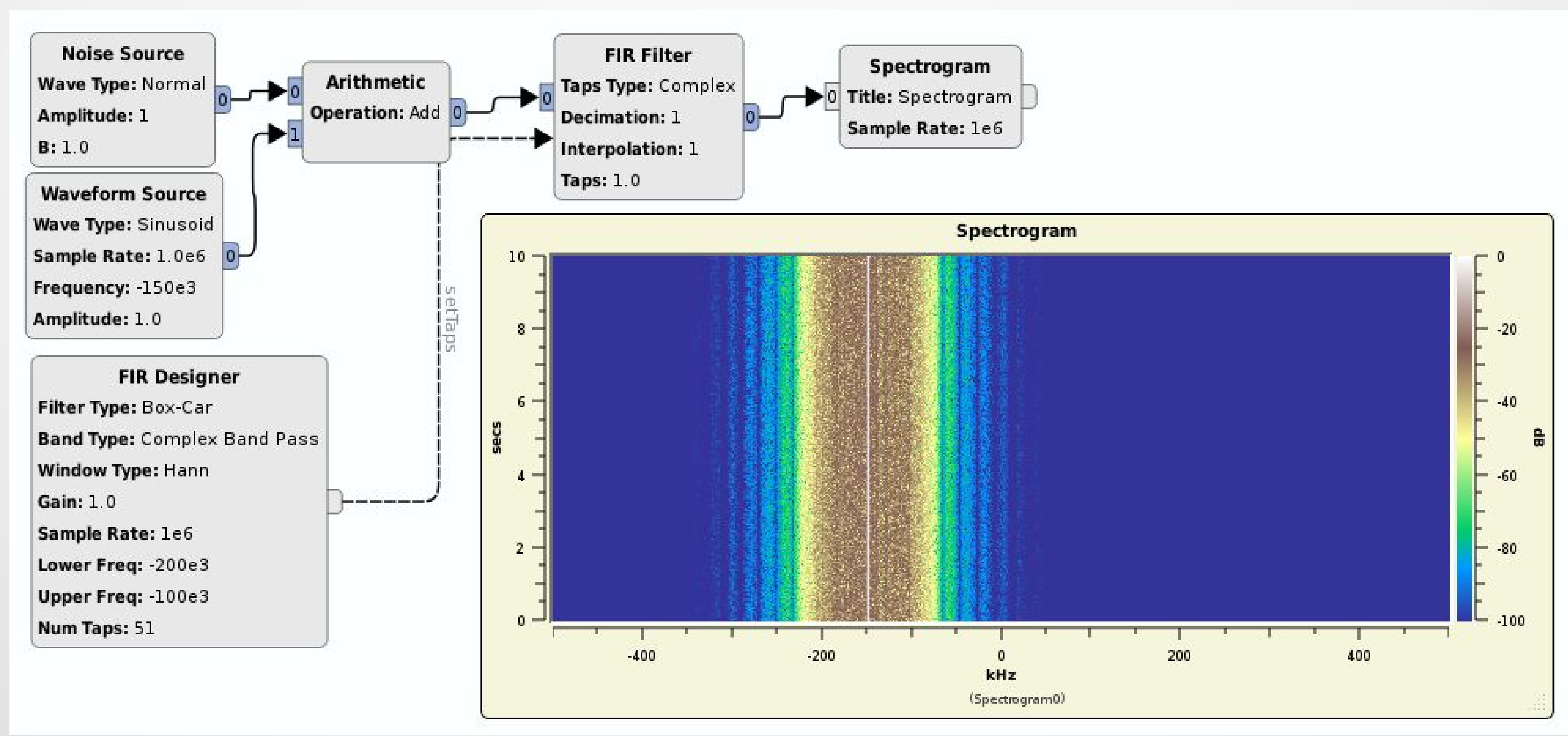
Pothos - http://pothosware.com/

Josh Blum presents Pothos – an open source computation framework, complete with graphical design interface, and companion project SoapySDR, for SDR hardware support.

