What HackRfDotnet Does

HackRfDotnet is a managed .NET wrapper and library that exposes and extends the C library shipped with the HackRF from Great Scott Gadgets.
It allows developers to interact with HackRF devices using idiomatic .NET patterns, making it easier to prototype radio workflows, stream signals, and integrate HackRF functionality into custom applications.

With HackRfDotnet, you can quickly set up device streams, apply signal processing pipelines, or integrate your HackRF device into dependency injection (DI) architectures for structured applications.

With HackRfDotnet's full-featured Digital Signal Processing (DSP) pipelines, you can configure and process a wide variety of signals, from analogue modulations such as FM and AM, to digital data streams including QAM, OFDM, BPSK, and QPSK. These pipelines allow for real-time demodulation, filtering, and transformation, enabling both experimentation and production-grade radio workflows.

Example Code Documentation

Below is a table of contents linking to our documentation and example programs, showing both basic usage and advanced scenarios.

Getting Started

01 Setting Up Radio – Learn how to initialize and configure your HackRF device.
02 Setting Up Radio In DI – Using HackRfDotnet with dependency injection for managed device lifecycles and service integration.

Advanced

05 Scanning Frequencies

> **CAUTION**
>
> **Legal Notice:** HackRF devices are capable of transmitting and receiving radio signals. Use of these devices is subject to local, regional, and national laws and regulations. HackRfDotnet and its maintainers (Realynx) provide this library for **educational, experimental, and research purposes only**.
>
> **You are solely responsible** for any use of your HackRF device, including compliance with spectrum licensing, transmission power limits, and prohibited frequency bands. HackRfDotnet, Realynx (including its members), and Great Scott Gadgets **are not responsible** for damages, legal penalties, or regulatory violations resulting from improper or unlawful use.
>
> **Always verify your local laws** before transmitting, and do not transmit on frequencies for which you do not have authorization.

Getting Started With HackRf Dotnet

The most basic way we can use HackRfDotnet is by playing an analogue audio stream.
Amplitude modulation - The phase of the audio signal is encoded in the changes of the amplitude on the carrier rf wave.
Frequency modulation - The phase of the audio signal is encoded in the changes of the phase on the carrier rf wave.

```
var rfDeviceService = new RfDeviceService();

Console.WriteLine("Looking for HackRf Device...");
var deviceList = rfDeviceService.FindDevices();

Console.WriteLine($"Found {deviceList.devicecount} HackRf devices... Opening Rx");
using var rfDevice = rfDeviceService.ConnectToFirstDevice();

if (rfDevice is null) {
    Console.WriteLine("Could not connect to Rf Device");
    return;
}

// Create an immutable read stream from an RF Device.
using var deviceStream = new IQDeviceStream(rfDevice);

// Open the receive channel on the SDR
deviceStream.OpenRx(SampleRate.FromMsps(20));
```

To play an FM stream you would use the following block of code.

```
// Tune the SDR to the target frequency and bandwidth
rfDevice.SetFrequency(Frequency.FromMHz(98.7f), Bandwidth.FromKHz(200));

// Create a SignalStream configured for FM decoding
var fmSignalStream = new FmSignalStream(deviceStream, Bandwidth.FromKHz(200), stereo: true);

// Create an AnaloguePlayer to play the FM audio stream
var fmPlayer = new AnaloguePlayer(fmSignalStream);
fmPlayer.PlayStreamAsync(rfDevice.Frequency, rfDevice.Bandwidth, SampleRate.FromKsps(48));
```

To play an AM stream you would use the following block of code.

```
// Tune the SDR to the target frequency and bandwidth
rfDevice.SetFrequency(Frequency.FromMHz(118.4f), Bandwidth.FromKHz(10));

// Create a SignalStream configured for AM decoding
var amSignalStream = new AmSignalStream(deviceStream, Bandwidth.FromKHz(10));

// Create an AnaloguePlayer to play the AM audio stream
var amPlayer = new AnaloguePlayer(amSignalStream);
amPlayer.PlayStreamAsync(rfDevice.Frequency, rfDevice.Bandwidth, SampleRate.FromKsps(48));
```

FmSignalStream and AmSignalStream are encapsulations of preconfigured SignalProcessingPipelines. These pipelines downsample, filter, and demodulate the tuned frequency from the Inphase & Quadrature stream into an analog audio signal. The resulting audio is then passed to NAudio, which resamples it to the configured playback rate—typically 48 kHz.

