고급 SDR 구현: pySDR 및 Python을 이용한 ADALM-PLUTO에서 Jupiter SDR로의 마이그레이션을 위한 40단계 튜토리얼

Part I: 기초: 시스템 설정 및 첫 신호 수집 (1-10단계)

이 첫 번째 파트에서는 ADALM-PLUTO 생태계에 대한 기존 지식을 새로운 Jupiter 하드웨어에 연결하는 데 중점을 둡니다. 물리적 설정, 소프트웨어 환경, 그리고 pyadi-iio API의 중요한 차이점을 다루며, 첫 번째 IQ 데이터 캡처를 성공적으로 수행하는 것을 목표로 합니다.

1단계: 서론 - Pluto에서 Jupiter로의 도약

본 튜토리얼의 목표는 기존의 pySDR 및 ADALM-PLUTO 활용 능력을 바탕으로 전문가 수준의 SDR(Software-Defined Radio) 플랫폼인 Jupiter SDR을 마스터하는 것입니다. Jupiter SDR은 Analog Devices(ADI) SDR 제품군의 차세대 플랫폼으로, ADALM-PLUTO를 통해 SDR의 기본 원리를 익힌 사용자가 자연스럽게 다음 단계로 나아갈 수 있도록 설계되었습니다.[1] 이 두 플랫폼은

pyadi-iio 라이브러리와 ADI Linux라는 공통된 소프트웨어 생태계를 공유하므로, 기존의 지식과 코드를 상당 부분 재활용하여 원활한 마이그레이션 경로를 제공합니다.[3]

이 업그레이드는 단순히 하드웨어 사양을 개선하는 것을 넘어, SDR을 활용한 실험과 개발의 패러다임을 근본적으로 변화시킵니다. ADALM-PLUTO가 단일 채널 중심의 학습 및 프로토타이핑에 최적화되어 있다면, Jupiter SDR은 실제 통신 시스템에 필적하는 성능과 유연성을 제공합니다. 예를 들어, Jupiter SDR에 탑재된 ADRV9002 트랜시버의 넓은 대역폭은 단일 채널이 아닌 전체 셀룰러 대역을 한 번에 수신하고 분석할 수 있게 해줍니다.[2] 또한, 강력한 MPSoC(Multi-Processor System-on-Chip)는 데이터 처리의 병목 현상을 호스트 PC에서 SDR 장치 자체로 옮겨, 현장에서 독립적으로 작동하는 자율적인 애플리케이션 개발을 가능하게 합니다.[5] 본 튜토리얼은 이러한 하드웨어의 잠재력을 최대한 활용하여, pySDR.org에서 다루는

이론적 개념들을 실제 고성능 시스템에서 구현하는 구체적인 방법을 40단계에 걸쳐 안내할 것입니다.

2단계: 하드웨어 아키텍처 비교 분석

ADALM-PLUTO에서 Jupiter SDR로의 전환을 이해하기 위해서는 두 플랫폼의 핵심 하드웨어 구성 요소 차이를 명확히 인지해야 합니다. 주요 차이점은 RF 트랜시버와 온보드 프로세싱을 담당하는 SoC(System-on-Chip)에서 비롯됩니다.

- RF 트랜시버: AD9363 vs. ADRV9002
  ADALM-PLUTO는 AD9363 RFIC를 사용하는 반면, Jupiter SDR은 훨씬 더 진보된 ADRV9002를 탑재합니다. ADRV9002는 30 MHz에서 6 GHz까지 더 넓은 주파수 범위를 지원하며, 최대 RF 대역폭 역시 Pluto의 20 MHz(공식) 또는 56 MHz(확장)를 뛰어넘는 최대 40 MHz(채널당)를 제공합니다.2 또한, ADRV9002는 협대역 및 광대역 파형을 위한 DPD(Digital Pre-Distortion), 빠른 주파수 호핑(Fast Frequency Hopping), 다중 칩 동기화와 같은 고급 기능들을 내장하고 있어 미션 크리티컬 통신과 같은 전문적인 애플리케이션에 적합합니다.4
- SoC(System-on-Chip): Zynq-7010 vs. Zynq UltraScale+ MPSoC-3EG
  프로세싱 능력의 차이는 더욱 극적입니다. ADALM-PLUTO의 Zynq-7010은 28nm 공정의 듀얼 코어 ARM Cortex-A9 프로세서를 탑재하고 있습니다.7 반면, Jupiter SDR의 Zynq UltraScale+ MPSoC-3EG는 16nm FinFET+ 공정 기술을 기반으로 한 이기종(heterogeneous) 아키텍처를 자랑합니다.7 이 MPSoC는 애플리케이션 처리를 위한 64비트 쿼드 코어 ARM Cortex-A53(최대 1.5 GHz)과 실시간 제어를 위한 듀얼 코어 ARM Cortex-R5F(최대 600 MHz)를 통합하고 있습니다.5 이러한 구조는 복잡한 애플리케이션 소프트웨어와 결정성(deterministic)이 요구되는 실시간 신호 처리를 동시에, 그리고 효율적으로 수행할 수 있게 합니다. 또한, MPSoC-3EG는 Zynq-7010에 비해 훨씬 더 많은 154k개의 프로그래머블 로직 셀과 360개의 DSP 슬라이스를 제공하여, FPGA를 활용한 하드웨어 가속의 범위를 크게 확장합니다.2

이러한 하드웨어의 발전은 pySDR 실습에 다음과 같은 영향을 미칩니다. 더 넓은 대역폭은 스펙트럼 분석의 범위를 넓히고, 강력한 MPSoC는 실시간 신호 처리를 호스트 PC의 부담 없이 장치 내에서 직접 수행할 수 있게 하여, 진정한 임베디드 SDR 시스템 구축의 가능성을 열어줍니다.

표 **1:** 하드웨어 아키텍처 비교**: ADALM-PLUTO vs. Jupiter SDR**

기능 (Feature)	ADALM-PLUTO	Jupiter SDR	pySDR 실습에 미치는 영향

RF 트랜시버	AD9363	ADRV9002	고급 RF 기능(DPD, 주파수 호핑) 실험 가능
주파수 범위	325 MHz – 3.8 GHz (확장 시 70 MHz - 6 GHz)	30 MHz – 6 GHz	VHF/UHF 대역을 포함한 더 넓은 스펙트럼 탐색 가능
최대 **RF** 대역폭	20 MHz (확장 시 56 MHz)	40 MHz (채널당)	WiFi, LTE 등 광대역 신호의 전체 대역 동시 수신 및 분석
SoC	Xilinx Zynq-7010	Xilinx Zynq UltraScale+ MPSoC-3EG	온보드 프로세싱 능력 대폭 향상
애플리케이션 프로세서	듀얼 코어 ARM Cortex-A9 (32-bit)	쿼드 코어 ARM Cortex-A53 (64-bit)	복잡한 Python 기반 신호 처리 알고리즘을 장치 내에서 직접 실행
실시간 프로세서	없음	듀얼 코어 ARM Cortex-R5F	결정성이 요구되는 제어 루프 및 실시간 작업 처리 가능
프로그래머블 로직 셀	28k	154k	대규모 병렬 신호 처리 알고리즘의 FPGA 하드웨어 가속
DSP 슬라이스	80	360	FFT, FIR 필터 등 연산 집약적 작업의 하드웨어 가속 능력 증대
온보드 메모리	512 MB DDR3L	2 GB DDR4	대용량 IQ 데이터 버퍼링 및 온보드 처리에 유리

주요 인터페이스	USB 2.0	USB 3.1 Gen 1, 1GbE Ethernet, DisplayPort	호스트와의 고속 데이터 전송 및 독립형 장치로 활용 가능

3단계**:** 개봉 및 물리적 연결

Jupiter SDR 하드웨어를 처음 다룰 때는 각 포트의 기능을 정확히 파악하는 것이 중요합니다. 장치는 견고한 알루미늄 케이스에 내장되어 있어 다양한 환경에서의 프로토타이핑에 적합합니다.[2]

1. 포트 식별: 장치 전면과 후면 패널을 확인하여 다음 포트들을 식별합니다.
   - **RF** 포트: 2개의 수신(RX) 포트와 2개의 송신(TX) 포트가 SMA 커넥터 형태로 제공됩니다. 각 포트는 ADRV9002의 독립적인 채널에 해당합니다.
   - **USB Type-C** 포트: 데이터 통신과 전원 공급을 위한 두 개의 USB-C 포트가 있습니다. 하나는 데이터 및 전원(Power Sink 5V/3A)을, 다른 하나는 전원 전용(Power Sink 5V, 9V/3A)으로 사용될 수 있습니다.[2]
   - **Ethernet** 포트: 1000BASE-T RGMII 이더넷 포트는 안정적인 네트워크 연결을 제공합니다.[2]
   - **DisplayPort**: Mali-400 MP2 그래픽 프로세서의 출력을 통해 GUI 기반 애플리케이션을 직접 실행할 수 있습니다.[2]
   - 기타 포트: Micro-USB 디버그 인터페이스, SATA 3, GPIO 핀 등 다양한 확장 인터페이스가 제공됩니다.[2]
2. 안테나 연결: pySDR 실습을 위해 수신할 주파수 대역에 맞는 안테나를 RX1 포트에 연결합니다. FM 방송 수신을 위해서는 VHF 대역 안테나를, WiFi 신호 분석을 위해서는 2.4 GHz 또는 5 GHz 안테나를 사용합니다.
3. 전원 연결: 가장 간단한 방법은 데이터 통신용 USB-C 포트를 호스트 PC의 USB 3.0 이상 포트에 연결하는 것입니다. 이 연결만으로 데이터 통신과 전원 공급이 동시에 이루어집니다. 또는, 802.3at PoE+(Power over Ethernet)를 지원하는 네트워크 스위치에 이더넷 케이블을 연결하여 전원을 공급할 수도 있습니다.[2]

4단계**:** 호스트 환경 및 드라이버 설치

Jupiter SDR은 ADALM-PLUTO와 동일한 ADI Linux 생태계 및 libiio 라이브러리를 기반으로 하므로, 호스트 PC 환경 설정 과정은 매우 유사합니다. 이미 Pluto를 사용해 본 경험이 있다면 이

과정은 익숙할 것입니다.[10]

1. **Linux** 환경 준비: Windows나 macOS보다는 네이티브 Linux 또는 가상 머신(VM)을 통한 Ubuntu 환경 사용이 권장됩니다. 이는 드라이버 및 라이브러리 호환성을 보장하는 가장 확실한 방법입니다. pySDR.org의 가이드에 따라 VirtualBox와 Ubuntu 22 LTS를 설치할 수 있습니다.[10]
2. **libiio** 설치: libiio는 ADI의 하드웨어를 제어하기 위한 핵심 C 라이브러리입니다. 소스 코드에서 직접 빌드하거나 패키지 매니저를 통해 설치할 수 있습니다. libiio와 함께 Python 바인딩(pylibiio)이 반드시 설치되어야 pyadi-iio가 작동합니다.[11]
3. **libadrv9002** 설치: pyadi-iio가 ADRV9002의 모든 기능을 활용하기 위해서는 libadrv9002라는 추가적인 라이브러리가 필요합니다.[12] 이 라이브러리는 ADRV9002의 복잡한 설정과 캘리브레이션을 관리하는 역할을 합니다. 일반적으로 libiio와 함께 ADI의 소스 저장소에서 빌드하여 설치합니다.
4. **pyadi-iio** 설치: 필요한 모든 C 라이브러리가 설치되었다면, Python 패키지 관리자인 pip을 사용하여 pyadi-iio를 설치합니다.
   Bash
   ```
   pip install pyadi-iio
   ```

   설치 후, Python 인터프리터에서 import iio와 import adi를 실행하여 오류 없이 라이브러리가 로드되는지 확인합니다. 이는 libiio와 pyadi-iio가 모두 올바르게 설치되었음을 의미합니다.[11]

5단계**:** 네트워크 연결 및 첫 통신

Jupiter SDR의 전원을 켜고 이더넷 케이블을 로컬 네트워크에 연결하면, 장치는 DHCP를 통해 자동으로 IP 주소를 할당받습니다. 호스트 PC에서 이 장치를 찾아 통신을 시작하는 과정은 다음과 같습니다.

