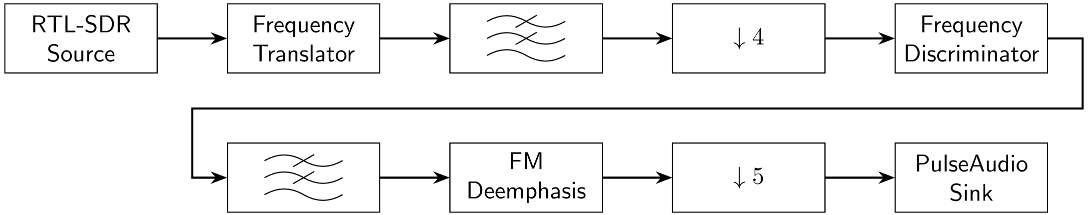
ZigRadio

A lightweight, ergonomic flow graph signal processing framework for SDR

Vanya Sergeev (vsergeev)

FOSDEM 2026

Introduction - FM Broadcast Receiver




```
const std = @import("std");

const radio = @import("radio");

pub fn main() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};

    const frequency: f64 = 91.1e6; // 91.1 MHz

    var source = radio.blocks.RtlSdrSource.init(frequency - 250e3, 960000, .{});
    var if_translator = radio.blocks.FrequencyTranslatorBlock.init(-250e3);
    var if_filter = radio.blocks.LowpassFilterBlock(std.math.Complex(f32), 128).init(200e3, .{});
    var if_downsampler = radio.blocks.DownsamplerBlock(std.math.Complex(f32)).init(4);
    var fm_demod = radio.blocks.FrequencyDiscriminatorBlock.init(75e3);
    var af_filter = radio.blocks.LowpassFilterBlock(f32, 128).init(15e3, .{});
    var af_deemphasis = radio.blocks.FMDeemphasisFilterBlock.init(75e-6);
    var af_downsampler = radio.blocks.DownsamplerBlock(f32).init(5);
    var sink = radio.blocks.PulseAudioSink(1).init();

    var top = radio.Flowgraph.init(gpa.allocator(), .{ .debug = true });
    defer top.deinit();
    try top.connect(&source.block, &if_translator.block);
    try top.connect(&if_translator.block, &if_filter.block);
    try top.connect(&if_filter.block, &if_downsampler.block);
    try top.connect(&if_downsampler.block, &fm_demod.block);
    try top.connect(&fm_demod.block, &af_filter.block);
    try top.connect(&af_filter.block, &af_deemphasis.block);
    try top.connect(&af_deemphasis.block, &af_downsampler.block);
    try top.connect(&af_downsampler.block, &sink.block);

    _ = try top.run();
}
```

Introduction

Introduction

- Build standalone radio receivers and transmitters, signal processing experiments

Introduction

- Build standalone radio receivers and transmitters, signal processing experiments
- Integrate as a signal processing engine in a host application

Introduction

- Build standalone radio receivers and transmitters, signal processing experiments
- Integrate as a signal processing engine in a host application
- Use existing blocks packaged with the framework, or create your own

Introduction

- Build standalone radio receivers and transmitters, signal processing experiments
- Integrate as a signal processing engine in a host application
- Use existing blocks packaged with the framework, or create your own
- External libraries for acceleration (VOLK, liquid-dsp) and I/O (librtlsdr, libairspy, etc.)

Introduction

- Build standalone radio receivers and transmitters, signal processing experiments
- Integrate as a signal processing engine in a host application
- Use existing blocks packaged with the framework, or create your own
- External libraries for acceleration (VOLK, liquid-dsp) and I/O (librtlsdr, libairspy, etc.)
- Written in the Zig language

Introduction

- Build standalone radio receivers and transmitters, signal processing experiments
- Integrate as a signal processing engine in a host application
- Use existing blocks packaged with the framework, or create your own
- External libraries for acceleration (VOLK, liquid-dsp) and I/O (librtlsdr, libairspy, etc.)
- Written in the Zig language
- Open source, MIT licensed

Introduction

- Build standalone radio receivers and transmitters, signal processing experiments
- Integrate as a signal processing engine in a host application
- Use existing blocks packaged with the framework, or create your own
- External libraries for acceleration (VOLK, liquid-dsp) and I/O (librtlsdr, libairspy, etc.)
- Written in the Zig language
- Open source, MIT licensed
- Spiritual successor to LuaRadio

Goals of ZigRadio

Goals of ZigRadio

1. High performance, but easy to use

Goals of ZigRadio

1. High performance, but easy to use
2. Zero dependencies* → easy to build, easy to deploy, easy to integrate

Goals of ZigRadio

1. High performance, but easy to use
2. Zero dependencies* → easy to build, easy to deploy, easy to integrate
   - Compiles into a single static executable

Goals of ZigRadio

1. High performance, but easy to use
2. Zero dependencies* → easy to build, easy to deploy, easy to integrate
   - Compiles into a single static executable
   - * I/O and acceleration libraries are dynamically loaded at runtime

Goals of ZigRadio

1. High performance, but easy to use
2. Zero dependencies* → easy to build, easy to deploy, easy to integrate
   - Compiles into a single static executable
   - * I/O and acceleration libraries are dynamically loaded at runtime
3. Rapid prototyping, especially for embedded targets (RPi 4/5)

Goals of ZigRadio

1. High performance, but easy to use
2. Zero dependencies* → easy to build, easy to deploy, easy to integrate
   - Compiles into a single static executable
   - * I/O and acceleration libraries are dynamically loaded at runtime
3. Rapid prototyping, especially for embedded targets (RPi 4/5)
   - Cross-compile from anywhere (e.g. x86-64) to anything (e.g. aarch64)

Goals of ZigRadio

1. High performance, but easy to use
2. Zero dependencies* → easy to build, easy to deploy, easy to integrate
   - Compiles into a single static executable
   - * I/O and acceleration libraries are dynamically loaded at runtime
3. Rapid prototyping, especially for embedded targets (RPi 4/5)
   - Cross-compile from anywhere (e.g. x86-64) to anything (e.g. aarch64)
   - Ideally as fast as LuaRadio / LuaJIT one day

LuaRadio

LuaRadio

v0.11.0
GitHub
Mailing List

v@sergeev.io

LuaRadio is a lightweight, embeddable flow graph signal processing framework for software-defined radio. It provides a suite of source, sink, and processing blocks, with a simple API for defining flow graphs, running flow graphs, creating blocks, and creating data types. LuaRadio is built on LuaJIT, has a small binary footprint of under 750 KB (including LuaJIT), has no external hard dependencies, and is MIT licensed.