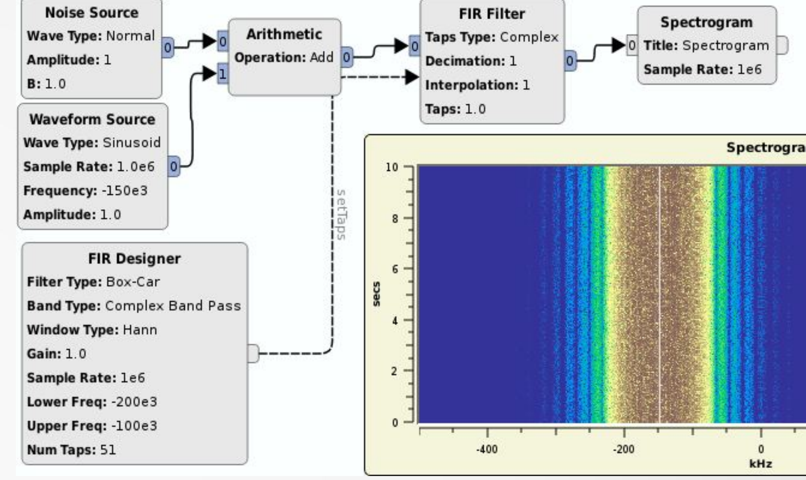


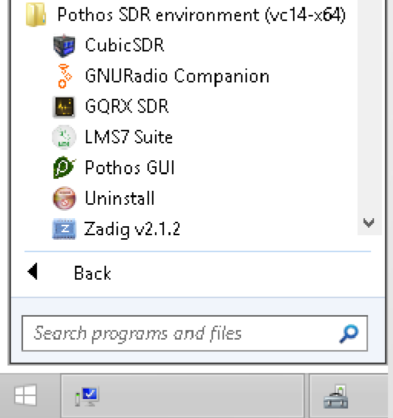
Josh Blum - Introduction

- Doing SDR stuff for a while now...
- GNU Radio Companion – JHU SRPL 2006
- GNU Radio things (VOLK, plotters, grextras, gras)
- USRP development (FPGA/FW, UHD, gr-uhd)
- Pothosware (Framework, PothosGUI, SoapySDR)
  - https://github.com/pothosware/
- Participant in LimeSDR campaign
- http://www.joshknows.com/projects
- https://github.com/guruofquality

Pothosware software stack

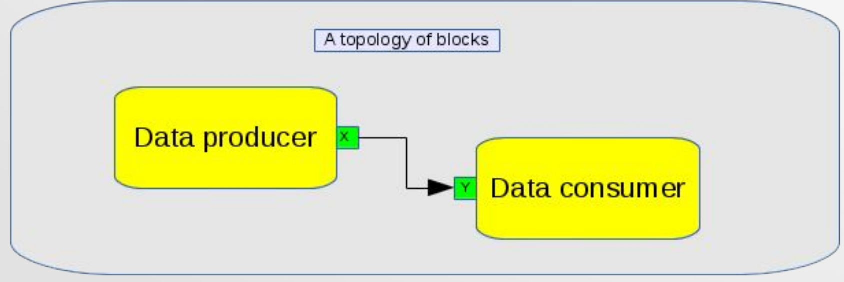
- Pothos framework - https://github.com/pothosware/pothos/wiki
  - Developing processing blocks
  - Connecting topologies of blocks
  - Comes with block and utilities
- Pothos GUI – https://github.com/pothosware/pothos-gui/wiki
  - Graphical topology design
  - Connections, signals, slots
  - Embedded graphical widgets
- SoapySDR - https://github.com/pothosware/SoapySDR/wiki
  - Library for SDR abstraction
  - C, C++, python languages
  - Based around plugins
  - pothos-sdr blocks
- PothosSDR windows installer - https://github.com/pothosware/PothosSDR/wiki
  - Pothos framework, GUI
  - SoapySDR + plugins
  - GNURadio and GRC
  - GQRX, CubicSDR





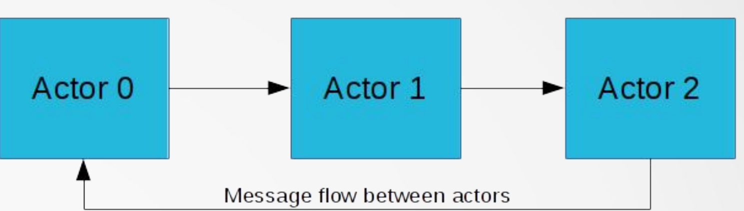
Pothos framework

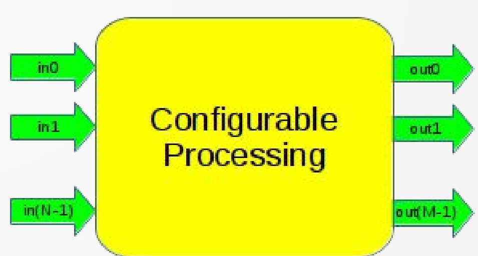
- Create interconnected topologies of re-usable, parameterized processing blocks to perform useful work.
- Permissive license for open source and commercial use
- Modular design based on loadable plugins, runtime extend-able, everything is a plugin: core data types, conversion functions, blocks...
- Writing blocks: C++11, compact style, minimal boiler plate, thread safe, available from the plugin tree, block factory access, GUI accessible
- Topologies can connect blocks across network/process boundaries
- Support toolkits: widgets, plotters, GUI designer, general purpose, communications, SDR, Audio, OpenCL, GNURadio
- Languages too: Python bindings, hopefully more