1. 장치 검색: 터미널에서 iio_info 유틸리티를 사용하여 네트워크에 연결된 IIO 장치를 검색합니다.
   Bash
   ```
   iio_info -s
   ```

   이 명령은 네트워크를 스캔하여 발견된 IIO 장치의 목록과 해당 IP 주소를 출력합니다. Jupiter SDR이 adrv9002-phy와 같은 이름으로 나타날 것입니다. 이 과정은 Pluto를 IP로 연결할 때와 동일한 원리입니다.[13]
2. **IP** 주소 확인: iio_info로 장치를 찾지 못한 경우, 공유기의 관리 페이지 등에서 'jupiter' 또는 'analog'와 같은 호스트 이름을 가진 장치를 찾아 IP 주소를 직접 확인할 수 있습니다.
3. **SSH** 연결: 확인된 IP 주소를 사용하여 Jupiter SDR에 SSH로 접속합니다. 기본 사용자 이름은 root, 비밀번호는 analog입니다.

Bash
```
ssh root@<jupiter_ip_address>
```

성공적으로 접속하면 Jupiter SDR의 ARM Cortex-A53 프로세서에서 실행 중인 ADI Linux 배포판의 셸 프롬프트가 나타납니다. uname -a와 같은 명령어를 실행하여 리눅스 커널 정보를 확인하면, 이 장치가 단순한 RF 주변기기가 아니라 강력한 임베디드 컴퓨터임을 실감할 수 있습니다. 이 SSH 연결은 향후 온보드에서 직접 Python 스크립트를 실행하거나 시스템 설정을 변경하는 데 사용됩니다.

6단계: pyadi-iio ADRV9002 클래스 소개

pyadi-iio 라이브러리의 핵심 철학은 복잡한 하드웨어를 추상화하여 사용하기 쉬운 Python 클래스로 제공하는 것입니다.[12] ADALM-PLUTO를 위해

adi.Pluto 클래스를 사용했던 것처럼, Jupiter SDR을 제어하기 위해서는 adi.adrv9002 클래스를 사용합니다.[15]

이 클래스는 pyadi-iio의 다른 디바이스 클래스와 마찬가지로, URI(Uniform Resource Identifier)를 통해 특정 하드웨어 인스턴스에 연결됩니다. 네트워크를 통해 연결된 Jupiter SDR의 인스턴스를 생성하는 기본 코드는 다음과 같습니다.

Python
```
import adi

# 네트워크 URI를 사용하여 Jupiter SDR 객체 생성
# <jupiter_ip_address>는 5단계에서 확인한 장치의 IP 주소로 대체합니다.
uri = "ip:<jupiter_ip_address>"
sdr = adi.adrv9002(uri=uri)

print(f"Connected to Jupiter SDR with API version: {sdr.api_version}")
```

adi.Pluto 클래스와 비교했을 때, adi.adrv9002 클래스는 훨씬 더 풍부하고 세분화된 속성(properties)과 메서드(methods)를 제공합니다. 이는 ADRV9002 트랜시버 자체가 AD9363보다 훨씬 더 많은 기능과 설정 옵션을 가지고 있기 때문입니다. 예를 들어, 자동 이득 제어(AGC), DC 오프셋, 직교 오차 보정(quadrature error correction) 등 다양한 캘리브레이션 루프를 개별적으로 제어할 수 있는 속성들이 포함되어 있습니다.[15] 다음 단계에서는 이 API의

구조적 차이를 자세히 분석할 것입니다.

7단계: API 전환 - Pluto 개념을 Jupiter에 매핑하기

ADALM-PLUTO에서 Jupiter SDR로 코드를 마이그레이션할 때 가장 중요한 변화는 단일 채널 중심의 암시적(implicit) API에서 다중 채널 중심의 명시적(explicit) API로의 전환을 이해하는 것입니다. ADALM-PLUTO의 adi.Pluto 클래스는 주로 하나의 RX/TX 쌍을 다루는 것처럼 설계되어 있어 sdr.rx_lo와 같은 속성은 기본 채널을 제어합니다. 반면, Jupiter SDR의 adi.adrv9002 클래스는 2x2 MIMO를 지원하는 하드웨어의 특성을 그대로 반영합니다.[4] 따라서 모든 채널 관련 설정은

rx0_, rx1_, tx0_, tx1_과 같은 접두사를 사용하여 제어할 채널을 명시적으로 지정해야 합니다.

이러한 코드의 장황함 증가는 복잡성을 더하는 것이 아니라, 더 강력하고 유연한 하드웨어 아키텍처를 직접적으로 제어할 수 있게 해주는 필수적인 설계입니다. 개발자는 처음부터 여러 개의 독립적인 데이터 경로를 다루고 있다는 사실을 인지하고 코드를 작성해야 합니다. 예를 들어, 수신 중심 주파수를 설정하는 것은 더 이상 하나의 값을 바꾸는 것이 아니라, sdr.rx0_lo와 sdr.rx1_lo라는 두 개의 독립적인 지역 발진기(Local Oscillator)를 각각 설정하는 작업이 됩니다.

아래 표는 기존 adi.Pluto 코드에서 사용되던 주요 개념들을 adi.adrv9002 클래스에 맞게 변환하는 방법을 요약한 "로제타 스톤" 역할을 합니다. 이 표는 사용자의 기존 지식을 새로운 플랫폼에 적용하는 데 있어 가장 중요한 참조 자료가 될 것입니다.

표 2: pyadi-iio API 변환: adi.Pluto에서 adi.adrv9002로

개념 (Concept)	adi.Pluto 속성	adi.adrv9002 동등 속성	새로운 기능에 대한 참고사항
장치 URI	adi.Pluto(uri=...)	adi.adrv9002(uri=...)	클래스 이름만 변경됩니다.
수신 중심 주파수	sdr.rx_lo	sdr.rx0_lo, sdr.rx1_lo	두 개의 수신 채널을 독립적으로 튜닝할 수 있습니다.
송신 중심 주파수	sdr.tx_lo	sdr.tx0_lo, sdr.tx1_lo	두 개의 송신 채널을 독립적으로 튜닝할 수 있습니다.

수신 샘플 레이트	sdr.sample_rate	sdr.rx0_sample_rate, sdr.rx1_sample_rate	각 수신 채널의 샘플 레이트를 개별적으로 설정합니다.
송신 샘플 레이트	sdr.sample_rate	sdr.tx0_sample_rate, sdr.tx1_sample_rate	각 송신 채널의 샘플 레이트를 개별적으로 설정합니다.
수신 **RF** 대역폭	sdr.rx_rf_bandwidth	sdr.rx0_rf_bandwidth, sdr.rx1_rf_bandwidth	각 수신 채널의 아날로그 필터 대역폭을 설정합니다.
이득 제어 모드	sdr.gain_control_mode_chan0	sdr.gain_control_mode_chan0, sdr.gain_control_mode_chan1	각 수신 채널의 AGC 모드('manual', 'fast_attack' 등)를 설정합니다.
수신 하드웨어 이득	sdr.rx_hardwaregain_chan0	sdr.rx_hardwaregain_chan0, sdr.rx_hardwaregain_chan1	'manual' 모드일 때 각 수신 채널의 이득을 dB 단위로 설정합니다.
IQ 데이터 수신	samples = sdr.rx()	samples = sdr.rx()	rx_enabled_channels 속성에 따라 활성화된 채널의 데이터가 리스트 형태로 반환됩니다. (예: [rx0_data, rx1_data])
IQ 데이터 송신	sdr.tx(samples)	sdr.tx(samples)	tx_enabled_channels 속성에 따라 활성화된 채널로 데이터를 전송합니다.
수신 버퍼 크기	sdr.rx_buffer_size	sdr.rx_buffer_size	rx() 호출 시

			반환되는 샘플 수를 설정합니다. (두 채널 공통)

8단계: 첫 Jupiter SDR 수신기 - 기본 Python 스크립트

이제 API의 차이점을 이해했으므로, pySDR.org의 Pluto 튜토리얼에 있는 간단한 수신 스크립트를 Jupiter SDR용으로 수정해 보겠습니다.[10] 이 스크립트는 장치를 초기화하고, 하나의 수신 채널(RX0)에 대한 필수 파라미터를 설정한 후,

sdr.rx() 메서드를 사용하여 한 번의 데이터 캡처를 수행합니다. 이 코드는 ADI 엔지니어존 포럼에서 검증된 예제 코드를 기반으로 작성되었습니다.[16]

이 스크립트는 지역 FM 방송국 주파수(예: 93.1 MHz)에 맞춰져 있습니다.

Python

```
import adi
import numpy as np
import matplotlib.pyplot as plt

# 1. Jupiter SDR에 연결
uri = "ip:192.168.2.1"  # 실제 Jupiter SDR의 IP 주소로 변경하세요.
try:
    sdr = adi.adrv9002(uri=uri)
    print(f"Successfully connected to Jupiter SDR @ {uri}")
except Exception as e:
    print(f"Error connecting to Jupiter SDR: {e}")
    exit()

# 2. 수신 채널 0 (RX0) 파라미터 설정
center_freq = int(93.1e6)     # 튜닝할 중심 주파수 (Hz)
sample_rate = int(2e6)        # 샘플 레이트 (Hz)
rx_gain = 70                  # 수신 이득 (dB)
num_samples = 2**16           # 수신할 샘플 수
```

```
sdr.rx_enabled_channels =   # RX0 채널만 활성화
sdr.rx0_lo = center_freq
sdr.rx0_sample_rate = sample_rate
sdr.rx0_rf_bandwidth = sample_rate
sdr.gain_control_mode_chan0 = "manual"
sdr.rx_hardwaregain_chan0 = rx_gain
sdr.rx_buffer_size = num_samples

print("--- RX0 Configuration ---")
print(f"Center Frequency: {sdr.rx0_lo / 1e6} MHz")
print(f"Sample Rate: {sdr.rx0_sample_rate / 1e6} MSPS")
print(f"RF Bandwidth: {sdr.rx0_rf_bandwidth / 1e6} MHz")
print(f"Gain: {sdr.rx_hardwaregain_chan0} dB")
print(f"Buffer Size: {sdr.rx_buffer_size} samples")

# 3. 데이터 수신
# rx()는 활성화된 채널의 데이터를 리스트로 반환합니다.
# 여기서는 RX0만 활성화했으므로 리스트의 첫 번째 요소()를 가져옵니다.
print("\nReceiving samples...")
samples = sdr.rx()
print(f"Received {len(samples)} samples.")

# 4. 수신된 데이터 시각화 (다음 단계에서 자세히 다룸)
plt.figure(figsize=(12, 6))
plt.plot(np.real(samples[:1000]))
plt.plot(np.imag(samples[:1000]))
plt.title("Time Domain IQ Samples (First 1000)")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.legend(["I", "Q"])
plt.grid(True)
plt.show()
```

이 스크립트를 실행하면 Jupiter SDR이 지정된 주파수에서 IQ 데이터를 수집하고, 수집된 샘플의 I (실수부)와 Q (허수부) 값을 시간 영역에서 간단히 플로팅합니다. 성공적으로 실행된다면, Jupiter SDR의 기본 수신 기능을 완벽하게 제어할 수 있게 된 것입니다.