LuaRadio can be used to rapidly prototype software radios, modulation/demodulation utilities, and signal processing experiments. It can also be embedded into existing radio applications to serve as a user scriptable engine for signal processing.

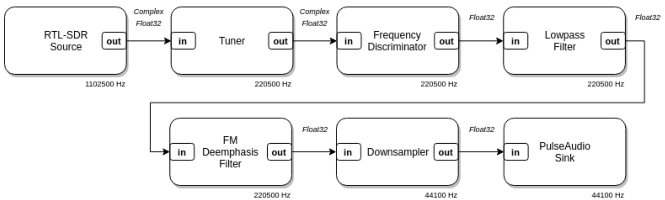
LuaRadio blocks are written in pure Lua, but can use LuaJIT's FFI to wrap external libraries, like VOLK, liquid-dsp, and others, for computational acceleration, sophisticated processing, and interfacing with SDR hardware.

Use GNU Radio? See how LuaRadio compares to GNU Radio.

See the LuaRadio mailing list for general discussion.

Example

Wideband FM Broadcast Radio Receiver




```lua
local radio = require('radio')

radio.CompositeBlock():connect(
    radio.RtlSdrSource(88.5e6 - 250e3, 1102500), -- RTL-SDR source, offset-tuned to 88.5 MHz - 250 kHz
    radio.TunerBlock(-250e3, 200e3, 5),          -- Translate -250 kHz, filter 200 kHz, decimate 5
    radio.FrequencyDiscriminatorBlock(1.25),     -- Frequency demodulate with 1.25 modulation index
    radio.LowpassFilterBlock(128, 15e3),         -- Low-pass filter 15 kHz for L+R audio
    radio.FMDeemphasisFilterBlock(75e-6),        -- FM de-emphasis filter with 75 uS time constant
    radio.DownsamplerBlock(5),                   -- Downsample by 5
    radio.PulseAudioSink(1)                      -- Play to system audio with PulseAudio
):run()
```

https://luaradio.io

LuaRadio vs ZigRadio

- What’s the same?
  - Goals
    - Performance + ease of use
    - Minimal dependencies
    - Rapid prototyping
  - Look and feel (API)
  - Application integration (but ZigRadio is easier)

LuaRadio vs ZigRadio

- What’s different?
  - Compiled (Zig) vs interpreted (LuaJIT)
  - Multi-threaded (with shared ring buffers) vs Multi-process (with UNIX sockets)
  - Many more targets supported
  - Asynchronous block control
  - Long-term viability

Zig Language

Zig Language

- Modern, low-level systems programming language – a "better C"

Zig Language

- Modern, low-level systems programming language – a "better C"

```zig
const std = @import("std");

pub fn main() void {
    const x = 2 + 2;
    std.debug.print("2 + 2 = {d}\n", .{x});
}
```

```
$ zig run test.zig
2 + 2 = 4
$
```

Zig Language

- Modern, low-level systems programming language – a "better C"

```zig
const std = @import("std");

pub fn main() void {
    const x = 2 + 2;
    std.debug.print("2 + 2 = {d}\n", .{x});
}
```

```
$ zig run test.zig
2 + 2 = 4
$
```

- Simple: "no hidden control flow", "no hidden memory allocations"

Zig Language

- Modern, low-level systems programming language – a "better C"

```zig
const std = @import("std");

pub fn main() void {
    const x = 2 + 2;
    std.debug.print("2 + 2 = {d}\n", .{x});
}
```

```
$ zig run test.zig
2 + 2 = 4
$
```

- Simple: "no hidden control flow", "no hidden memory allocations"
  - No operator overloading, no implicit destructors, no implicit inheritance, no implicit heap allocations, etc.

Zig Language

- Modern, low-level systems programming language – a "better C"

```zig
const std = @import("std");

pub fn main() void {
    const x = 2 + 2;
    std.debug.print("2 + 2 = {d}\n", .{x});
}
```

```
$ zig run test.zig
2 + 2 = 4
$
```

- Simple: "no hidden control flow", "no hidden memory allocations"
  - No operator overloading, no implicit destructors, no implicit inheritance, no implicit heap allocations, etc.
  - Compile-time metaprogramming in the same language

Zig Language

- Modern, low-level systems programming language – a "better C"

```zig
const std = @import("std");

pub fn main() void {
    const x = 2 + 2;
    std.debug.print("2 + 2 = {d}\n", .{x});
}
```

```
$ zig run test.zig
2 + 2 = 4
$
```

- Simple: "no hidden control flow", "no hidden memory allocations"
  - No operator overloading, no implicit destructors, no implicit inheritance, no implicit heap allocations, etc.
  - Compile-time metaprogramming in the same language
- Excellent target and platform support

Zig Language

- Modern, low-level systems programming language – a "better C"

```zig
const std = @import("std");

pub fn main() void {
    const x = 2 + 2;
    std.debug.print("2 + 2 = {d}\n", .{x});
}
```

```
$ zig run test.zig
2 + 2 = 4
$
```

- Simple: "no hidden control flow", "no hidden memory allocations"
  - No operator overloading, no implicit destructors, no implicit inheritance, no implicit heap allocations, etc.
  - Compile-time metaprogramming in the same language
- Excellent target and platform support
- Fast compilation