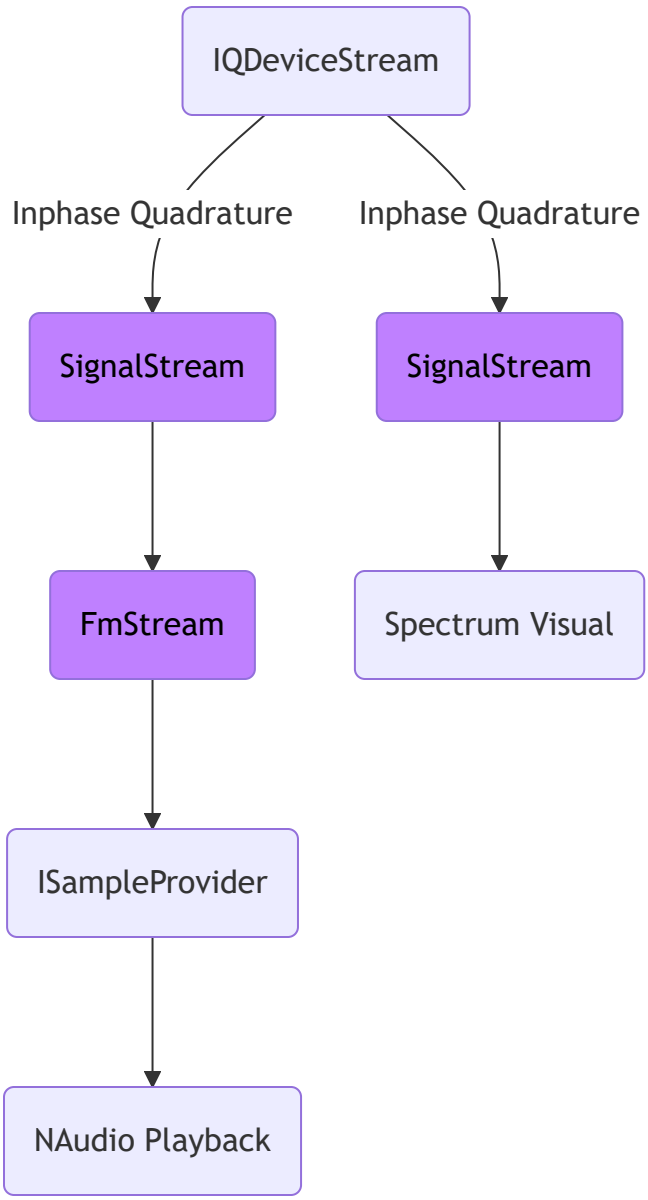
Signal Processing Pipelines

A Signal Processing Pipeline is a chain of effects applied to a signal from a SignalStream. Each SignalStream is constructed with a configured pipeline, which processes the incoming data before it is returned through Read. Without a pipeline, the SignalStream simply exposes raw capture data from the immutable IQDeviceStream, which serves as the root source of IQ samples.

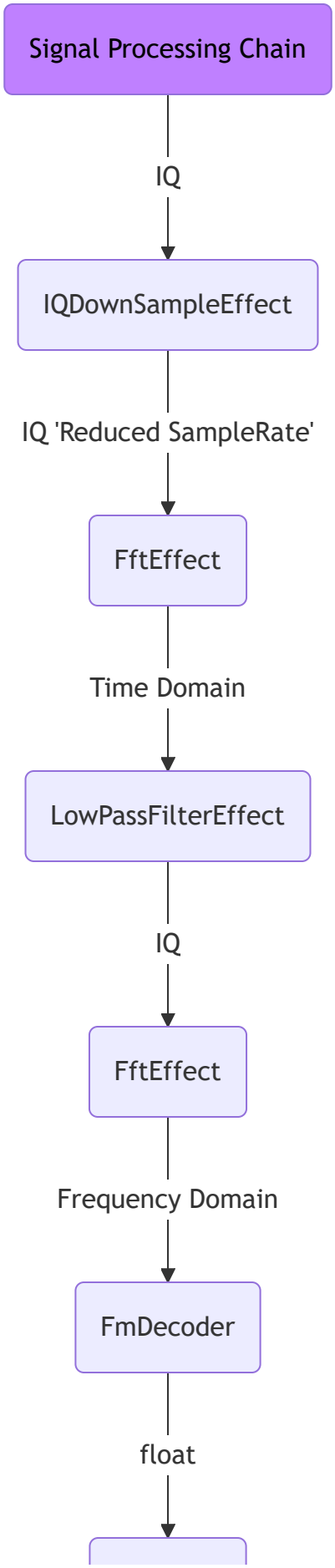
The example project demonstrates a typical digital signal processing workflow. The IQDeviceStream maintains a ring buffer filled by a background worker. Every SignalStream created from it also runs its own background task to execute the Signal Processing Pipeline, ensuring the stream's buffer remains populated and ready to read.