9단계: 첫 IQ 데이터 캡처 분석

8단계에서 수신한 IQ 데이터를 분석하여 하드웨어가 올바르게 설정되었고 유효한 신호를

수신하고 있는지 확인하는 것은 매우 중요합니다. pySDR.org의 접근 방식에 따라 numpy와 matplotlib 라이브러리를 사용하여 시간 영역 파형과 주파수 영역 스펙트럼을 시각화합니다.[17]

1. 시간 영역 분석: 8단계의 코드 마지막 부분에서 이미 처음 1000개의 샘플에 대한 시간 영역 파형을 플로팅했습니다. 이 플롯을 통해 신호의 진폭이 적절한지, 클리핑(clipping)이 발생하지 않았는지 등을 시각적으로 확인할 수 있습니다. 수신 이득(rx_gain)이 너무 높으면 신호가 포화되어 왜곡될 수 있고, 너무 낮으면 신호가 노이즈에 묻혀 보이지 않을 수 있습니다.
2. 주파수 영역 분석 **(FFT)**: 신호의 스펙트럼을 확인하는 것이 가장 확실한 검증 방법입니다. 수신된 샘플에 대해 고속 푸리에 변환(FFT)을 수행하여 전력 스펙트럼 밀도(PSD)를 계산하고 플로팅합니다. 이는 튜닝한 중심 주파수 주변에 어떤 신호들이 존재하는지 보여줍니다.

아래 코드를 8단계 스크립트의 마지막 부분에 추가하여 PSD 플롯을 생성할 수 있습니다.

Python

```
#... (8단계 코드의 데이터 수신 부분 이후)

# 5. 주파수 영역 분석 (FFT)
fft_size = 1024
psd = np.abs(np.fft.fftshift(np.fft.fft(samples, fft_size)))**2
freqs = np.fft.fftshift(np.fft.fftfreq(fft_size, 1/sample_rate))

plt.figure(figsize=(12, 6))
plt.plot(freqs / 1e6, 10 * np.log10(psd))
plt.title("Power Spectral Density (PSD)")
plt.xlabel("Frequency Offset (MHz)")
plt.ylabel("Power (dB)")
plt.grid(True)
plt.show()

# 6. 스펙트로그램 (Waterfall) 분석
plt.figure(figsize=(12, 6))
plt.specgram(samples, NFFT=1024, Fs=sample_rate)
plt.title("Spectrogram")
plt.xlabel("Time (s)")
plt.ylabel("Frequency (Hz)")
plt.show()
```

FM 방송국 주파수(예: 93.1 MHz)에 튜닝했다면, PSD 플롯의 중심(0 MHz 오프셋) 근처에서

강력한 신호 피크를 관찰할 수 있어야 합니다. 스펙트로그램은 시간에 따른 주파수 변화를 보여주므로, 음성이나 음악에 따라 변화하는 FM 신호의 특징을 시각적으로 확인할 수 있습니다. 이 분석을 통해 Jupiter SDR이 성공적으로 RF 신호를 디지털 IQ 데이터로 변환했음을 최종적으로 검증할 수 있습니다.

10단계: 첫 Jupiter SDR 송신기 - 기본 톤 생성

수신 기능 검증을 마쳤으니, 이제 송신 기능을 테스트하여 Jupiter SDR의 기본 RF 송수신 루프를 완성합니다. 가장 간단한 송신 테스트는 내장된 DDS(Direct Digital Synthesizer)를 사용하여 단일 주파수의 순수한 사인파(톤)를 생성하고 이를 공중으로 방사하는 것입니다.

이 테스트를 위해서는 두 개의 장비가 필요합니다. 하나는 신호를 송신할 Jupiter SDR이고, 다른 하나는 송신된 신호를 수신하여 확인할 장비입니다. ADALM-PLUTO, 다른 SDR 수신기(예: RTL-SDR), 또는 스펙트럼 분석기를 사용할 수 있습니다. 주의: 안테나를 연결하여 신호를 송신할 때는 해당 국가의 전파 법규를 반드시 준수해야 합니다. 허가되지 않은 주파수에서 높은 출력으로 송신하는 것은 불법일 수 있습니다. 안전한 실내 테스트를 위해 낮은 이득과 더미 로드(dummy load) 사용을 권장합니다.

다음 Python 스크립트는 Jupiter SDR의 송신 채널 0(TX0)을 설정하고 1 MHz 주파수의 톤 신호를 생성하여 1 GHz로 송신합니다.

Python

```
import adi
import time

# 1. Jupiter SDR에 연결
uri = "ip:192.168.2.1"  # 실제 Jupiter SDR의 IP 주소로 변경하세요.
sdr = adi.adrv9002(uri=uri)

# 2. 송신 채널 0 (TX0) 파라미터 설정
center_freq = int(1e9)          # 1 GHz 중심 주파수
sample_rate = int(5e6)          # 5 MSPS 샘플 레이트
tx_gain = -10                   # 송신 이득 (dB), 낮은 값으로 시작
tone_freq = int(1e6)            # 1 MHz 톤 주파수

sdr.tx_enabled_channels =     # TX0 채널만 활성화
```

```
sdr.tx0_lo = center_freq
sdr.tx0_sample_rate = sample_rate
sdr.tx_hardwaregain_chan0 = tx_gain
sdr.tx_cyclic_buffer = True    # DDS는 보통 순환 버퍼 모드에서 사용

print("--- TX0 Configuration ---")
print(f"Center Frequency: {sdr.tx0_lo / 1e6} MHz")
print(f"Sample Rate: {sdr.tx0_sample_rate / 1e6} MSPS")
print(f"Gain: {sdr.tx_hardwaregain_chan0} dB")
print(f"Tone Frequency: {tone_freq / 1e6} MHz")

# 3. DDS를 사용하여 단일 톤 신호 생성
# dds_single_tone(frequency, scale, channel)
# scale은 0.0에서 1.0 사이의 값으로 진폭을 조절합니다.
sdr.dds_single_tone(tone_freq, 0.9, 0) # 채널 0에 톤 설정

print("\nTransmitting 1 MHz tone at 1 GHz...")
print("Press Ctrl+C to stop.")

try:
    # DDS는 백그라운드에서 계속 신호를 생성하므로,
    # 스크립트가 종료되지 않도록 유지합니다.
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    print("\nStopping transmission.")
    # DDS 출력 비활성화
    sdr.dds_single_tone(0, 0, 0)
    sdr.tx_destroy_buffer()
```

이 스크립트를 실행하면 Jupiter SDR은 1 GHz 중심 주파수에서 1 MHz 오프셋을 가진, 즉 1.001 GHz 주파수의 연속적인 톤 신호를 송신하기 시작합니다. 다른 수신기로 이 주파수를 모니터링하면 강력하고 깨끗한 단일 피크 신호를 확인할 수 있습니다. 이로써 Jupiter SDR의 기본적인 송신 및 수신 기능 제어를 모두 마스터했으며, 이제 더 복잡한 pySDR 실습을 진행할 준비가 되었습니다.

Part II: 핵심 개념**: Jupiter**에서 **pySDR** 커리큘럼 구현 **(11-25**단계**)**

이 파트에서는 pySDR.org의 핵심 디지털 신호 처리(DSP) 및 통신 이론 챕터들을 Jupiter SDR에 특화된 Python 코드로 구현합니다. 각 단계는 ADALM-PLUTO보다 우수한 Jupiter SDR의 하드웨어가 기존의 실습을 어떻게 향상시키는지 보여줄 것입니다.

11단계: 더 넓은 대역폭을 이용한 주파수 영역 분석

pySDR.org의 주파수 영역 챕터에서 다루는 FFT 및 스펙트로그램 실습은 SDR의 가장 기본적인 활용법입니다.[18] ADALM-PLUTO를 사용할 때는 한 번에 수 MHz의 대역폭만 볼 수 있었지만, Jupiter SDR은 최대 40 MHz의 RF 대역폭을 제공하여 스펙트럼 분석의 패러다임을 바꿉니다.[2] 이는 더 이상 특정 신호 채널 하나를 보는 '채널 뷰어'가 아니라, 여러 통신 시스템이 공존하는 넓은 스펙트럼을 한눈에 파악하는 '스펙트럼 분석기'로서의 역할을 수행할 수 있음을 의미합니다.

이 실습에서는 Jupiter SDR의 넓은 대역폭을 활용하여 전체 FM 방송 대역(미국 기준 약 88-108 MHz)의 일부를 한 번에 캡처하고 분석합니다.

Python

```
import adi
import numpy as np
import matplotlib.pyplot as plt

# Jupiter SDR 연결
uri = "ip:192.168.2.1"
sdr = adi.adrv9002(uri=uri)

# 광대역 수신 설정 (FM 방송 대역 중심)
center_freq = int(98e6)       # 98 MHz (FM 대역의 중심)
sample_rate = int(20e6)       # 20 MSPS 샘플 레이트
rx_bandwidth = int(20e6)      # 20 MHz RF 대역폭
rx_gain = 60
num_samples = 2**18

sdr.rx_enabled_channels =
sdr.rx0_lo = center_freq
sdr.rx0_sample_rate = sample_rate
sdr.rx0_rf_bandwidth = rx_bandwidth
```

```
sdr.gain_control_mode_chan0 = "manual"
sdr.rx_hardwaregain_chan0 = rx_gain
sdr.rx_buffer_size = num_samples

# 데이터 수신
print("Receiving wideband data from FM broadcast band...")
samples = sdr.rx()
print(f"Received {len(samples)} samples.")

# PSD 계산 및 시각화
fft_size = 4096
psd = np.abs(np.fft.fftshift(np.fft.fft(samples, fft_size)))**2
freqs = np.fft.fftshift(np.fft.fftfreq(fft_size, 1/sample_rate)) + center_freq

plt.figure(figsize=(15, 7))
plt.plot(freqs / 1e6, 10 * np.log10(psd))
plt.title("Wideband PSD of FM Broadcast Band")
plt.xlabel("Frequency (MHz)")
plt.ylabel("Power (dB)")
plt.grid(True)
# 주파수 축 범위를 88MHz ~ 108MHz로 설정
plt.xlim((center_freq - rx_bandwidth/2)/1e6, (center_freq + rx_bandwidth/2)/1e6)
plt.show()
```

이 코드를 실행하면 88 MHz부터 108 MHz까지의 20 MHz 스펙트럼에 대한 PSD 플롯이 생성됩니다. 이 플롯에서 여러 개의 FM 방송국 신호가 각각의 주파수 위치에서 뚜렷한 피크로 나타나는 것을 확인할 수 있습니다. 이 단일 캡처 데이터를 통해 각 방송국의 중심 주파수, 상대적 신호 세기, 점유 대역폭 등을 한눈에 파악할 수 있습니다. 이는 Pluto로는 여러 번 튜닝해야만 가능했던 작업으로, Jupiter SDR의 강력한 스펙트럼 모니터링 능력을 명확히 보여줍니다.