Zig Language - Sample

```zig
const x: u32 = 123; // Statically typed

var y: [3]i32 = .{ 1000, 2000, 3000 }; // Fixed size arrays

const p: *i32 = &y[1]; // Pointers

const z: []i32 = &y; // Slices

const str: []const u8 = "abc"; // Strings

var w: ?i32 = y[1]; // Optionals, can be null
```

```zig
// Functions can return errors
fn process(x: u32, y: u32) !u32 {
    if (x > 5 or y > 5) return error.OutOfBounds;
    return x + y;
}

// u32 or error
const result: !u32 = process(1, 2);
// u32 or catch and return error
const result: u32 = process(1, 10) catch |err| return err;
// u32 or catch and return error (same as above)
const result: u32 = try process(1, 10);
```

```zig
// For loop
var x: []f32;
for (x) |e| { ... }
// For loop with index
for (x, 0..) |e, i| { ... }
// While loop
while (y < 5) { ... }
```

```zig
// Structures
const Point = struct {
    x: f32,
    y: f32,

    // Functions with self
    pub fn add(self: *Point, other: Point) Point {
        return .{ .x = self.x + other.x,
                  .y = self.x + other.y };
    }
};

var p1 = Point{ .x = 2, .y = 3 };
var p2 = Point{ .x = 1, .y = 1 };
var p3 = p1.add(p2);
std.debug.print("{any}\n", .{p3}); // .{ .x = 3, .y = 4 }
```

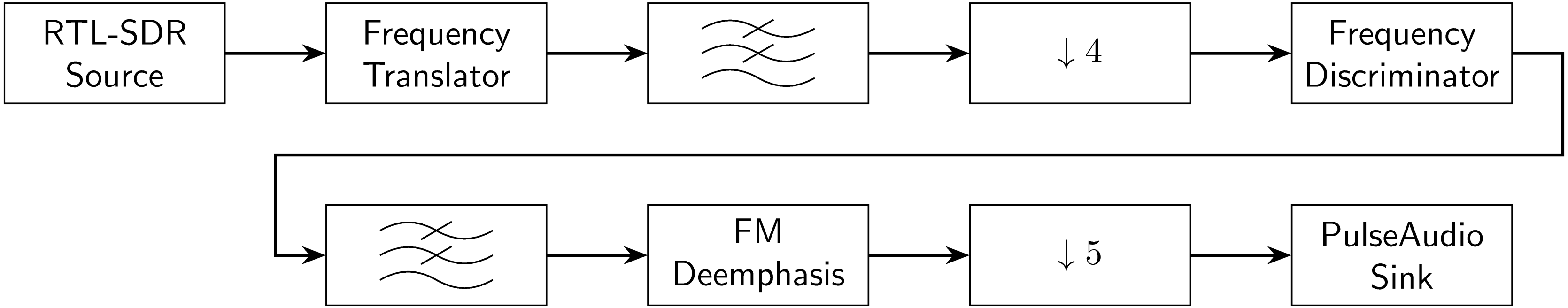
Zig Language - Comptime

```zig
// Compute Fibonacci number
fn fib(n: usize) usize {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}

pub fn main() void {
    // Runs at runtime
    std.debug.print("fib(10): {d}\n", .{fib(10)});
    // Runs at compile time
    std.debug.print("fib(10): {d}\n", .{comptime fib(10)});
}
```

```zig
// Types are available at comptime
const T: type = i32;

// Conditional compilation
if (T == f32) { ... }
if (T == i32) { ... }
if (comptime fib(7) < 10) { ... }

// Loop unrolling
inline for (0..comptime fib(7)) |e| { ... }
// will unroll 0, 1, 2, 3, ..., 12
```

```zig
// Generic structures can be parametrized at comptime
pub fn MovingAverage(comptime T: type, N: usize) type {
    return struct {
        const Self = @This(); // Type of anonymous struct
        history: [N]T,

        pub fn push(self: *Self, value: T) void {
            for (self.history[1..], 0..) |e, i| {
                self.history[i] = e;
            }
            self.history[N - 1] = value;
        }

        pub fn average(self: *const Self) T {
            var sum: T = 0;
            for (self.history) |e| sum += e;
            return sum / N;
        }
    };
}

var sma = MovingAverage(f32, 3){ .history = .{ 0, 0, 0 } };
sma.push(2.1); sma.push(3.5); sma.push(7.0);
std.debug.print("{d}\n", .{sma.average()}); // 4.2000003
```

Creating Flow Graphs - FM Broadcast Receiver



Creating Flow Graphs - FM Broadcast Receiver

- Instantiate blocks

```
const std = @import("[std](std)");

const radio = @import("radio");

pub fn main() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};

    const frequency: f64 = 91.1e6; // 91.1 MHz

    var source = radio.blocks.RtlSdrSource.init(frequency - 250e3, 960000, .{});
    var if_translator = radio.blocks.FrequencyTranslatorBlock.init(-250e3);
    var if_filter = radio.blocks.LowpassFilterBlock(std.math.Complex(f32), 128).init(200e3, .{});
    var if_downsampler = radio.blocks.DownsamplerBlock(std.math.Complex(f32)).init(4);
    var fm_demod = radio.blocks.FrequencyDiscriminatorBlock.init(75e3);
    var af_filter = radio.blocks.LowpassFilterBlock(f32, 128).init(15e3, .{});
    var af_deemphasis = radio.blocks.FMDeemphasisFilterBlock.init(75e-6);
    var af_downsampler = radio.blocks.DownsamplerBlock(f32).init(5);
    var sink = radio.blocks.PulseAudioSink(1).init();

    var top = radio.Flowgraph.init(gpa.allocator(), .{ .debug = true });
    defer top.deinit();
```