•Pothos framework – dive in!

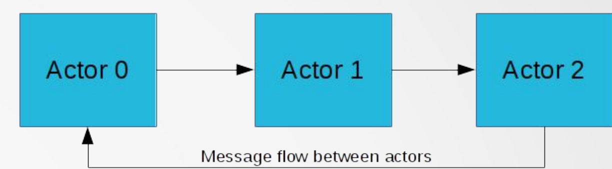
- Scheduler – how it works
  - Actors and message passing
  - Advanced threading options
  - Buffer management for streams
- The anatomy of a block
  - Blocks, ports, calls
  - Streams, labels, messages
  - Signals and slots
- Advanced stuff
  - Crossing processes/networks
  - Crossing language boundaries
- Future developments...

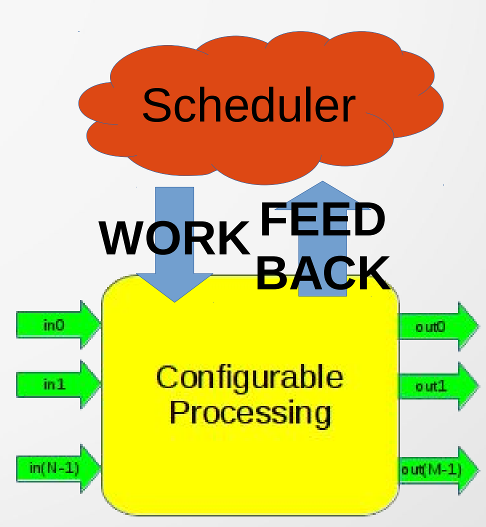




Pothos framework - actor model

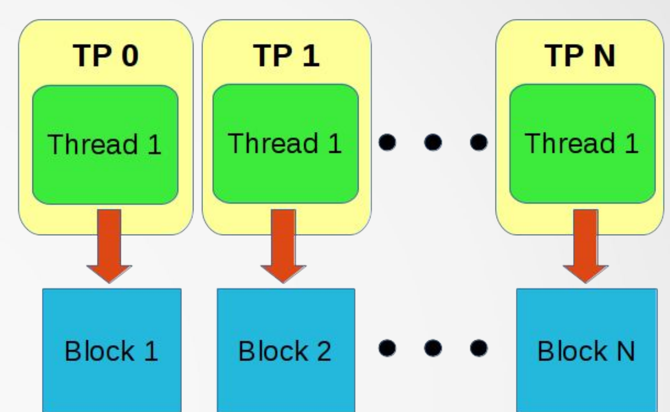
- https://github.com/pothosware/pothos/wiki/SchedulerExplained
- Actor model for concurrency
  - http://en.wikipedia.org/wiki/Actor_model
- Every block is an actor
  - Many functions (work, setters, allocators)
  - Block's state protected from concurrency
- When to work: Stimulus event + feedback
  - Activation/deactivation
  - Upstream/downstream resource
  - Function calls on the block
  - Other conditions...