12단계: IQ 샘플링 및 DC 오프셋 보정

pySDR.org의 IQ 샘플링 챕터에서는 SDR의 가장 일반적인 문제 중 하나인 DC 오프셋(DC offset) 또는 DC 스파이크(DC spike)에 대해 설명합니다.[18] 이는 RF 프론트엔드의 불완전성으로 인해 발생하는 현상으로, 스펙트럼의 중심(0 Hz 오프셋)에 원치 않는 강력한 신호 성분이 나타나는 문제입니다.

ADALM-PLUTO와 같은 SDR에서는 이 문제를 해결하기 위해 소프트웨어적으로 DC 블로킹 필터를 적용하거나, 튜닝 주파수를 약간 오프셋하여 원하는 신호를 DC 스파이크 위치에서

벗어나게 하는 기법을 사용합니다. 하지만 Jupiter SDR에 탑재된 ADRV9002는 하드웨어 수준에서 이러한 문제를 자동으로 처리하는 정교한 캘리브레이션 기능을 제공합니다.

pyadi-iio API를 통해 이러한 추적 캘리브레이션(tracking calibration) 기능들을 활성화할 수 있습니다. 대표적인 예는 다음과 같습니다.[15]

- sdr.bbdc_rejection_tracking_en_chan0: 기저대역(Baseband) DC 제거 추적 캘리브레이션을 활성화합니다.
- sdr.rfdc_tracking_en_chan0: RF DC 오프셋 추적 캘리브레이션을 활성화합니다.
- sdr.quadrature_tracking_en_chan0: I/Q 불균형으로 인한 직교 오차(Quadrature error)를 보정하는 캘리브레이션을 활성화합니다.

아래 코드는 특정 신호에 튜닝한 후, DC 오프셋 보정 기능을 켜고 껐을 때의 스펙트럼 변화를 비교하여 하드웨어 캘리브레이션의 효과를 보여줍니다.

Python

```
import adi
import numpy as np
import matplotlib.pyplot as plt

#... (SDR 연결 코드)...
sdr = adi.adrv9002(uri="ip:192.168.2.1")

# 수신 설정
center_freq = int(433.92e6) # ISM 대역
sample_rate = int(1e6)
sdr.rx_enabled_channels =
sdr.rx0_lo = center_freq
sdr.rx0_sample_rate = sample_rate
sdr.rx0_rf_bandwidth = sample_rate
sdr.gain_control_mode_chan0 = "manual"
sdr.rx_hardwaregain_chan0 = 70
sdr.rx_buffer_size = 2**16

# 1. DC 보정 기능 비활성화 상태에서 데이터 수신
sdr.bbdc_rejection_tracking_en_chan0 = False
print("Receiving with DC correction OFF...")
samples_off = sdr.rx()

# 2. DC 보정 기능 활성화 상태에서 데이터 수신
```

```
sdr.bbdc_rejection_tracking_en_chan0 = True
print("Receiving with DC correction ON...")
samples_on = sdr.rx()

# 3. 두 결과의 PSD 비교
def plot_psd(samples, title):
    fft_size = 1024
    psd = np.abs(np.fft.fftshift(np.fft.fft(samples, fft_size)))**2
    freqs = np.fft.fftshift(np.fft.fftfreq(fft_size, 1/sample_rate))
    plt.plot(freqs / 1e3, 10 * np.log10(psd), label=title)

plt.figure(figsize=(15, 7))
plot_psd(samples_off, "DC Correction OFF")
plot_psd(samples_on, "DC Correction ON")
plt.title("Effect of Hardware DC Offset Correction")
plt.xlabel("Frequency Offset (kHz)")
plt.ylabel("Power (dB)")
plt.legend()
plt.grid(True)
plt.show()
```

이 코드를 실행하면 두 개의 PSD 플롯이 겹쳐서 나타납니다. "DC Correction OFF" 플롯에서는 주파수 오프셋 0 kHz 지점에서 뚜렷한 DC 스파이크를 볼 수 있는 반면, "DC Correction ON" 플롯에서는 이 스파이크가 현저하게 감소하거나 사라진 것을 확인할 수 있습니다. 이는 ADRV9002의 고급 하드웨어 기능이 SDR 시스템의 성능을 어떻게 향상시키는지 보여주는 좋은 예입니다.

13단계: 디지털 변조 - BPSK 송신

pySDR.org의 디지털 변조 챕터에서는 정보를 무선으로 전송하기 위한 기본 기술인 BPSK(Binary Phase Shift Keying)를 소개합니다.[19] BPSK는 디지털 비트 '0'과 '1'을 각각 0도와 180도의 위상 변화로 표현하는 방식입니다.

이 실습에서는 Python으로 BPSK 변조된 신호를 생성하고, 이를 Jupiter SDR의 송신 버퍼에 로드하여 공중으로 방사합니다.

1. **BPSK** 신호 생성: numpy를 사용하여 랜덤 비트 시퀀스를 생성하고, 이를 BPSK 심볼(1과 -1)로 매핑합니다. 그런 다음, 각 심볼을 여러 개의 샘플로 확장(upsampling)하여 기저대역 파형을 만듭니다.
2. **Jupiter SDR** 송신 설정: 10단계에서와 같이 송신 채널(TX0)의 중심 주파수, 샘플 레이트, 이득을 설정합니다. 중요한 점은 sdr.tx_cyclic_buffer = True로 설정하여 버퍼에 로드된

BPSK 신호가 계속해서 반복적으로 송신되도록 하는 것입니다.

3. 데이터 송신: 생성된 BPSK 샘플 배열을 sdr.tx() 메서드에 전달합니다. 이 메서드는 샘플 데이터를 Jupiter SDR의 FPGA로 전송하고, FPGA는 이를 DAC를 통해 아날로그 신호로 변환한 후 ADRV9002를 통해 RF 신호로 송신합니다.

Python

```
import adi
import numpy as np
import time

# Jupiter SDR 연결
uri = "ip:192.168.2.1"
sdr = adi.adrv9002(uri=uri)

# BPSK 신호 파라미터
num_symbols = 1000  # 심볼 수
sps = 8             # 심볼 당 샘플 수 (Samples Per Symbol)

# 랜덤 비트 생성 및 BPSK 심볼 매핑 (0 -> 1, 1 -> -1)
bits = np.random.randint(0, 2, num_symbols)
symbols = np.array()
for bit in bits:
    symbols = np.append(symbols, bit*2-1)

# 심볼을 샘플로 확장 (Upsampling)
x = np.array()
for symbol in symbols:
    x = np.append(x, symbol * np.ones(sps))

# 송신 파형 생성 (진폭 조절)
# DAC는 보통 정수형 데이터를 사용하므로, 2^15 스케일링 후 타입 변환
tx_samples = x * 2**14
tx_samples = tx_samples.astype(np.int16)

# 송신 설정
center_freq = int(1.2e9)
sample_rate = int(1e6)
tx_gain = -20
```

```
sdr.tx_enabled_channels =
sdr.tx0_lo = center_freq
sdr.tx0_sample_rate = sample_rate
sdr.tx_hardwaregain_chan0 = tx_gain
sdr.tx_cyclic_buffer = True

print("Transmitting BPSK signal...")
try:
    sdr.tx(tx_samples) # 버퍼에 데이터 전송 및 송신 시작
    print("Transmission started. Press Ctrl+C to stop.")
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    print("\nStopping transmission.")
    sdr.tx_destroy_buffer()
```

이 코드를 실행하면, 다른 SDR 수신기로 1.2 GHz 주파수를 모니터링했을 때, 약 125 kHz (샘플 레이트 / sps = 1e6 / 8) 대역폭을 가진 BPSK 신호의 스펙트럼을 관찰할 수 있습니다.

14단계**:** 디지털 변조 **- BPSK** 수신 및 복조

이제 13단계에서 송신한 BPSK 신호를 수신하고 원래의 비트스트림으로 복원하는 과정을 구현합니다. 이를 위해 두 번째 SDR 장비(또는 Jupiter SDR의 두 번째 채널)를 수신기로 사용합니다. 이 실습은 무선 통신 시스템의 가장 기본적인 수신 체인을 Python으로 구현하는 경험을 제공합니다.

수신 및 복조 과정은 다음과 같은 단계로 이루어집니다.

1. 신호 수신: 수신기는 송신기와 동일한 중심 주파수 및 샘플 레이트로 설정되어야 합니다. sdr.rx()를 통해 IQ 샘플을 수신합니다.
2. 동기화: 실제 시스템에서는 주파수, 위상, 타이밍 동기화 과정이 필요하지만, 이 간단한 예제에서는 송신기와 수신기가 이상적으로 동기화되었다고 가정합니다.
3. 복조: BPSK의 경우, 수신된 심볼의 위상을 확인하는 것으로 복조가 가능합니다. IQ 평면에서 실수부(I 채널)의 부호가 양수이면 '0', 음수이면 '1'로 판단할 수 있습니다.
4. 다운샘플링: 수신된 신호는 심볼 당 여러 개의 샘플(sps)을 포함하므로, 각 심볼의 중앙에 위치한 샘플 하나만을 선택하여(다운샘플링) 최종 비트를 결정합니다.

Python

```python
import adi
import numpy as np
import matplotlib.pyplot as plt

# 수신용 SDR 연결 (Jupiter 또는 Pluto)
# 송신기와 다른 장비를 사용하거나, Jupiter의 다른 채널을 사용
rx_sdr = adi.adrv9002(uri="ip:192.168.2.1") # 예시로 동일 장비 사용

# 수신 설정 (송신기와 동일하게)
center_freq = int(1.2e9)
sample_rate = int(1e6)
sps = 8
num_samples_to_receive = 10000

rx_sdr.rx_enabled_channels =  # RX1 채널 사용
rx_sdr.rx1_lo = center_freq
rx_sdr.rx1_sample_rate = sample_rate
rx_sdr.rx1_rf_bandwidth = sample_rate
rx_sdr.gain_control_mode_chan1 = "slow_attack" # AGC 사용
rx_sdr.rx_buffer_size = num_samples_to_receive

print("Receiving BPSK signal...")
rx_samples = rx_sdr.rx()

# IQ Constellation Plot
plt.figure(figsize=(8, 8))
plt.scatter(np.real(rx_samples), np.imag(rx_samples), alpha=0.3)
plt.title("Received BPSK Constellation")
plt.xlabel("I")
plt.ylabel("Q")
plt.grid(True)
plt.axis('equal')
plt.show()

# 간단한 복조
# 각 심볼의 중앙 샘플 선택
demod_symbols = rx_samples[sps//2::sps]
# 실수부의 부호로 비트 결정
received_bits = (np.real(demod_symbols) < 0).astype(int)

print("First 50 received bits:")
```

```
print(received_bits[:50])
```

실행 결과로 나타나는 IQ 성상도(Constellation) 플롯은 수신된 샘플들이 IQ 평면의 양의 실수축과 음의 실수축 주변에 두 개의 군집을 형성하는 것을 보여줍니다. 이는 BPSK 신호가 성공적으로 수신되었음을 시각적으로 증명합니다. 또한, 복조된 비트 시퀀스를 출력하여 원래 데이터가 복원되었음을 확인할 수 있습니다. (송신 데이터와 정확히 일치하는지 확인하려면 송신 측에서 랜덤 비트 대신 고정된 패턴을 사용해야 합니다.)