Creating Flow Graphs - FM Broadcast Receiver

- Connect blocks in a `Flowgraph`

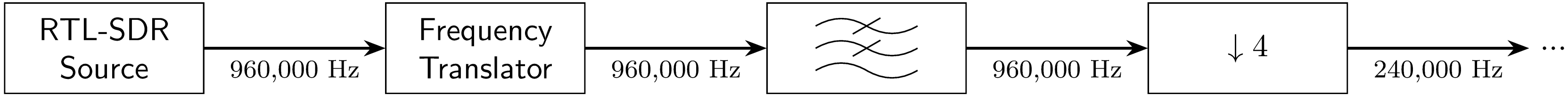
```
    var if_downsampler = radio.blocks.DownsamplerBlock(std.math.Complex(f32)).init(4);
    var fm_demod = radio.blocks.FrequencyDiscriminatorBlock.init(75e3);
    var af_filter = radio.blocks.LowpassFilterBlock(f32, 128).init(15e3, .{});
    var af_deemphasis = radio.blocks.FMDeemphasisFilterBlock.init(75e-6);
    var af_downsampler = radio.blocks.DownsamplerBlock(f32).init(5);
    var sink = radio.blocks.PulseAudioSink(1).init();

    var top = radio.Flowgraph.init(gpa.allocator(), .{ .debug = true });
    defer top.deinit();
    try top.connect(&source.block, &if_translator.block);
    try top.connect(&if_translator.block, &if_filter.block);
    try top.connect(&if_filter.block, &if_downsampler.block);
    try top.connect(&if_downsampler.block, &fm_demod.block);
    try top.connect(&fm_demod.block, &af_filter.block);
    try top.connect(&af_filter.block, &af_deemphasis.block);
    try top.connect(&af_deemphasis.block, &af_downsampler.block);
    try top.connect(&af_downsampler.block, &sink.block);

    _ = try top.run();
}
```

Creating Flow Graphs - FM Broadcast Receiver

- Run the `Flowgraph`

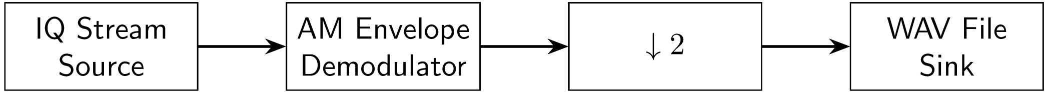
```
    var if_downsampler = radio.blocks.DownsamplerBlock(std.math.Complex(f32)).init(4);
    var fm_demod = radio.blocks.FrequencyDiscriminatorBlock.init(75e3);
    var af_filter = radio.blocks.LowpassFilterBlock(f32, 128).init(15e3, .{});
    var af_deemphasis = radio.blocks.FMDeemphasisFilterBlock.init(75e-6);
    var af_downsampler = radio.blocks.DownsamplerBlock(f32).init(5);
    var sink = radio.blocks.PulseAudioSink(1).init();

    var top = radio.Flowgraph.init(gpa.allocator(), .{ .debug = true });
    defer top.deinit();
    try top.connect(&source.block, &if_translator.block);
    try top.connect(&if_translator.block, &if_filter.block);
    try top.connect(&if_filter.block, &if_downsampler.block);
    try top.connect(&if_downsampler.block, &fm_demod.block);
    try top.connect(&fm_demod.block, &af_filter.block);
    try top.connect(&af_filter.block, &af_deemphasis.block);
    try top.connect(&af_deemphasis.block, &af_downsampler.block);
    try top.connect(&af_downsampler.block, &sink.block);

    _ = try top.run();
}
```

ZigRadio Features - Sample Rate Propagation




```zig
// Source sets downstream sample rate, all subsequent frequency offsets and bandwidths specified in Hz
var source = radio.blocks.RtlSdrSource.init(91.1e6 - 250e3, 960000, .{});
var translator = radio.blocks.FrequencyTranslatorBlock.init(-250e3);
var filter = radio.blocks.LowpassFilterBlock(std.math.Complex(f32), 64).init(200e3 / 2, .{});
var downsampler = radio.blocks.DownsamplerBlock(std.math.Complex(f32)).init(4);
...
```

ZigRadio Features - Finite Stream Processing




```
var source = radio.blocks.IQStreamSource.init(&input_reader.interface, .s16le, 96e3, .{});
var demod = radio.blocks.AMEnvelopeDemodulator(.{ .bandwidth = 5e3 });
var downsampler = radio.blocks.DownsamplerBlock(f32).init(2);
var sink = radio.blocks.WAVFileSink(1).init(&output_file, .{});
...
// Will run to completion
_ = try top.run();
```

ZigRadio Features - Asynchronous Block Control

- Blocks can provide asynchronous APIs to change parameters, do I/O, etc.

ZigRadio Features - Asynchronous Block Control

- Blocks can provide asynchronous APIs to change parameters, do I/O, etc.

- Tune to a different frequency:

```
var source = radio.blocks.RtlSdrSource.init(91.1e6, 960000, .{});
try flowgraph.call(&source.block, radio.blocks.RtlSdrSource.setFrequency, .{105.3e6});
```

ZigRadio Features - Asynchronous Block Control

- Blocks can provide asynchronous APIs to change parameters, do I/O, etc.
- Tune to a different frequency:

```
var source = radio.blocks.RtlSdrSource.init(91.1e6, 960000, .{});
try flowgraph.call(&source.block, radio.blocks.RtlSdrSource.setFrequency, .{105.3e6});
```

- Change filter cutoff:

```
var filter = radio.blocks.LowpassFilterBlock(f32, 64).init(10e3, .{});
try flowgraph.call(&filter.block, radio.blocks.LowpassFilterBlock(f32, 64).setCutoff, .{ 5e3 });
```

ZigRadio Features - Asynchronous Block Control

- Blocks can provide asynchronous APIs to change parameters, do I/O, etc.
- Tune to a different frequency:

```
var source = radio.blocks.RtlSdrSource.init(91.1e6, 960000, .{});
try flowgraph.call(&source.block, radio.blocks.RtlSdrSource.setFrequency, .{105.3e6});
```

- Change filter cutoff:

```
var filter = radio.blocks.LowpassFilterBlock(f32, 64).init(10e3, .{});
try flowgraph.call(&filter.block, radio.blocks.LowpassFilterBlock(f32, 64).setCutoff, .{ 5e3 });
```

- Change AGC preset:

```
var agc = radio.blocks.AGCBlock(f32)).init(.{ .preset = .Fast }, .{});
try flowgraph.call(&agc.block, radio.blocks.AGCBlock(f32).setMode, .{ .preset = .Slow });
```

