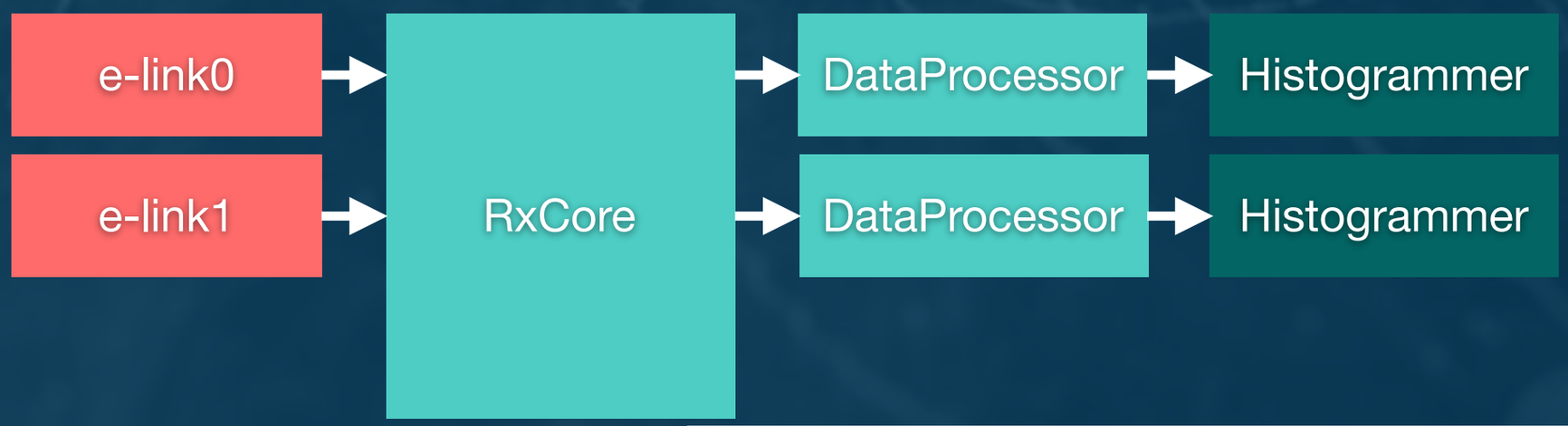
FELIX Integration

- FELIX is in many ways similar to YARR-PCIe as it is trying to stay agnostic and just shuffle data from and to the chip
- However primarily difference is that one does not interact directly with FELIX, but rather NetIO
  - NetIO is an IPC package and enables subscription to single data channels
  - NetIO has been successfully implemented as a hardware controller, however the interface is somewhat unoptimised towards it

NetIO optimisation



Currently assuming data comes through single interface, hence demultiplexing in DataProcessor. For NetIO we have to aggregate again because of this.



If data is available already demultiplexed, should just pass it on

Otherwise hw specific RxCore takes care of demux

Database

Other sources of data

QC Analysis

QC result

Multiple scans/configs

Viewer\ Editor

local DB

Config from last step

Production DB

Retriever

Configs

Scans

Viewer

local DB:

- Possibly gitDB based (prototype exists but needs some scrutinisation)
- Via git features can be used to sync configs over multiple machines or even to other institutes (aka remotes)
- All stored files are json based, plane file editing still possible

Production DB:

- Should only store good and interpreted QC data (result based on input from multiple scans)
- Can retrieve configs from last step (or before) to local DB
- QC Analysis can also take other source of data into account (e.g. pictures)

Further Outlook

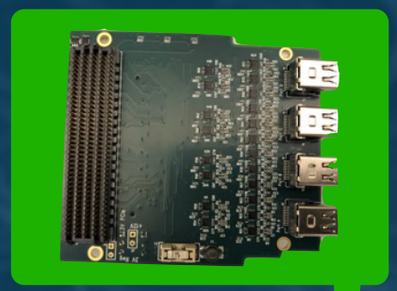
- Target supporting larger system tests:
  - Distributed processing
  - O(100) chip operation
  - FELIX
- Develop and document routines for QC
  - Interaction with database
  - Also interesting for Strips as we will run surface tests with FELIX and have to compare to previous QC
- Test and benchmark detector-level operation of the code
  - Pulling/Pushing configurations from DB
  - Crashing sub-processes
- Develop and implement SW ROD

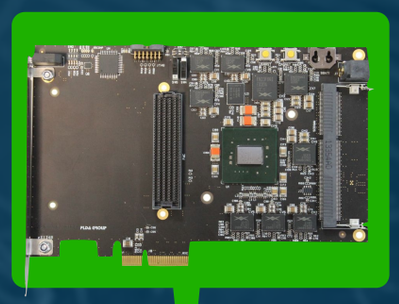
Backup

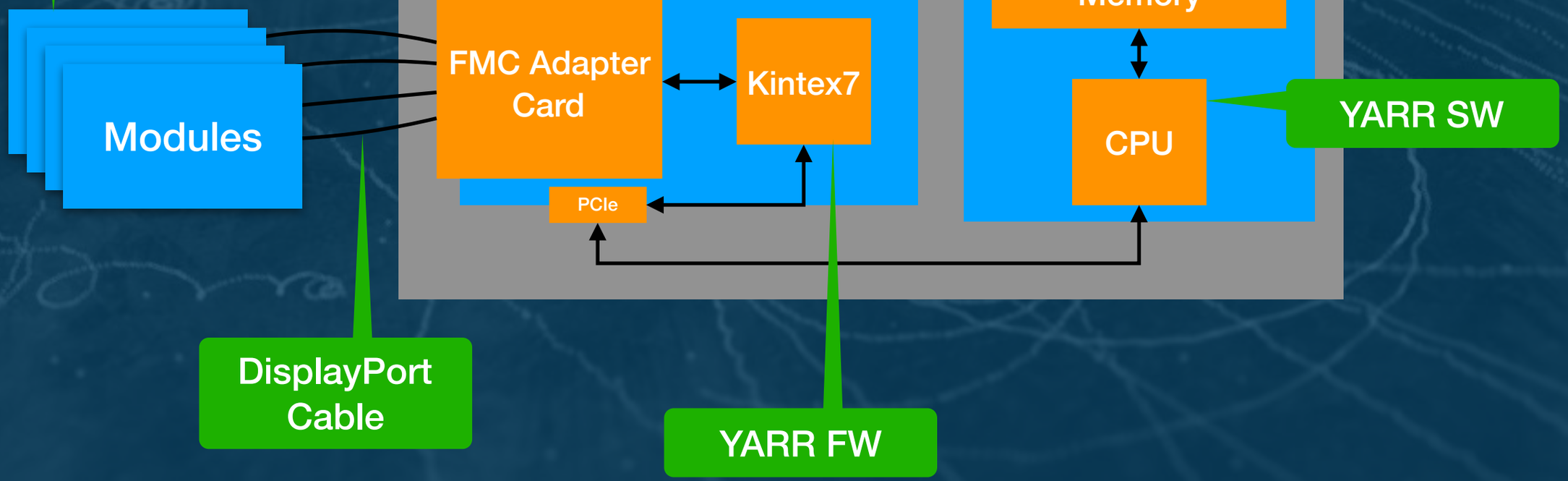
References

- Gitlab: https://gitlab.cern.ch/YARR/YARR
-

An example







HW: https://gitlab.cern.ch/YARR/YARR-FW

Loop Actions