15단계: 고급 변조 - QPSK/16-QAM 송신

BPSK를 마스터했다면, 더 높은 데이터 전송률을 위해 더 복잡한 변조 방식인 QPSK(Quadrature Phase Shift Keying)와 16-QAM(Quadrature Amplitude Modulation)으로 확장할 수 있습니다.[19]

- **QPSK**: 4개의 위상(45, 135, 225, 315도)을 사용하여 한 번에 2비트의 정보를 전송합니다.
- **16-QAM**: 16개의 서로 다른 위상과 진폭의 조합을 사용하여 한 번에 4비트의 정보를 전송합니다.

이러한 고차 변조 방식은 각 심볼 포인트 간의 거리가 가깝기 때문에 신호 품질에 더 민감합니다. 특히 송신기의 선형성(linearity)이 매우 중요합니다. 송신기가 비선형적이면 심볼 포인트들이 왜곡되어(intermodulation distortion) 수신 측에서 올바르게 복조하기 어려워집니다. Jupiter SDR의 ADRV9002는 고품질의 RF 선형성 성능을 제공하여 이러한 고차 변조 신호를 깨끗하게 송신하는 데 강점을 가집니다.[2]

아래 코드는 16-QAM 신호를 생성하고 송신하는 예제입니다.

Python

```
import adi
import numpy as np
import time

#... (SDR 연결 코드)...
sdr = adi.adrv9002(uri="ip:192.168.2.1")

# 16-QAM 파라미터
num_symbols = 2048
sps = 8
M = 16 # 16-QAM
```

```
bits_per_symbol = int(np.log2(M))

# 16-QAM 심볼 매핑 테이블 (Gray coding)
qam_map = {
    (0,0,0,0) : -3+3j, (0,0,0,1) : -3+1j, (0,0,1,1) : -3-1j, (0,0,1,0) : -3-3j,
    (0,1,0,0) : -1+3j, (0,1,0,1) : -1+1j, (0,1,1,1) : -1-1j, (0,1,1,0) : -1-3j,
    (1,1,0,0) :  1+3j, (1,1,0,1) :  1+1j, (1,1,1,1) :  1-1j, (1,1,1,0) :  1-3j,
    (1,0,0,0) :  3+3j, (1,0,0,1) :  3+1j, (1,0,1,1) :  3-1j, (1,0,1,0) :  3-3j
}

# 랜덤 비트 생성 및 16-QAM 심볼로 변환
bits = np.random.randint(0, 2, num_symbols * bits_per_symbol)
symbols =
for i in range(0, len(bits), bits_per_symbol):
    bit_group = tuple(bits[i:i+bits_per_symbol])
    symbols.append(qam_map[bit_group])
symbols = np.array(symbols)

# Upsampling
x = np.repeat(symbols, sps)

# 정규화 및 스케일링
x /= np.sqrt(np.mean(np.abs(x)**2)) # 평균 전력 1로 정규화
tx_samples = (x * 2**13).astype(np.complex64)

#... (송신 설정 및 sdr.tx() 호출은 BPSK 예제와 동일)...
# 송신 이득은 더 낮게 시작하는 것이 좋습니다 (예: -30 dB)
sdr.tx_hardwaregain_chan0 = -30
#...
sdr.tx(tx_samples)
#...
```

이 신호를 수신하여 IQ 성상도를 그려보면, 16개의 뚜렷한 점 군집이 격자 형태로 나타나는 것을 확인할 수 있습니다. 만약 점들이 퍼져 있거나 왜곡되어 보인다면, 송신 이득을 낮추거나 DPD와 같은 고급 선형화 기술의 필요성을 고려해볼 수 있습니다.

16단계: Python에서의 FIR 필터링

pySDR.org의 필터 챕터에서는 FIR(Finite Impulse Response) 필터가 SDR에서 원치 않는 신호를 제거하고 원하는 신호만 선택하는 데 어떻게 사용되는지 설명합니다.[18] 이 실습에서는

11단계에서처럼 광대역 신호를 수신한 다음, Python의

scipy.signal 라이브러리를 사용하여 특정 채널만 분리해내는 디지털 필터링을 수행합니다.

이 과정은 수신기의 핵심 기능 중 하나인 채널 선택(channel selection)을 소프트웨어적으로 구현하는 것입니다.

1. 광대역 신호 수신: 먼저 여러 신호가 포함된 넓은 대역폭의 IQ 데이터를 수신합니다. 예를 들어, 여러 개의 FM 방송국이 포함된 10 MHz 대역폭의 데이터를 캡처합니다.
2. **FIR** 필터 설계: scipy.signal.firwin 함수를 사용하여 저역 통과 필터(Low-pass filter)를 설계합니다. 이 필터의 차단 주파수(cutoff frequency)는 우리가 분리하고자 하는 단일 FM 채널의 대역폭(약 200 kHz)에 맞춰 설정합니다.
3. 주파수 이동: 분리하려는 채널이 스펙트럼의 중심에 있지 않다면, 먼저 해당 채널이 중심(0 Hz)으로 오도록 수신된 전체 신호를 주파수 이동시켜야 합니다. 이는 수신된 샘플에 복소 지수 함수를 곱하여 수행합니다.
4. 필터링: 주파수 이동된 신호에 설계된 저역 통과 필터를 적용(컨볼루션)합니다. 이렇게 하면 이제 중심에 위치한 원하는 채널의 신호만 남고 나머지 신호들은 모두 제거됩니다.
5. 다운샘플링: 필터링 후 신호의 대역폭이 크게 줄었으므로, 나이퀴스트 이론에 따라 샘플 레이트를 낮출 수 있습니다. 이를 다운샘플링(downsampling) 또는 데시메이션(decimation)이라고 하며, 불필요한 데이터를 줄여 후속 처리의 계산량을 감소시킵니다.

Python

```
import adi
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

#... (11단계와 유사하게 광대역 신호 수신)...
# 예: 98MHz 중심, 10MHz 샘플 레이트로 수신하여 samples 변수에 저장

# 필터링할 목표 채널의 주파수 오프셋 (예: 98.5MHz -> 0.5MHz 오프셋)
target_offset_freq = 0.5e6

# 1. 목표 채널을 0Hz로 이동
t = np.arange(len(samples)) / sample_rate
freq_shift = np.exp(-1j * 2 * np.pi * target_offset_freq * t)
samples_shifted = samples * freq_shift

# 2. 저역 통과 FIR 필터 설계
```

```
num_taps = 101
cutoff_freq = 100e3  # 100 kHz (FM 채널 대역폭)
nyquist_rate = sample_rate / 2
normalized_cutoff = cutoff_freq / nyquist_rate
taps = signal.firwin(num_taps, normalized_cutoff)

# 3. 필터링 수행
samples_filtered = np.convolve(samples_shifted, taps, 'same')

# 4. 다운샘플링
decimation_factor = 10
samples_decimated = samples_filtered[::decimation_factor]
new_sample_rate = sample_rate / decimation_factor

# 결과 비교 시각화
plt.figure(figsize=(15, 10))
# 원본 신호 PSD
plt.subplot(2, 1, 1)
plt.psd(samples, NFFT=1024, Fs=sample_rate, Fc=center_freq)
plt.title("Original Wideband Signal")
# 필터링 및 다운샘플링된 신호 PSD
plt.subplot(2, 1, 2)
plt.psd(samples_decimated, NFFT=1024, Fs=new_sample_rate, Fc=center_freq +
target_offset_freq)
plt.title("Filtered and Decimated Signal (Isolated Channel)")
plt.tight_layout()
plt.show()
```

결과 플롯을 보면, 상단에는 여러 개의 신호가 존재하는 원본 광대역 스펙트럼이, 하단에는 우리가 선택한 단 하나의 채널만 깨끗하게 분리된 스펙트럼이 나타납니다. 이는 소프트웨어를 통해 RF 필터와 동일한 기능을 구현한 것으로, SDR의 유연성을 보여주는 핵심적인 예제입니다.

17단계: 펄스 성형 - 루트 레이즈드 코사인(RRC) 필터 구현

디지털 변조된 신호를 그대로 송신하면, 사각 펄스 형태 때문에 스펙트럼이 넓게 퍼져 인접 채널에 간섭(Inter-Symbol Interference, ISI)을 일으킬 수 있습니다. pySDR.org의 펄스 성형 챕터에서는 이 문제를 해결하기 위해 RRC(Root-Raised Cosine) 필터와 같은 펄스 성형 필터를 사용하는 방법을 설명합니다.[18]

송신 측에서는 RRC 필터를 적용하여 신호의 스펙트럼을 매끄럽게 만들고, 수신 측에서는

동일한 RRC 필터(정합 필터, Matched Filter)를 사용하여 신호 대 잡음비(SNR)를 최대화하면서 원래 신호를 복원합니다. 송신과 수신 필터가 결합되면 ISI가 없는 이상적인 레이즈드 코사인 필터 특성을 갖게 됩니다.

이 실습에서는 15단계에서 생성한 QPSK 신호에 RRC 필터를 적용하여 송신합니다.

Python

```
import numpy as np
from scipy import signal

#... (15단계의 QPSK 심볼 생성 코드 이후)...
# symbols 변수에 QPSK 심볼이 저장되어 있다고 가정

sps = 8

# RRC 필터 설계
num_taps = 101
beta = 0.35  # Roll-off factor
t = np.arange(-num_taps//2, num_taps//2 + 1)
# 시간 0에서의 분모 0 오류 방지
t[t == 0] = 1e-8
h_rrc = (np.sin(np.pi*t/sps*(1-beta)) + 4*beta*t/sps*np.cos(np.pi*t/sps*(1+beta))) /
(np.pi*t/sps*(1-(4*beta*t/sps)**2))
h_rrc[num_taps//2] = (1 - beta + 4*beta/np.pi)

# Upsampling
x_upsampled = np.zeros(len(symbols) * sps, dtype=np.complex64)
x_upsampled[::sps] = symbols

# 펄스 성형 필터링
tx_samples_shaped = np.convolve(x_upsampled, h_rrc, 'same')

#... (정규화, 스케일링 후 sdr.tx()로 송신)...

# 스펙트럼 비교 시각화
plt.figure(figsize=(15, 7))
# 펄스 성형 전 (사각 펄스)
plt.subplot(1, 2, 1)
plt.psd(np.repeat(symbols, sps), NFFT=1024, Fs=sample_rate)
plt.title("PSD before Pulse Shaping (Rectangular Pulses)")
```

```
# 펄스 성형 후
plt.subplot(1, 2, 2)
plt.psd(tx_samples_shaped, NFFT=1024, Fs=sample_rate)
plt.title("PSD after RRC Pulse Shaping")
plt.tight_layout()
plt.show()
```

결과 PSD 플롯을 비교하면, 펄스 성형 전 신호의 스펙트럼은 양 옆으로 넓게 퍼져 있는 반면(sinc 함수 형태), RRC 필터를 적용한 후의 스펙트럼은 대역폭 내에 에너지가 집중되고 대역외 감쇠(out-of-band attenuation)가 훨씬 가파르게 나타나는 것을 볼 수 있습니다. 이는 한정된 주파수 자원을 효율적으로 사용하고 다른 통신 시스템에 대한 간섭을 줄이는 데 필수적인 기술입니다.