Creating Blocks

- A block is a `struct` with a `block: Block` field and a `process(...)` function

```
const radio = @import("radio");

pub const MultiplyBlock = struct {
    block: radio.Block,

    pub fn init() MultiplyBlock {
        return .{ .block = radio.Block.init(@This()) };
    }

    pub fn process(_: *MultiplyBlock, x: []const f32, y: []const f32, z: []f32) !radio.ProcessResult {
        for (x, 0..) |_, i| {
            z[i] = x[i] * y[i];
        }

        return radio.ProcessResult.init(&[2]usize{ x.len, x.len }, &[1]usize{x.len});
    }
};
```

Creating Blocks - Parametric

- Blocks can be made parametric by accepting a comptime type and returning a parameterized struct

```
const radio = @import("radio");

pub fn MultiplyBlock(comptime T: type) type {
    return struct {
        const Self = @This();
        block: radio.Block,

        pub fn init() Self {
            return .{ .block = radio.Block.init(@This()) };
        }

        pub fn process(_: *Self, x: []const T, y: []const T, z: []T) !radio.ProcessResult {
            for (x, 0..) |_, i| {
                z[i] = x[i] * y[i];
            }

            return radio.ProcessResult.init(&[2]usize{ x.len, x.len }, &[1]usize{x.len});
        }
    };
}
```

Creating Blocks - Data Types

- Blocks can use any Zig type
- Common types include:
  - `std.math.Complex(f32)` for complex-valued samples
  - `f32` for real-valued samples
  - `u8` for byte samples
  - `u1` for bit samples

Creating Blocks - Hooks

Creating Blocks - Hooks

- Blocks can provide hooks that are called during flow graph lifecycle

Creating Blocks - Hooks

- Blocks can provide hooks that are called during flow graph lifecycle
- `initialize(self: *Self, allocator: std.mem.Allocator) !void`
  - Memory allocation, I/O initialization, sample rate dependent initialization

Creating Blocks - Hooks

- Blocks can provide hooks that are called during flow graph lifecycle
- `initialize(self: *Self, allocator: std.mem.Allocator) !void`
  - Memory allocation, I/O initialization, sample rate dependent initialization
- `deinitialize(self: *Self, allocator: std.mem.Allocator) !void`
  - Memory deallocation, I/O deinitialization

Creating Blocks - Hooks

- Blocks can provide hooks that are called during flow graph lifecycle
- `initialize(self: *Self, allocator: std.mem.Allocator) !void`
  - Memory allocation, I/O initialization, sample rate dependent initialization
- `deinitialize(self: *Self, allocator: std.mem.Allocator) !void`
  - Memory deallocation, I/O deinitialization
- `setRate(self: *Self, upstream_rate: f64) !f64`
  - Override sample rate
  - Sources define initial rate for downstream blocks
  - Upsamplers / downsamplers can multiply / divide the upstream rate, etc.

Creating Blocks - Data Types

- Custom data types can be made from `struct` or `union`

```
pub const WeatherPacket = struct {
    temperature: i8 = 0,
    humidity: u8 = 0,
    wind_speed: u8 = 0,
    wind_direction: enum { N, E, S, W } = .N,

    pub fn typeName() []const u8 {
        return "WeatherPacket";
    }
};
```

- Dynamically allocated types are supported too (see website)

Creating Blocks - Asynchronous Control

Creating Blocks - Asynchronous Control

- Ordinary functions can manipulate block state at runtime
  - Change parameters like filter cut-offs, AGC parameters, etc.
  - Tune frequency or trigger I/O

Creating Blocks - Asynchronous Control

- Ordinary functions can manipulate block state at runtime
  - Change parameters like filter cut-offs, AGC parameters, etc.
  - Tune frequency or trigger I/O
- Framework guarantees mutual exclusion

Creating Blocks - Asynchronous Control

- Ordinary functions can manipulate block state at runtime
  - Change parameters like filter cut-offs, AGC parameters, etc.
  - Tune frequency or trigger I/O
- Framework guarantees mutual exclusion
- Asynchronous calls made through `Flowgraph` `call()` API

Creating Blocks - Asynchronous Control

```
const radio = @import("radio");

pub const MultiplyConstantBlock = struct {
    block: radio.Block,
    constant: f32,

    pub fn init(constant: f32) MultiplyConstantBlock {
        return .{ .block = radio.Block.init(@This()), .constant = constant };
    }

    pub fn process(self: *MultiplyConstantBlock, x: []const f32, z: []f32) !radio.ProcessResult {
        for (x, 0..) |_, i| {
            z[i] = x[i] * self.constant;
        }
        return radio.ProcessResult.init(&[1]usize{x.len}, &[1]usize{x.len});
    }

    pub fn setConstant(self: *MultiplyConstantBlock, constant: f32) !void {
        if (constant > 9000) return error.OutOfBounds;
        self.constant = constant;
    }
}
...
const result = try flowgraph.call(&multiplyconstantblock.block, MultiplyConstantBlock.setConstrant, .{123});
```

Cross-compiling ZigRadio

Cross-compiling ZigRadio

- Build for host

```
$ zig build examples
$ file -b zig-out/bin/example-rtlsdr_wbfm_mono
ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 4.4.0, with debug_info, not stripped
$
```

Cross-compiling ZigRadio

- Build for host

```
$ zig build examples
$ file -b zig-out/bin/example-rtlsdr_wbfm_mono
ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 4.4.0, with debug_info, not stripped
$
```

- Build for aarch64 target

```
$ zig build -Dtarget=aarch64-linux-gnu examples
$ file -b zig-out/bin/example-rtlsdr_wbfm_mono
ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
for GNU/Linux 2.0.0, with debug_info, not stripped
$
```

Cross-compiling ZigRadio

- Build for host

```
$ zig build examples
$ file -b zig-out/bin/example-rtlsdr_wbfm_mono
ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 4.4.0, with debug_info, not stripped
$
```