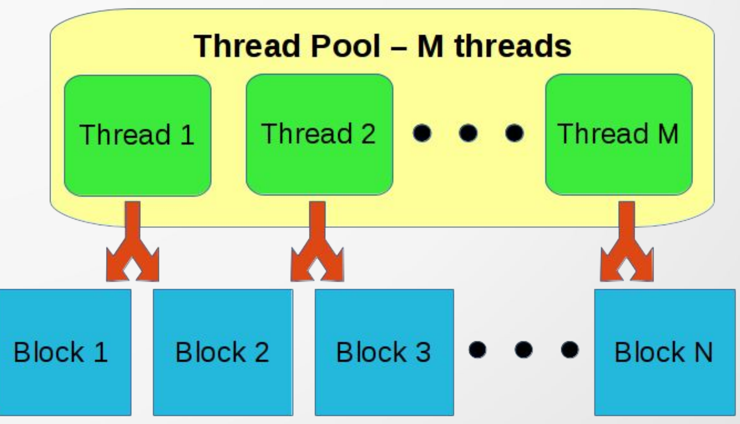


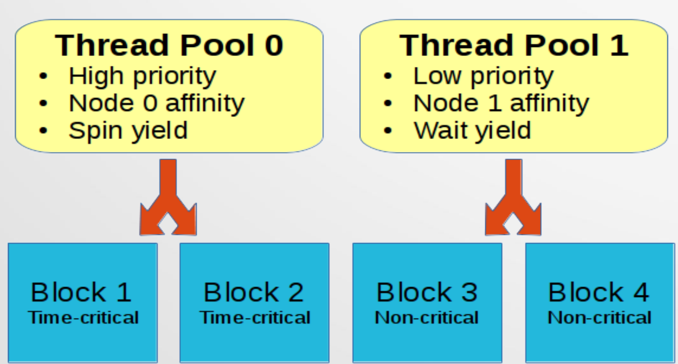


Pothos framework - threading

- Scheduler threads do the work.
- Default: each block gets its own thread with default priority
- Or custom thread pools
  - Custom affinity, priority
  - Waiting: block vs spin
  - Round robin through blocks

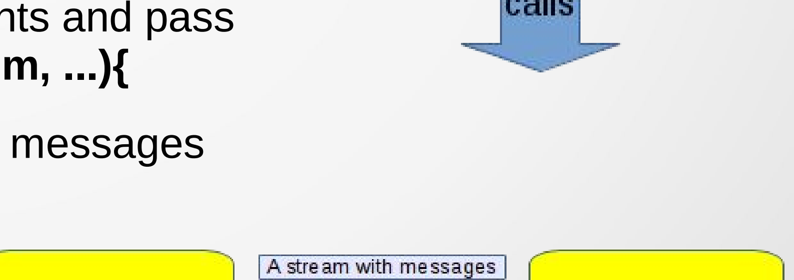


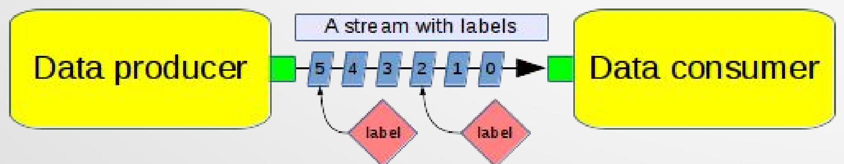




The anatomy of a block

- https://github.com/pothosware/pothos/wiki/BlocksCodingGuide
- Blocks have calls/methods, input ports, output ports
- Blocks have framework hooks (work, de/activate, buffer allocation)
- Ports can pass arbitrary messages, streams of buffers, and stream decorations – labels
- Signals/slots – a topologically friendly way to make function calls (think Qt)
  - Signals - output ports that emit arguments to downstream slots: **this->emitSignal("change", 1234, ...);**
  - Slots – input ports that accept upstream arguments and pass them to a block method: **void myHandler(int num, ...){**
  - Signals + slots are regular ports and interop with messages