18단계: 링크 버짓 실제 분석

pySDR.org의 링크 버짓 챕터는 송신기에서 수신기까지 신호가 전달되는 과정에서의 전력 변화를 계산하는 방법을 다룹니다.[20] 링크 버짓 분석은 무선 시스템을 설계할 때 수신단에서 충분한 SNR을 확보할 수 있는지 예측하는 데 사용됩니다.

링크 버짓의 기본 방정식은 다음과 같습니다:

$$PRx = PTx + GTx - LPath + GRx - LMisc$$

여기서 각 항은 데시벨(dB 또는 dBm) 단위입니다.

- PRx: 수신 전력
- PTx: 송신 전력
- GTx: 송신 안테나 이득
- LPath: 경로 손실 (거리에 따른 신호 감쇠)
- GRx: 수신 안테나 이득
- LMisc: 케이블 손실 등 기타 손실

이 실습에서는 Jupiter SDR을 송신기로, ADALM-PLUTO를 수신기로 사용하여 간단한 링크 버짓을 실제로 측정하고 계산값과 비교해 봅니다.

1. 송신 설정 **(Jupiter)**: 알려진 출력 전력을 설정합니다. tx_hardwaregain_chan0을 특정 값(예: -20 dB)으로 설정합니다. 이 값은 절대적인 dBm 출력이 아니므로, 실제 출력 전력은 교정된 장비(스펙트럼 분석기)로 미리 측정해 두는 것이 이상적입니다. 여기서는 상대적인 비교에 초점을 맞춥니다.
2. 수신 설정 **(Pluto)**: 수신된 신호의 전력을 측정합니다. 수신된 IQ 샘플의 평균 전력은 np.mean(np.abs(rx_samples)**2)로 계산할 수 있습니다.
3. 경로 손실 계산: 자유 공간 경로 손실(FSPL) 공식은 다음과 같습니다.

$$FSPL(dB) = 20\log_{10}(d) + 20\log_{10}(f) + 20\log_{10}(\frac{4\pi}{c}) - G_{Tx} - G_{Rx}$$

여기서 d는 거리(m), f는 주파수(Hz), c는 빛의 속도입니다.

4. 비교: (송신 전력 - 경로 손실) 값이 (측정된 수신 전력) 값과 유사한지 확인합니다.

Python

```
# 이 코드는 개념적인 흐름을 보여줍니다.
# 실제 구현 시에는 두 개의 SDR을 동시에 제어하는 스크립트가 필요합니다.

# --- 송신기 (Jupiter) ---
# sdr_tx.tx_hardwaregain_chan0 = -20 # dB
#... 톤 신호 송신...

# --- 수신기 (Pluto) ---
# sdr_rx.rx_hardwaregain_chan0 = 0 # dB (상대 측정을 위해 고정)
# rx_samples = sdr_rx.rx()
# received_power_linear = np.mean(np.abs(rx_samples)**2)
# received_power_db = 10 * np.log10(received_power_linear)

# --- 링크 버짓 계산 ---
distance = 1.0 # 미터
frequency = 1e9 # 헤르츠
c = 3e8 # 빛의 속도
tx_antenna_gain = 2.15 # dBi (다이폴 안테나 가정)
rx_antenna_gain = 2.15 # dBi

# FSPL 계산
path_loss_db = 20 * np.log10(distance) + 20 * np.log10(frequency) - 147.55

# 예상 수신 전력 (상대값)
# tx_power_db는 교정된 값이라고 가정
# expected_rx_power_db = tx_power_db + tx_antenna_gain - path_loss_db + rx_antenna_gain

# print(f"Measured Relative Rx Power: {received_power_db:.2f} dB")
# print(f"Calculated Path Loss: {path_loss_db:.2f} dB")
```

이 실험을 통해 거리가 멀어질수록 수신 전력이 어떻게 감소하는지, 주파수가 높아질수록 경로 손실이 어떻게 증가하는지를 정량적으로 확인할 수 있습니다. 이는 무선 시스템의 통신 가능 거리를 예측하고 설계하는 데 있어 필수적인 과정입니다.

19단계: 채널 코딩 - 반복 부호 구현

채널 코딩(Channel Coding) 또는 순방향 오류 정정(Forward Error Correction, FEC)은 무선 채널에서 발생하는 비트 오류를 수신 측에서 스스로 감지하고 정정하기 위해 송신 데이터에 의도적으로 중복성(redundancy)을 추가하는 기술입니다.[21]

가장 간단한 채널 코딩 방식은 반복 부호(Repetition Code)입니다. 예를 들어, 부호율(code-rate) 1/3 반복 부호는 '0'을 보내고 싶으면 '000'을, '1'을 보내고 싶으면 '111'을 보냅니다. 수신 측에서는 수신된 세 비트 중 다수결(majority voting)을 통해 원래 비트를 결정합니다. 예를 들어 '001'을 받으면 '0'으로, '101'을 받으면 '1'로 판단합니다. 이 방식은 하나의 비트 오류를 정정할 수 있습니다.

이 실습에서는 13단계의 BPSK 송수신 시스템에 1/3 반복 부호를 적용하여, 잡음이 많은 환경에서 통신 성능이 향상되는 것을 확인합니다.

Python

```
# --- 송신기 ---
# 원본 비트
original_bits = np.random.randint(0, 2, 100)

# 1/3 반복 부호화
coded_bits = np.repeat(original_bits, 3)

#... (coded_bits를 BPSK 심볼로 변환하여 송신)...

# --- 수신기 ---
#... (BPSK 신호 수신 및 복조하여 received_coded_bits 획득)...

# 다수결 복호화
decoded_bits =
for i in range(0, len(received_coded_bits), 3):
    three_bits = received_coded_bits[i:i+3]
    # 1의 개수가 2개 이상이면 1, 아니면 0
    if np.sum(three_bits) >= 2:
        decoded_bits.append(1)
    else:
        decoded_bits.append(0)
decoded_bits = np.array(decoded_bits)
```

```
# BER 계산
# num_errors = np.sum(original_bits!= decoded_bits)
# ber = num_errors / len(original_bits)
# print(f"Bit Error Rate (BER) with Repetition Code: {ber}")
```

동일한 SNR 환경에서 채널 코딩을 적용하지 않았을 때와 적용했을 때의 비트 오류율(Bit Error Rate, BER)을 비교하면, 반복 부호를 사용했을 때 BER이 현저하게 낮아지는 것을 확인할 수 있습니다. 이는 데이터 전송률을 1/3로 희생하는 대신 통신의 신뢰성을 높인 결과입니다. 실제 시스템에서는 터보 부호(Turbo codes)나 LDPC(Low-Density Parity-Check) 부호와 같은 훨씬 더 효율적인 채널 코딩 기법이 사용됩니다.[21]

20단계**:** 동기화 **-** 조악한 주파수 오프셋 보정

실제 무선 환경에서 수신기는 송신기와 완벽하게 동일한 주파수를 가질 수 없습니다. 두 장치의 발진기(oscillator) 부정확성 때문에 약간의 주파수 오프셋(frequency offset)이 발생하며, 이는 수신된 신호의 성상도를 회전시켜 복조를 불가능하게 만듭니다. 따라서 수신기는 이 주파수 오프셋을 추정하고 보정하는 동기화(synchronization) 과정을 거쳐야 합니다.[22]

동기화는 보통 조악한(coarse) 보정과 정밀한(fine) 보정의 두 단계로 나뉩니다. 조악한 주파수 보정은 보통 FFT를 사용하여 큰 주파수 오프셋을 찾아내는 방식입니다. 신호의 스펙트럼을 FFT로 분석했을 때, 가장 큰 전력을 가진 주파수 빈(bin)이 바로 주파수 오프셋에 해당할 가능성이 높습니다.

pySDR.org에서 제안하는 방법 중 하나는 수신 신호를 제곱(또는 네제곱)하여 변조 성분을 제거하고 반송파(carrier) 성분만 남긴 뒤, FFT를 통해 이 반송파의 주파수를 찾는 것입니다.[22]

Python

```
import numpy as np
import matplotlib.pyplot as plt

#... (신호 수신 후, rx_samples 변수에 저장)...
# 의도적으로 주파수 오프셋 시뮬레이션
fs = 1e6
offset = 50e3 # 50 kHz 오프셋
t = np.arange(len(rx_samples)) / fs
```

```
rx_samples_with_offset = rx_samples * np.exp(1j * 2 * np.pi * offset * t)

# 조악한 주파수 오프셋 추정 (FFT-based)
N = len(rx_samples_with_offset)
# BPSK의 경우 제곱하면 반송파 주파수가 2배가 됨
psd = np.abs(np.fft.fft(rx_samples_with_offset**2))
freqs = np.fft.fftfreq(N, 1/fs)

# 최대 전력 주파수 찾기
max_freq_index = np.argmax(psd)
estimated_offset = freqs[max_freq_index] / 2 # 2배가 되었으므로 2로 나눔

print(f"Estimated Coarse Frequency Offset: {estimated_offset / 1e3:.2f} kHz")

# 주파수 오프셋 보정
correction_signal = np.exp(-1j * 2 * np.pi * estimated_offset * t)
corrected_samples = rx_samples_with_offset * correction_signal

# 보정 전후 성상도 비교
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(np.real(rx_samples_with_offset), np.imag(rx_samples_with_offset), alpha=0.1)
plt.title("Constellation Before Correction")
plt.axis('equal')
plt.grid(True)
plt.subplot(1, 2, 2)
plt.scatter(np.real(corrected_samples), np.imag(corrected_samples), alpha=0.1)
plt.title("Constellation After Coarse Correction")
plt.axis('equal')
plt.grid(True)
plt.show()
```

결과 플롯을 보면, 보정 전에는 성상도가 회전하는 모습(링 형태)을 보이지만, 조악한 주파수 보정을 거친 후에는 회전이 멈추고 BPSK의 두 지점에 샘플들이 모이는 것을 확인할 수 있습니다. 아직 약간의 잔여 위상 오프셋이 남아있을 수 있으며, 이는 다음 단계인 정밀한 동기화 과정에서 보정됩니다.

21단계**:** 동기화 **-** 정밀한 주파수 및 위상 오프셋 보정

조악한 주파수 보정 후에도 남아있는 작은 주파수 오프셋과 위상 오프셋을 보정하기 위해 위상

고정 루프(Phase-Locked Loop, PLL)와 같은 피드백 기반의 정밀한 동기화 기법이 사용됩니다.[22]

디지털 통신에서는 코스타스 루프(Costas Loop)가 BPSK나 QPSK 신호의 반송파를 복원하는 데 널리 사용됩니다. 코스타스 루프는 위상 오차를 측정하고, 이 오차를 기반으로 수신 신호에 곱해주는 복소 지수 신호를 계속해서 조정하여 최종적으로 위상 오차를 0으로 수렴시키는 피드백 제어 시스템입니다.