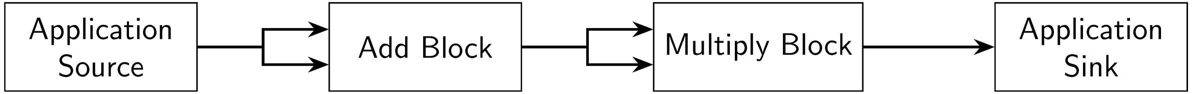
- Build for aarch64 target

```
$ zig build -Dtarget=aarch64-linux-gnu examples
$ file -b zig-out/bin/example-rtlsdr_wbfm_mono
ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
for GNU/Linux 2.0.0, with debug_info, not stripped
$
```

- Run on target

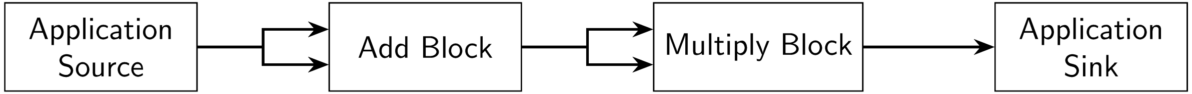
```
$ scp zig-out/bin/example-rtlsdr_wbfm_mono alarm@radio-pi4.local:
$ ssh radio-pi4.local
alarm@radio-pi4.local $ ./example-rtlsdr_wbfm_mono 99.9e6
```

Integrating ZigRadio




```
var source = radio.blocks.ApplicationSource(f32).init(10000);
var adder = radio.blocks.AddBlock(f32).init();
var multiplier = radio.blocks.MultiplyBlock(f32).init();
var sink = radio.blocks.ApplicationSink(f32).init();
...
```

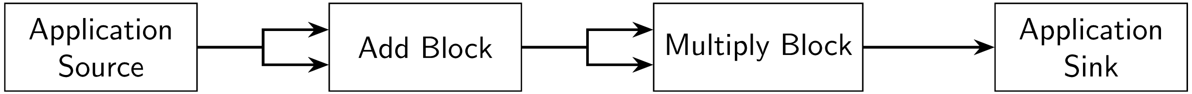
Integrating ZigRadio




```
var source = radio.blocks.ApplicationSource(f32).init(10000);
var adder = radio.blocks.AddBlock(f32).init();
var multiplier = radio.blocks.MultiplyBlock(f32).init();
var sink = radio.blocks.ApplicationSink(f32).init();
...
```

- Host applications can use ZigRadio as a signal processing engine

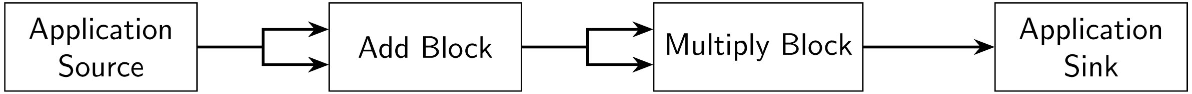
Integrating ZigRadio




```
var source = radio.blocks.ApplicationSource(f32).init(10000);
var adder = radio.blocks.AddBlock(f32).init();
var multiplier = radio.blocks.MultiplyBlock(f32).init();
var sink = radio.blocks.ApplicationSink(f32).init();
...
```

- Host applications can use ZigRadio as a signal processing engine
- `ApplicationSource` block injects samples into a flow graph

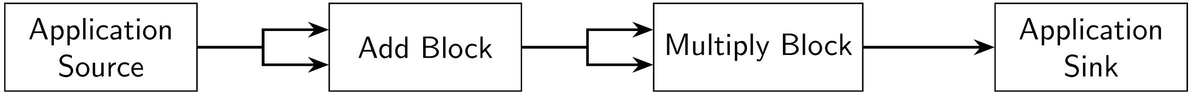
Integrating ZigRadio




```
var source = radio.blocks.ApplicationSource(f32).init(10000);
var adder = radio.blocks.AddBlock(f32).init();
var multiplier = radio.blocks.MultiplyBlock(f32).init();
var sink = radio.blocks.ApplicationSink(f32).init();
...
```

- Host applications can use ZigRadio as a signal processing engine
- `ApplicationSource` block injects samples into a flow graph
- `ApplicationSink` block consumes samples from a flow graph

Integrating ZigRadio




```
var source = radio.blocks.ApplicationSource(f32).init(10000);
var adder = radio.blocks.AddBlock(f32).init();
var multiplier = radio.blocks.MultiplyBlock(f32).init();
var sink = radio.blocks.ApplicationSink(f32).init();
...
```