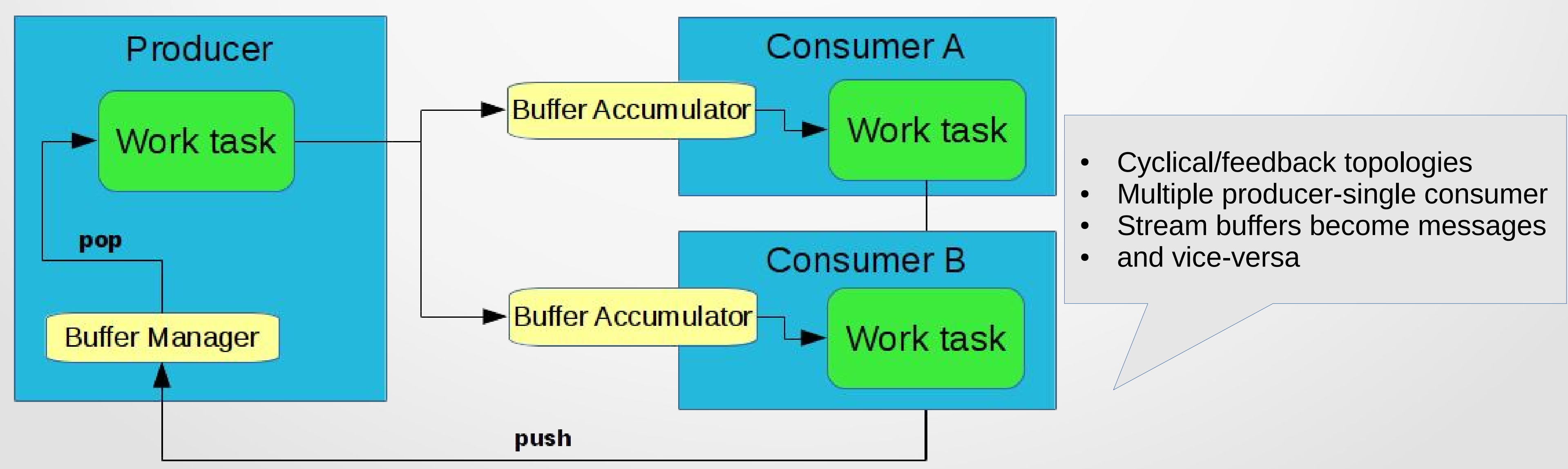




Pothos

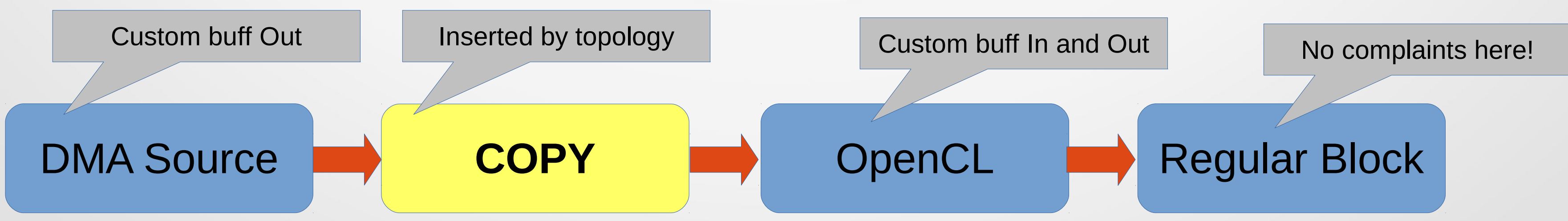
Pothos framework - streams

- Build streaming abstraction on top of Buffers and queues
- Flow backpressure is driven by limited resources
- Output ports get a buffer manager
  - buffer managers can be customized for size, circular, DMA
- Input port gets a buffer accumulator
  - Can also force a custom manager on upstream output port



Buffer managers & domains

- A custom output buffer manager replaces the output port's default buffer manager..
- An input buffer manager replaces **the upstream block's** buffer manager
  - What if theres two upstream blocks (multi producer)?
  - What if one of those upstream blocks has a custom output manager as well?
- Solution
  - Ports have configurable domains – this->setupInput(0, typeid(float), “openClDomainXYZ”);
  - Buffer manager hooks know this domain and can: abdicate, throw, enforce
  - The Topology tries its best! When everything fails → insert a **COPY** block



Writing a block – simple example

Class MyBlock

```
MyBlock::MyBlock(const int foo) {
    this->setupInput(0, typeid(float));
    this->setupOutput(“xyz”);
    this->registerCall(this, “setMode”, &MyBlock::setMode);
    this->registerCall(this, “getMode”, &MyBlock::getMode);
    this->registerSignal("valueChanged");

static Block *make(const int foo) {
    return new MyBlock(foo);
```

```
void MyBlock::activate(void) { //called when the topology is committed
    this->emitSignal("valueChanged", 0);
    _someInternalState = 0;
```

```
void MyBlock::work(void) {
    auto inPort = this->input(0);
    auto inBuff = inPort->buffer().as<const float *>();
    const size_t N = inPort->elements();
    //do something with buff
    inPort->consume(N);

    //state changed? Emit a new value to connected slots
    this->emitSignal("valueChanged", _currentValue);

     //buffer of interest? Forward it as a message
    auto outPort = this->output(“xyz”);
    outPort->postMessage(inPort->buffer());
```

```
void MyBlock::setMode(const std::string &mode) {
        _mode = mode;
```

```
std::string MyBlock::getMode(void) const {
        return _mode;
```