Python으로 코스타스 루프를 구현하는 것은 복잡할 수 있지만, 그 원리는 다음과 같습니다.

1. 위상 오차 검출: 현재 수신된 심볼과 가장 가까운 이상적인 성상도 점 사이의 위상 차이를 계산합니다. BPSK의 경우, real(symbol) * imag(symbol)과 같은 간단한 계산으로 위상 오차에 비례하는 값을 얻을 수 있습니다.
2. 루프 필터: 측정된 위상 오차 값은 노이즈가 많으므로, 루프 필터(보통 비례-적분(PI) 제어기)를 통과시켜 안정적인 제어 신호를 만듭니다.
3. 수치 제어 발진기**(NCO)**: 루프 필터의 출력을 기반으로 다음 샘플에 적용할 위상 보정량을 계산합니다. 이는 수신 신호에 곱해질 복소 지수 신호의 위상을 조정하는 역할을 합니다.

이 과정을 샘플 단위로 반복하면, 루프는 점차 송신기의 반송파 위상에 고정(lock)되고, 그 결과로 수신된 성상도는 더 이상 회전하지 않고 안정적으로 유지됩니다.

22단계**:** 프레임 동기화

주파수와 위상 동기화가 완료되면, 수신기는 이제 연속적인 비트 스트림을 얻게 됩니다. 하지만 이 스트림에서 어디가 데이터 패킷의 시작이고 끝인지 알 수 없습니다. 프레임 동기화(Frame Synchronization)는 이 패킷의 경계를 찾는 과정입니다.[22]

가장 일반적인 방법은 데이터 패킷의 시작 부분에 모든 수신기가 미리 알고 있는 특별한 비트 시퀀스, 즉 프리앰블(preamble) 또는 동기 워드(sync word)를 삽입하는 것입니다. 수신기는 들어오는 비트 스트림을 지속적으로 모니터링하다가, 이 프리앰블과 일치하는 패턴을 발견하면 그 지점을 패킷의 시작으로 간주합니다.

이 과정은 상관(correlation) 연산을 통해 구현됩니다. 수신기는 수신된 샘플의 이동 윈도우(sliding window)와 프리앰블 시퀀스 간의 상관 값을 계속해서 계산합니다. 두 시퀀스가 정확히 일치하는 지점에서 상관 값은 최대 피크를 나타내게 되며, 이 피크를 검출함으로써 프레임의 시작을 알 수 있습니다.

Python

```
import numpy as np

#... (송신 측에서 데이터 앞에 프리앰블 추가)...
preamble = np.array([1, 1, 1, -1, -1, -1, 1, -1, -1, 1, -1]) # Barker-11 코드
# tx_symbols = np.concatenate((preamble, data_symbols))
#...

# --- 수신기 ---
#... (수신된 샘플이 rx_samples에 저장)...

# 상관 연산을 통한 프리앰블 검출
correlation = np.abs(np.correlate(rx_samples, preamble, mode='valid'))

# 상관 피크 찾기
peak_index = np.argmax(correlation)
peak_value = correlation[peak_index]

# 문턱값(threshold)을 이용한 검출
threshold = 0.8 * np.max(np.abs(preamble)**2) * len(preamble)
if peak_value > threshold:
    print(f"Preamble detected at sample index: {peak_index}")
    # 데이터 시작 위치
    data_start_index = peak_index + len(preamble)
    # received_data = rx_samples[data_start_index:]
else:
    print("Preamble not detected.")

plt.plot(correlation)
plt.title("Correlation with Preamble")
plt.xlabel("Sample Index")
plt.ylabel("Correlation Magnitude")
plt.grid(True)
plt.show()
```

상관 결과 플롯에서 뚜렷한 피크가 나타나는 것을 통해, 수신 데이터 스트림 내에서 패킷의 정확한 시작 위치를 성공적으로 찾아냈음을 확인할 수 있습니다.

23단계**:** 완전한 **FDD** 송수신기 구축 **- 1**부 **(**소프트웨어**)**

이제 15단계(QPSK 변조), 17단계(RRC 펄스 성형), 22단계(프레임 동기화)에서 구현한 기능들을 하나로 통합하여, 완전한 통신 패킷을 생성하는 Python 스크립트를 작성합니다. 이 스크립트는 실제 무선 통신 시스템에서 사용되는 송신 체인의 소프트웨어 부분을 구현하는 것입니다.

송신 패킷의 구조는 일반적으로 다음과 같습니다.
[프리앰블] - [헤더] - [페이로드] -

- 프리앰블: 프레임 동기화를 위한 알려진 시퀀스.
- 헤더: 페이로드의 길이, 데이터 타입 등 패킷에 대한 정보를 담고 있음.
- 페이로드: 실제 전송하려는 데이터 (예: 텍스트 메시지, 이미지 파일).
- **CRC (Cyclic Redundancy Check)**: 전송 중 발생한 오류를 검출하기 위한 체크섬.

이 실습에서는 프리앰블과 페이로드로 구성된 간단한 패킷을 생성합니다.

Python

```
import numpy as np

def generate_tx_packet(payload_bits, sps=8):
    """
    페이로드 비트를 받아 완전한 QPSK 송신 파형을 생성합니다.
    """
    # 1. 프리앰블 생성 (BPSK)
    preamble_symbols = np.array([1, 1, 1, -1, -1, 1, -1, 1])

    # 2. 페이로드 비트를 QPSK 심볼로 변환
    payload_symbols =
    for i in range(0, len(payload_bits), 2):
        I = 1 - 2 * payload_bits[i]
        Q = 1 - 2 * payload_bits[i+1]
        payload_symbols.append((I + 1j*Q) / np.sqrt(2))
    payload_symbols = np.array(payload_symbols)

    # 3. 프리앰블과 페이로드 결합
    all_symbols = np.concatenate((preamble_symbols, payload_symbols))

    # 4. RRC 펄스 성형
    num_taps = 101
    beta = 0.35
    #... (17단계의 RRC 필터 h_rrc 생성 코드)...
    t = np.arange(-num_taps//2, num_taps//2 + 1)
```

```
    t[t == 0] = 1e-8
    h_rrc = (np.sin(np.pi*t/sps*(1-beta)) + 4*beta*t/sps*np.cos(np.pi*t/sps*(1+beta))) /
(np.pi*t/sps*(1-(4*beta*t/sps)**2))
    h_rrc[num_taps//2] = (1 - beta + 4*beta/np.pi)

    x_upsampled = np.zeros(len(all_symbols) * sps, dtype=np.complex64)
    x_upsampled[::sps] = all_symbols
    tx_waveform = np.convolve(x_upsampled, h_rrc, 'same')

    # 5. 정규화 및 스케일링
    tx_waveform /= np.max(np.abs(tx_waveform))
    tx_waveform = (tx_waveform * 2**14).astype(np.complex64)

    return tx_waveform

# --- 사용 예시 ---
message = "Hello Jupiter SDR!"
# 메시지를 비트로 변환
payload_bits = np.unpackbits(np.array([ord(c) for c in message], dtype=np.uint8))

# 송신 파형 생성
tx_packet = generate_tx_packet(payload_bits)

#... (이 tx_packet을 sdr.tx()를 통해 송신)...
```

이 generate_tx_packet 함수는 텍스트 메시지를 입력받아, 이를 비트로 변환하고, QPSK 변조, 프리앰블 추가, RRC 펄스 성형을 거쳐 최종적으로 Jupiter SDR로 전송할 수 있는 IQ 파형을 생성합니다.

24단계: 완전한 FDD 송수신기 구축 - 2부 (하드웨어)

소프트웨어에서 송수신 파형을 처리할 준비가 되었다면, 이제 하드웨어를 동시에 송신하고 수신할 수 있도록 설정해야 합니다. 이를 FDD(Frequency Division Duplex) 방식이라고 하며, 송신과 수신에 서로 다른 주파수를 사용하여 전이중(full-duplex) 통신을 구현합니다.

Jupiter SDR의 ADRV9002는 2개의 송신 채널과 2개의 수신 채널을 가지고 있어 FDD 구현에 이상적입니다. pyadi-iio를 사용하여 TX0 채널은 송신용으로, RX1 채널은 수신용으로 동시에 설정할 수 있습니다. (동일한 안테나를 송수신에 함께 사용하려면 듀플렉서(duplexer)라는 추가적인 RF 부품이 필요하지만, 이 실습에서는 송신과 수신에 각각 별도의 안테나를

사용한다고 가정합니다.)

Python

```
import adi

#... (SDR 연결)...
sdr = adi.adrv9002(uri="ip:192.168.2.1")

# --- FDD 하드웨어 설정 ---
# 송신 채널 (TX0) 설정
tx_freq = int(2.41e9)
tx_sample_rate = int(2e6)
tx_gain = -15

sdr.tx_enabled_channels =
sdr.tx0_lo = tx_freq
sdr.tx0_sample_rate = tx_sample_rate
sdr.tx_hardwaregain_chan0 = tx_gain
sdr.tx_cyclic_buffer = False # 단일 패킷 송신

# 수신 채널 (RX1) 설정
rx_freq = int(2.45e9)
rx_sample_rate = int(2e6)
rx_gain = 50

sdr.rx_enabled_channels =
sdr.rx1_lo = rx_freq
sdr.rx1_sample_rate = rx_sample_rate
sdr.rx1_rf_bandwidth = rx_sample_rate
sdr.gain_control_mode_chan1 = "manual"
sdr.rx_hardwaregain_chan1 = rx_gain
sdr.rx_buffer_size = 2**16

print("Jupiter SDR configured for FDD operation.")
print(f"TX0 @ {sdr.tx0_lo / 1e6} MHz")
print(f"RX1 @ {sdr.rx1_lo / 1e6} MHz")
```

이 코드는 Jupiter SDR의 두 채널을 서로 다른 주파수로 독립적으로 설정합니다. 이제 sdr.tx(tx_packet)를 호출하여 송신을 시작하는 동시에, 별도의 스레드나 루프에서 rx_samples =

sdr.rx()를 호출하여 수신을 수행할 수 있습니다. 이는 ADALM-PLUTO에서는 구현하기 어려웠던 진정한 동시 송수신 기능입니다.

25단계: 완전한 FDD 송수신기 구축 - 3부 (종단 간 테스트)

이제 소프트웨어와 하드웨어가 모두 준비되었으므로, 23단계와 24단계의 코드를 결합하여 완전한 종단 간(end-to-end) 테스트를 수행합니다. 이 테스트의 목표는 한쪽 Jupiter SDR(또는 다른 SDR)에서 생성한 패킷을 송신하고, 다른 쪽에서 이를 수신하여 원래 데이터를 성공적으로 복원하는지 확인하는 것입니다.

이 테스트를 통해 시스템의 전체 성능을 정량적으로 평가할 수 있는 지표인 BER(Bit Error Rate)을 계산합니다.

전체적인 테스트 흐름은 다음과 같습니다.