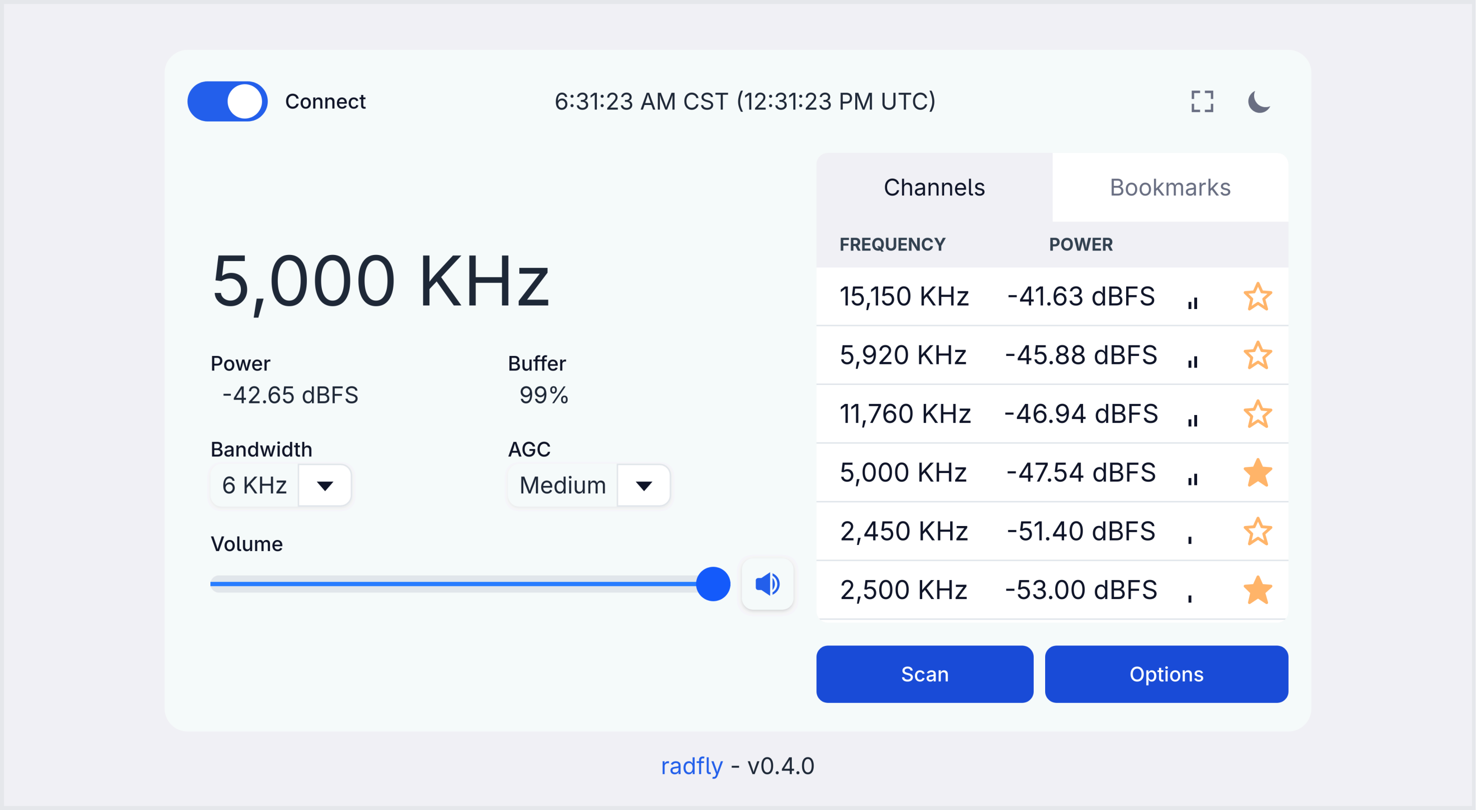
- Host applications can use ZigRadio as a signal processing engine
- `ApplicationSource` block injects samples into a flow graph
- `ApplicationSink` block consumes samples from a flow graph
- Thread-safe API with blocking and non-blocking variants

Integrating ZigRadio

```
try top.start();

// Write samples 1, 2, 3
try source.push(1);
try source.push(2);
try source.push(3);

// Wait for 3 samples available for reading
try sink.wait(3, null);

// Read three samples
std.debug.print("{any}\n", .{sink.pop()});
std.debug.print("{any}\n", .{sink.pop()});
std.debug.print("{any}\n", .{sink.pop()});

// Set end-of-stream on source
source.setEOS();

// Wait for flow graph collapse
_ = try top.wait();
```