Register block into plugin tree

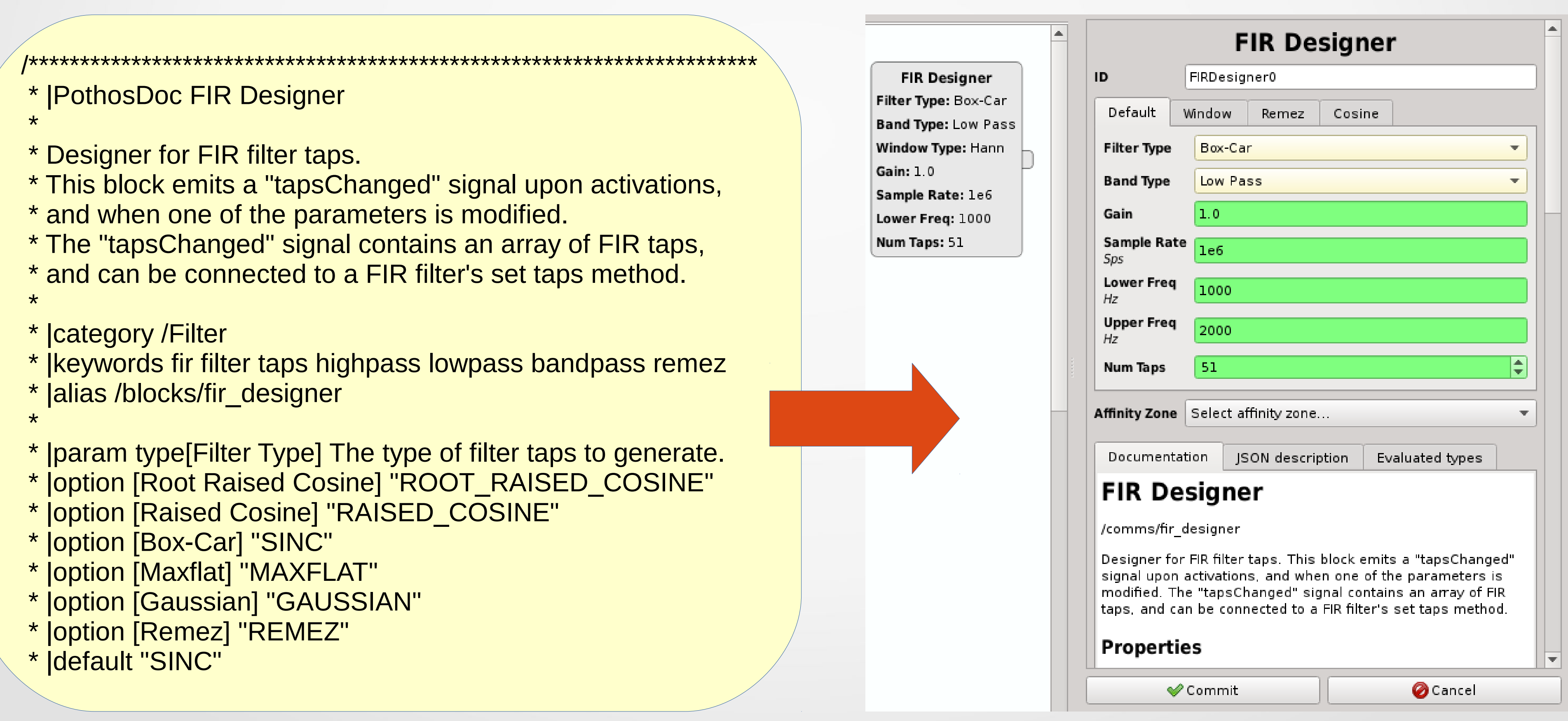
```
static Pothos::BlockRegistry registerMyBlock(
    "/myProject/my_block", &MyBlock::make);
```

Instantiate a block

```
auto myBlock = Pothos::BlockRegistry::make(
    "/myProject/my_block", 1234);
myBlock->callVoid(“setMode”, “MODE0”);
```

Writing a block – block description

- https://github.com/pothosware/pothos/wiki/BlockDescriptionMarkup
- Block descriptions are inline comments that the build parses into JSON and bundles with the module. It shows up in the GUI:



The Pothos data type system

- Goal: configure remote objects, pass arbitrary data type around, support language bindings, serialize for networking
- Pothos::Object – a container for arbitrary C++ objects (think boost::any)
  - With extensible support for conversions, hashing, sorting...
- Pothos::Proxy – an abstraction for an underlying object with generic ways to make calls, construct objects, access fields (think Python.h, jni.h)
  - Looks decent in C++ **myObj.call<ReturnType>("foo", 1234);**
  - Completely transparent in Python: **myObj.foo(1234)**
  - Implementations: registered C++ classes, remote access, Python, Java
- Used internally everywhere to support generic block factories, remote topologies, python blocks...