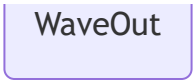
A pipeline can alter both the format and representation of the signal. For instance, an FMDecoder effect consumes IQ samples as input and outputs float samples. Each effect defines its expected input and output types, ensuring compatibility within the chain.

Signal Chain



Signal Processing Pipeline





Signal Pipeline Example

The following is an FM Decoder example for a Signal Processing Pipeline.
An FMStream will pre-configure this same processing pipeline before sending data to NAudio as an ISampleProvider.

```
// Create a processing pipeline.
var signalPipeline = new SignalProcessingPipeline<IQ>();

signalPipeline
    // Add a root effect, this is used to track the parent effect in the chain.
    .WithRootEffect(new IQDownSampleEffect(deviceStream.SampleRate,
        stationBandwidth.NyquistSampleRate, out reducedRate, out var producedChunkSize))

    // Add remaining effects as Child Effect.
    .AddChildEffect(new FftEffect(true, producedChunkSize))
    .AddChildEffect(new LowPassFilterEffect(reducedRate, stationBandwidth))
    .AddChildEffect(new FftEffect(false, producedChunkSize))
    .AddChildEffect(new FmDecoder());

// You can use a signal processing pipeline by passing in a buffer of data to be processed
to the AffectSignal function.
```

HackRfDotnet Using Dependency Injection

HackRfDotnet is highly DI-friendly and provides extension methods for HostBuilder to configure single or multiple device access. Below is a basic example using an analogue FM radio as an IHostedService to stream from a device stream singleton.

Configure the HostBuilder

Configure the HostBuilder as shown in the example using UseFirstRadioDevice. UseFirstRadioDevice is an extension method from HackRfDotnet that sets up single-device access. It adds an RfDevice singleton to the DI host. Adding this RfDevice also creates an IQ stream and opens the device RX, starting the flow of IQ samples into the ring buffer.

```
internal class Program {
    static async Task Main(string[] args) {
        // Create a new host builder.
        var appHost = new HostBuilder();
        appHost
            // Use HackRfDotnet's extension method to configure single device access.
            .UseFirstRadioDevice(SampleRate.FromMsps(20))
            .ConfigureServices(ConfigureServices);

        // Run the DI Host
        await appHost.RunConsoleAsync();
    }

    static void ConfigureServices(IServiceCollection serviceCollection) {
        // Configure additional services, FMRadioService in this case.
        serviceCollection
            .AddHostedService<FmRadioService>();
    }
}
```

Radio Service (Consumer)

This is our FmRadioService, this is our IHostedService that actually starts the buffer feeding into NAudio.

```
internal class FmRadioService : IHostedService, IDisposable {
    private readonly IDigitalRadioDevice _radioDevice;
    private readonly FmSignalStream _signalStream;

    // We get IDigitalRadioDevice from the DI, as was prepared earlier for us by
our "UseFirstRadioDevice".
```

```
    public FmRadioService(IDigitalRadioDevice radioDevice) {
        _radioDevice = radioDevice;

        if (_radioDevice.DeviceStream is null) {
            throw new Exception("Radio device stream cannot be null!");
        }

        // Create a new FmSignalStream, this pre-configures a DSP chain for DM demodulation.
        _signalStream = new FmSignalStream(_radioDevice.DeviceStream,
Bandwidth.FromKHz(200));
    }

    public void Dispose() {
        _signalStream.Dispose();
    }

    public Task StartAsync(CancellationToken cancellationToken) {
        _radioDevice.SetFrequency(Frequency.FromMHz(98.7f), Bandwidth.FromKHz(120));

        // Create an AnaloguePlayer and begin playing with NAudio.
        var fmPlayer = new AnaloguePlayer(_signalStream);
        fmPlayer.PlayStreamAsync(_radioDevice.Frequency, _radioDevice.Bandwidth,
SampleRate.FromKsps(48));

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken) {
        throw new NotImplementedException();
    }
}
```

DI and Single Device Access

When you configure your HostBuilder with the `UseFirstRadioDevice` extension method, your DI container is populated with a fully initialized and connected RfDevice singleton. During the instantiation of the RfDevice, an IQ stream is automatically created, and streaming begins immediately. While the device resides in the DI container, it continuously maintains a ring buffer in the background, ensuring that IQ samples are always available for consumption by your hosted services.

Multi Device Access

For scenarios involving multiple devices, you can use the `UseRfDeviceController` extension. This sets up a DeviceController service in your DI container, allowing your application to manage and interact with multiple connected devices. The DeviceController tracks all device instances, enabling you to reconnect

to the same device across different hosted services, coordinate parallel streaming, and manage device lifecycles consistently.

DI Runtime

The following graph illustrates the different DI states when using Multi-Device vs Single-Device HostBuilder configurations.