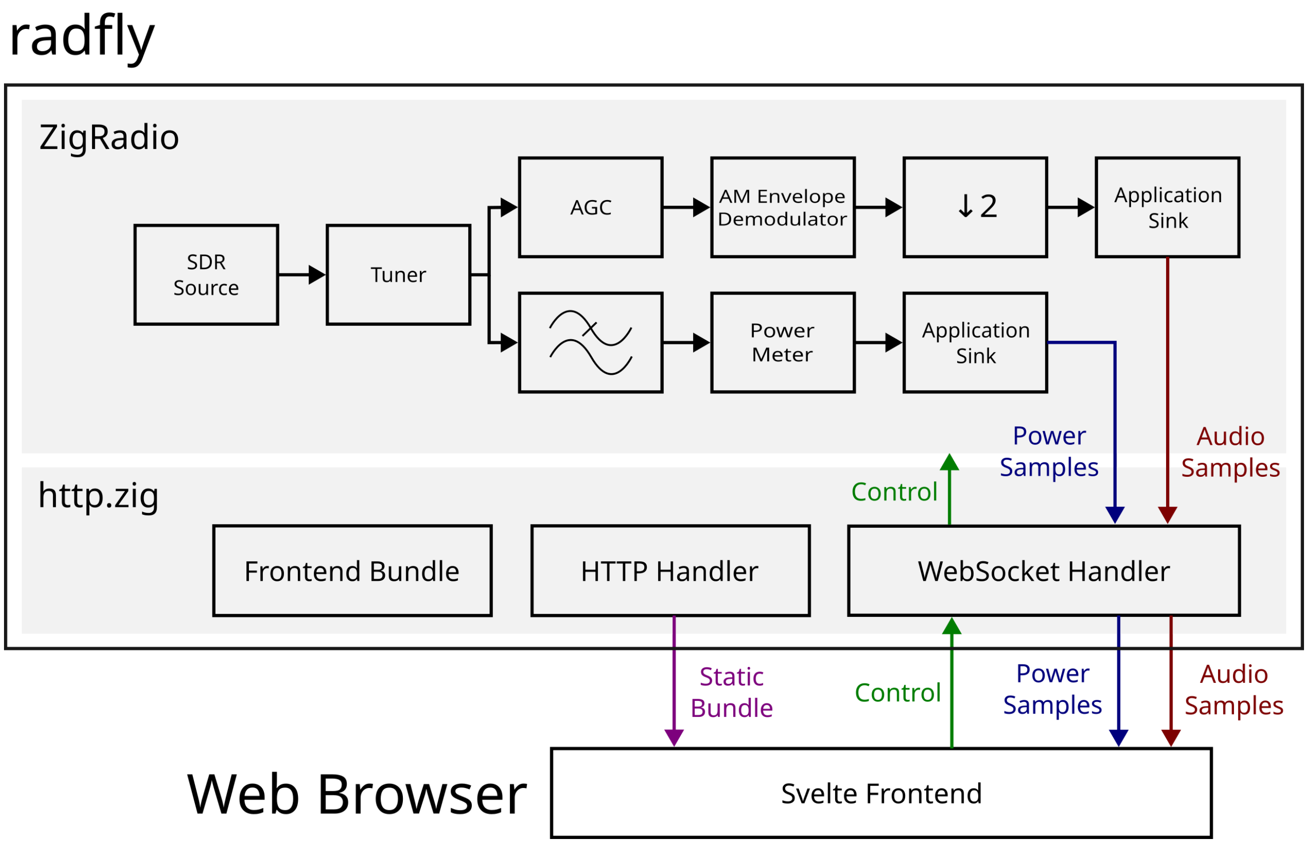
```
$ ./example
4
16
36
$
```

Integrating ZigRadio - radfly example



https://github.com/vsergeev/radfly

Integrating ZigRadio - radfly example



Website

ZigRadio

v0.10.0
GitHub

v@sergeev.io

ZigRadio is a lightweight flow graph signal processing framework for software-defined radio. It provides a suite of source, sink, and processing blocks, with a simple API for defining flow graphs, running flow graphs, and creating blocks. ZigRadio has an API similar to that of LuaRadio and is also MIT licensed.

ZigRadio can be used to rapidly prototype software radios, modulation/demodulation utilities, and signal processing experiments.

Example

Wideband FM Broadcast Radio Receiver

```
const std = @import("std");

const radio = @import("radio");

pub fn main() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};

    const frequency: f64 = 91.1e6; // 91.1 MHz

    var source = radio.blocks.RtlSdrSource.init(frequency - 250e3, 960000, .{});
    var if_translator = radio.blocks.FrequencyTranslatorBlock.init(-250e3);
    var if_filter = radio.blocks.LowpassFilterBlock(std.math.Complex(f32), 128).init(200e3, .{});
    var if_downsampler = radio.blocks.DownsamplerBlock(std.math.Complex(f32)).init(4);
    var fm_demod = radio.blocks.FrequencyDiscriminatorBlock.init(75e3);
    var af_filter = radio.blocks.LowpassFilterBlock(f32, 128).init(15e3, .{});
    var af_deemphasis = radio.blocks.FMDeemphasisFilterBlock.init(75e-6);
    var af_downsampler = radio.blocks.DownsamplerBlock(f32).init(5);
    var sink = radio.blocks.PulseAudioSink(1).init();

    var top = radio.Flowgraph.init(gpa.allocator(), .{ .debug = true });
    defer top.deinit();
    try top.connect(&source.block, &if_translator.block);
    try top.connect(&if_translator.block, &if_filter.block);
    try top.connect(&if_filter.block, &if_downsampler.block);
    try top.connect(&if_downsampler.block, &fm_demod.block);
    try top.connect(&fm_demod.block, &af_filter.block);
    try top.connect(&af_filter.block, &af_deemphasis.block);
    try top.connect(&af_deemphasis.block, &af_downsampler.block);
    try top.connect(&af_downsampler.block, &sink.block);

    _ = try top.run();
}
```

Check out some more examples of what you can build with ZigRadio.

https://zigradio.org

Reference Manual

ZigRadio

v0.10.0
GitHub

v@sergeev.io

Math Operations

AddBlock

Add two signals.

$$y[n] = x_1[n] + x_2[n]$$

`radio.blocks.AddBlock(comptime T: type).init()`

Comptime Arguments

- `T` (*type*): Complex(f32), f32, etc.

Type Signature

- `in1` *T*, `in2` *T* →□→ `out1` *T*

Example

```
var summer = radio.blocks.AddBlock(std.math.Complex(f32)).init();
try top.connectPort(&src1.block, "out1", &summer.block, "in1");
try top.connectPort(&src2.block, "out1", &summer.block, "in2");
try top.connect(&summer.block, &sink.block);
```

ComplexMagnitudeBlock

Compute the magnitude of a complex-valued signal.

$$y[n] = |x[n]|$$

$$y[n] = \sqrt{\mathrm{Re}(x[n])^2 + \mathrm{Im}(x[n])^2}$$

`radio.blocks.ComplexMagnitudeBlock.init()`

Type Signature

- `in1` *Complex(f32)* →□→ `out1` *f32*

Example

```
var mag = radio.blocks.ComplexMagnitudeBlock.init();
```

MultiplyBlock

https://zigradio.org/reference-manual.html

Roadmap

Roadmap

- Short-term: more blocks
  - Digital
  - Acceleration
  - Plotting sinks

Roadmap

- Short-term: more blocks
  - Digital
  - Acceleration
  - Plotting sinks
- Medium-term: platform support
  - Windows
  - WebAssembly

Roadmap

- Short-term: more blocks
  - Digital
  - Acceleration
  - Plotting sinks
- Medium-term: platform support
  - Windows
  - WebAssembly
- Long-term: TBD

Contributing

Contributing

- https://github.com/vsergeev/zigradio

Contributing

- https://github.com/vsergeev/zigradio
- Issues and PRs are appreciated

Contributing

- https://github.com/vsergeev/zigradio
- Issues and PRs are appreciated
- A lot of low hanging fruit

Contributing

- https://github.com/vsergeev/zigradio
- Issues and PRs are appreciated
- A lot of low hanging fruit
    - Rudimentary signal processing blocks

Contributing

- https://github.com/vsergeev/zigradio
- Issues and PRs are appreciated
- A lot of low hanging fruit
  - Rudimentary signal processing blocks
  - Asynchronous APIs for existing blocks

Contributing

- https://github.com/vsergeev/zigradio
- Issues and PRs are appreciated
- A lot of low hanging fruit
  - Rudimentary signal processing blocks
  - Asynchronous APIs for existing blocks
  - I/O blocks (SDRs, Audio, Network, etc.)

Try ZigRadio

- Clone:

```
$ git clone https://github.com/vsergeev/zigradio.git
$ cd zigradio
```

- Build examples:

```
$ zig build examples
```

- Try out one of the examples with an RTL-SDR dongle:

```
$ ./zig-out/bin/example-rtlsdr_wbfm_mono 89.7e6
```

- Optionally install VOLK, liquid-dsp, fftw for runtime acceleration.

Questions?