Custom C++ data type

```
class FluffyData {
    FluffyData(const int fluff);
    int getFluff(void) const;
    std::string wiggles;
```

FluffySource C++

```
void work(void) {
    auto outPort = this->output(0);

    //setup the data
    FluffyData data(1);
    data.wiggles = "Wiggle1";

    //produce the data as a message
    outPort->postMessage(data);
```

FluffySource Python

```
def work(self):
    outPort = self.output(0)

    #setup the data
    FluffyData = self._env.findProxy("FluffyData")
    data = FluffyData(3)
    data.wiggles = "Wiggle3"

    #produce the data as a message
    outPort.postMessage(data)
```

FluffySink Python

```
def work(self):
    inPort = self.input(0)

    #do we have an input message?
    if not inPort.hasMessage(): return

    #extract the data
    data = inPort.popMessage()

    print("FluffySinkPy: fluff=%d"%data.getFluff())
    print("FluffySinkPy: wiggles=%s"%data.wiggles)
```

FluffySink C++

```
void work(void) {
    auto inPort = this->input(0);

    //do we have an input message?
    if (not inPort->hasMessage()) return;

    //extract the data
    const auto msg = inPort->popMessage();
    const auto &data = msg.extract<FluffyData>();

    cout << "FluffySink: fluff=" << data.getFluff() << std::endl;
    cout << "FluffySink: wiggles=" << data.wiggles << std::endl;
```

- https://github.com/pothosware/pothos-demos/tree/master/custom_types

Custom Python data type

```
class SpikeyData:
    def __init__(self, spike=0):
        self._spike = spike
    def getSpike(self):
        return self._spike
```

SpikeySource C++

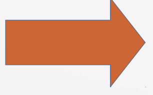
```
void work(void) {
    auto outPort = this->output(0);

    //setup the data
    auto DemoModule = _env->findProxy("DemoModule");
    auto SpikeyData = DemoModule.get("SpikeyData");
    auto data = SpikeyData(5);
    data.set("ouch", "Ouch5");

    //produce the data as a message
    outPort->postMessage(data);
```

SpikeySink Python

```
def work(self):
    inPort = self.input(0)

    #do we have an input message?
    if not inPort.hasMessage(): return

    #extract the data
    data = inPort.popMessage()

    print("SpikeySinkPy: spike=%d"%data.getSpike())
    print("SpikeySinkPy: ouch=%s"%data.ouch)
```

SpikeySource Python

```
def work(self):
    outPort = self.output(0)

    #setup the data
    data = SpikeyData(4)
    data.ouch = "Ouch4"

    #produce the data as a message
    outPort.postMessage(data)
```



SpikeySink C++

```
void work(void) {
    auto inPort = this->input(0);

    //do we have an input message?
    if (not inPort->hasMessage()) return;

    //extract the data
    const auto msg = inPort->popMessage();
    const auto &data = msg.extract<Pothos::Proxy>();

    cout << "SpikeySink: spike=" << data.call<int>("getSpike") << std::endl;
    cout << "SpikeySink: ouch=" << data.get<std::string>("ouch") << std::endl;
```

- https://github.com/pothosware/pothos-demos/tree/master/custom_types

Data types – remote access

On the server: PothosUtil --proxy-server=""

On the client: ./FluffyRemote tcp://remotehost

```
Pothos::RemoteClient client(uri);                       – //connect to the remote server
auto env = client.makeEnvironment("managed");

auto FluffyDataCls = env->findProxy("FluffyData");      – //create a FluffyData on the server
auto remoteData = FluffyDataCls(123);
remoteData.set("wiggles", "yippee");
std::cout << "FluffyRemote: fluff=" << remoteData.call<int>("getFluff") << std::endl;
std::cout << "FluffyRemote: wiggles=" << remoteData.get<std::string>("wiggles") << std::endl;

auto localData = remoteData.convert<FluffyData>();      – //get a FluffyData locally
std::cout << "FluffyLocal: fluff=" << localData.getFluff() << std::endl;
std::cout << "FluffyLocal: wiggles=" << localData.wiggles << std::endl;

auto remoteData2 = env->makeProxy(localData);           –  //copy into a second remote object
remoteData2.callVoid("setFluff", 987);
std::cout << "FluffyRemote2: fluff=" << remoteData2.call<int>("getFluff") << std::endl;
std::cout << "FluffyRemote2: wiggles=" << remoteData2.get<std::string>("wiggles") << std::endl;

auto localEnv = Pothos::ProxyEnvironment::make("managed");   – //get a FluffyData locally as an object
auto localData2 = localEnv->makeProxy(remoteData2.toObject());
std::cout << "FluffyLocal2: fluff=" << localData2.call<int>("getFluff") << std::endl;
std::cout << "FluffyLocal2: wiggles=" << localData2.get<std::string>("wiggles") << std::endl;
```

Pothos GUI – Live Demo

- https://github.com/pothosware/pothos-gui/wiki/Tutorial
- GUI to match features in the framework
- Instantiation and connection of blocks
- Graphical widgets, connecting signals + slots
- Running the topology, live reconfiguration
- Graph pages, connection breakers, zooming...
- Affinity zones, remote stuff, view rendered topology

SoapySDR – hardware abstraction library

- https://github.com/pothosware/SoapySDR/wiki
- One API, many devices - Python, C, and C++ API
- Plugin based SDR abstraction layer
  - Most devices: RTL, BladeRF, HackRF, Play, Airspy...
  - SoapyRemote – transparent remote device support: https://github.com/pothosware/SoapyRemote/wiki
  - SoapyMultiSDR – many devices one device handle
  - Also useful HAL for non-SDR devices
- Platforms -
  - GNU Radio (gr-osmosdr support blocks)
  - Pothos SDR source and sink blocks
  - CubicSDR - http://cubicsdr.com/
  - Rx Tools - https://github.com/rxseger/rx_tools

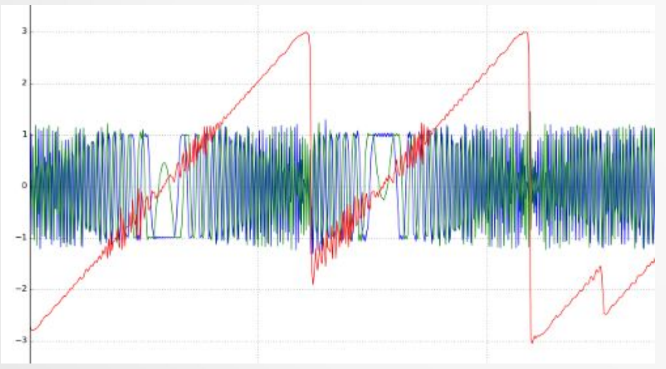
SDR apps + platforms

SoapySDR Application API

Support modules
RTL, BladeRF...
Remote, Multi...

SoapySDR – python bindings

- https://github.com/pothosware/SoapySDR/wiki/PythonSupport
- Get started with SDR using SoapySDR+Python
  - Numpy – vectorized math
  - Scipy – re-sampling, filters
  - Matplotlib – plotting library
- PC can handle ~5 Msps



Read From RTLSDR with Python

```
import SoapySDR
from SoapySDR import * #SOAPY_SDR_ constants

#setup device
sdr = SoapySDR.Device('driver=rtlsdr')
sdr.setFrequency(SOAPY_SDR_RX, 0, 868.1e6)
sdr.setSampleRate(SOAPY_SDR_RX, 0, 2*1024e3)

#setup stream
rxStream = sdr.setupStream(SOAPY_SDR_RX, SOAPY_SDR_CF32)
sdr.activateStream(rxStream) #start streaming
buff = np.array([0]*1024, np.complex64)
sr = sdr.readStream(rxStream, [buff], len(buff))
sdr.deactivateStream(rxStream) #stop streaming
sdr.closeStream(rxStream)
```

Demo: Capture and plot LoRa to debug decoder with RN2483 and RTLSDR - https://github.com/myriadrf/LoRa-SDR (RN2483Capture.py)

Future features, improvements

- Just in time (JIT) block registry – No one compiles any more
  - Python blocks, JSON topologies, simple C++ blocks
- More blocks/core toolkits:
  - Add to pothos-comms, wrap liquidDSP
  - Graphical filter design widgets from Spuce - https://github.com/audiofilter/spuce/
- UI improvements
  - GUI evaluator improvements (better detection and recovery)
  - GUI – dynamic block properties (overlays, almost working)

Thanks!

- https://github.com/pothosware/pothos/wiki/Support
  - https://groups.google.com/d/forum/pothos-users
  - https://twitter.com/pothosware
  - #pothos on freenode

- Questions?