1. 송신기:
    - 알려진 데이터 페이로드(예: 의사 난수 비트 시퀀스)를 생성합니다.
    - generate_tx_packet 함수를 사용하여 완전한 송신 파형을 만듭니다.
    - Jupiter SDR의 TX0 채널을 통해 이 파형을 송신합니다.
2. 수신기:
    - Jupiter SDR의 RX1 채널(또는 다른 SDR)을 통해 신호를 수신합니다.
    - 수신된 샘플에서 프리앰블을 찾아 프레임 동기화를 수행합니다 (22단계).
    - 주파수 및 위상 동기화를 수행합니다 (20, 21단계).
    - RRC 정합 필터링을 수행합니다.
    - QPSK 심볼을 비트로 복조합니다.
3. **BER** 계산:
    - 수신 측에서 복원된 비트 시퀀스를 송신 측의 원본 비트 시퀀스와 비교합니다.
    - 서로 다른 비트의 수를 전체 비트 수로 나누어 BER을 계산합니다.

이 종단 간 테스트는 지금까지 배운 모든 pySDR 개념(변조, 펄스 성형, 동기화 등)을 종합적으로 활용하는 중요한 실습입니다. 성공적으로 낮은 BER(예: 10−3 이하)을 달성했다면, Jupiter SDR을 사용하여 기본적인 디지털 통신 시스템을 성공적으로 구현한 것입니다. 이는 더 복잡한 통신 프로토콜(예: WiFi, LTE)을 분석하고 구현하기 위한 견고한 기반이 됩니다.

Part III: 고급 기능: ADRV9002 및 MPSoC 활용 (26-35단계)

이 파트에서는 ADALM-PLUTO의 능력을 훨씬 뛰어넘는 Jupiter SDR만의 고유한 고성능 기능들을 탐구합니다. ADRV9002 트랜시버의 고급 기능과 Zynq MPSoC의 강력한 처리 능력을 활용하는 방법을 배웁니다.

26단계: ADRV9002 프로파일 이해 및 사용

ADALM-PLUTO에서는 중심 주파수, 샘플 레이트, 이득 등 모든 파라미터를 개별적으로 설정하는 데 익숙할 것입니다. 그러나 ADRV9002와 같은 고성능 트랜시버는 훨씬 더 복잡한 내부 설정(필터 구성, 캘리브레이션 알고리즘, 전력 모드 등)을 가지고 있습니다. 이러한 모든 설정을 수동으로 관리하는 것은 매우 어렵고 오류가 발생하기 쉽습니다.

이 문제를 해결하기 위해 ADRV9002는 프로파일(profile) 기반의 구성 시스템을 도입했습니다. 프로파일은 특정 애플리케이션(예: 20MHz 대역폭의 LTE FDD 통신)에 최적화된 수백 개의 파라미터 설정을 담고 있는 하나의 파일입니다. 사용자는 ADI에서 제공하는 TES(Transceiver Evaluation Software) GUI를 사용하여 자신의 요구사항에 맞는 프로파일을 생성할 수 있습니다.

pyadi-iio에서는 이 프로파일을 장치에 쉽게 로드할 수 있는 방법을 제공합니다. 이 접근 방식은 두 단계의 구성 패러다임을 만듭니다.

1. 프로파일 로드: sdr.write_profile() 메서드를 사용하여 프로파일 파일(.json 형식)을 Jupiter SDR에 로드합니다. 이는 트랜시버의 근본적인 동작 모드(샘플 레이트, 대역폭, 필터 체인 등)를 결정하는 정적인 설정 단계입니다.
2. 동적 파라미터 조정: 프로파일이 로드된 후, sdr.rx0_lo나 sdr.rx_hardwaregain_chan0과 같은 동적 파라미터를 Python 코드에서 실시간으로 조정합니다. 이 파라미터들은 프로파일이 설정한 제약 조건 내에서 작동합니다.

이러한 2단계 구성 방식은 미션 크리티컬 애플리케이션에서 매우 중요합니다.[4] 프로파일을 사용함으로써 트랜시버가 항상 검증되고 안정적인 상태에서 동작함을 보장할 수 있기 때문입니다. 이는 제한이 아니라, 신뢰성 있고 반복 가능한 시스템 동작을 만들기 위한 강력한 기능입니다.

Python

```
import adi

#... (SDR 연결)...
sdr = adi.adrv9002(uri="ip:192.168.2.1")
```

```
# 프로파일 파일 경로 (이 파일은 TES에서 생성하거나 ADI에서 제공받아야 함)
profile_path = "my_adrv9002_profile.json"

try:
    # 프로파일을 문자열로 읽어와서 장치에 쓰기
    with open(profile_path, 'r') as f:
        profile_content = f.read()

    print(f"Writing profile from {profile_path}...")
    sdr.write_profile(profile_content)
    print("Profile successfully loaded.")

    # 프로파일이 로드된 후, 샘플 레이트와 같은 속성은 읽기 전용이 될 수 있습니다.
    # 프로파일에 정의된 값을 확인합니다.
    print(f"RX0 Sample Rate (from profile): {sdr.rx0_sample_rate / 1e6} MSPS")
    print(f"RX0 RF Bandwidth (from profile): {sdr.rx0_rf_bandwidth / 1e6} MHz")

    # LO 주파수와 같은 동적 파라미터는 여전히 변경 가능합니다.
    sdr.rx0_lo = int(2.4e9)
    print(f"RX0 LO set to: {sdr.rx0_lo / 1e6} MHz")

except Exception as e:
    print(f"Error loading profile: {e}")
```

27단계**:** 동기화된 듀얼 채널 수신

Jupiter SDR의 핵심 기능 중 하나는 두 개의 수신 채널에서 위상 동기화된(phase-coherent) 데이터를 동시에 수신하는 능력입니다. 이는 두 채널이 동일한 클럭 소스를 공유하고 샘플링 시점이 정확하게 정렬되어 있음을 의미합니다. 이 기능은 다중 안테나 시스템을 구현하는 데 필수적이며, 수신 다이버시티, 방향 탐지(Direction Finding), 빔포밍(Beamforming)과 같은 고급 기술의 기반이 됩니다.

pyadi-iio를 사용하여 듀얼 채널 수신을 설정하는 것은 간단합니다. rx_enabled_channels 속성에 원하는 채널 번호(0과 1)를 리스트 형태로 지정하기만 하면 됩니다. sdr.rx() 메서드를 호출하면, 이 리스트에 지정된 각 채널의 IQ 데이터가 NumPy 배열의 리스트 형태로 반환됩니다.

아래 코드는 동일한 신호 소스를 RF 스플리터를 사용하여 RX0와 RX1 포트에 동시에 입력하고, 두 채널에서 수신된 데이터가 실제로 동기화되어 있는지 확인하는 예제입니다.

Python

```python
import adi
import numpy as np
import matplotlib.pyplot as plt

#... (SDR 연결)...
sdr = adi.adrv9002(uri="ip:192.168.2.1")

# 듀얼 채널 수신 설정
sdr.rx_enabled_channels =  # RX0와 RX1 모두 활성화
sdr.rx0_lo = int(1e9)
sdr.rx1_lo = int(1e9) # 두 채널을 동일한 주파수로 설정
sdr.rx0_sample_rate = int(5e6)
sdr.rx1_sample_rate = int(5e6)
#... (이득 등 기타 파라미터 설정)...
sdr.rx_buffer_size = 2**14

print("Receiving from both RX0 and RX1 channels simultaneously...")
# sdr.rx()는 [rx0_data, rx1_data] 형태의 리스트를 반환
received_data = sdr.rx()
rx0_samples = received_data
rx1_samples = received_data

print(f"Received {len(rx0_samples)} samples from RX0")
print(f"Received {len(rx1_samples)} samples from RX1")

# 두 채널 간의 위상 차이 계산 및 시각화
# 이상적으로 위상 차이는 시간에 따라 일정해야 함
phase_diff = np.angle(rx0_samples * np.conj(rx1_samples))

plt.figure(figsize=(15, 7))
plt.plot(np.unwrap(phase_diff))
plt.title("Phase Difference between RX0 and RX1")
plt.xlabel("Sample Index")
plt.ylabel("Phase Difference (Radians)")
plt.ylim(-np.pi, np.pi)
plt.grid(True)
plt.show()
```

실행 결과로 나타나는 위상 차이 플롯이 시간에 따라 거의 일정한 값을 유지한다면(약간의 노이즈는 있을 수 있음), 두 채널이 성공적으로 위상 동기화되었음을 의미합니다. 이 일정한 위상 차이는 두 포트까지의 케이블 길이 차이 등에서 비롯된 것입니다. 이 동기화 기능은 다음 단계에서 다룰 수신 다이버시티와 같은 실제 애플리케이션의 핵심입니다. 커뮤니티 포럼에서는 이와 관련된 일반적인 문제들과 해결책에 대한 논의를 찾아볼 수 있습니다.[23]

28단계: 애플리케이션 - 수신 다이버시티 (최대비 결합)

수신 다이버시티(Receive Diversity)는 둘 이상의 안테나를 사용하여 동일한 신호를 수신하고, 이들을 지능적으로 결합하여 페이딩(fading) 현상을 극복하고 SNR을 향상시키는 기술입니다. 가장 널리 사용되는 결합 기법 중 하나가 최대비 결합(Maximal-Ratio Combining, MRC)입니다.

MRC의 원리는 각 안테나에서 수신된 신호의 위상을 정렬한 후, 신호 품질(SNR)이 좋은 채널에 더 높은 가중치를 부여하여 더하는 것입니다. 간단한 구현에서는 각 채널의 신호 세기에 비례하는 가중치를 사용합니다.

27단계에서 얻은 동기화된 듀얼 채널 데이터를 사용하여 간단한 MRC 알고리즘을 Python으로 구현할 수 있습니다.

1. 위상 보정: 두 채널 간의 평균 위상 차이를 계산하여 한쪽 채널의 위상을 회전시켜 두 신호의 위상을 정렬합니다.
2. 가중치 계산: 각 채널의 평균 신호 전력을 계산하여 가중치로 사용합니다.
3. 결합: 위상이 보정된 각 채널의 신호에 해당 가중치를 곱한 후 더하여 최종 결합된 신호를 얻습니다.

Python

```
#... (27단계에서 rx0_samples, rx1_samples 획득 후)...

# 1. 위상 보정
avg_phase_diff = np.mean(np.angle(rx0_samples * np.conj(rx1_samples)))
rx1_corrected = rx1_samples * np.exp(1j * avg_phase_diff)

# 2. 가중치 계산 (신호 진폭에 비례)
w0 = np.mean(np.abs(rx0_samples))
w1 = np.mean(np.abs(rx1_corrected))

# 3. 결합
```

```
combined_samples = (w0 * rx0_samples + w1 * rx1_corrected) / (w0**2 + w1**2)

# SNR 비교 (개념적)
# 노이즈 전력이 동일하다고 가정할 때, 신호 전력 비교
snr_rx0 = 10 * np.log10(np.mean(np.abs(rx0_samples)**2))
snr_combined = 10 * np.log10(np.mean(np.abs(combined_samples)**2))

print(f"Relative SNR of RX0: {snr_rx0:.2f} dB")
print(f"Relative SNR after MRC: {snr_combined:.2f} dB")
print(f"SNR Improvement: {snr_combined - snr_rx0:.2f} dB")
```

이론적으로, 두 채널의 노이즈가 독립적일 때 MRC는 최대 3 dB의 SNR 향상을 가져올 수 있습니다. 이 실습은 다중 안테나 기술이 무선 통신의 신뢰성을 어떻게 향상시키는지에 대한 직관적인 이해를 제공하며, 이는 MIMO 및 빔포밍과 같은 더 진보된 기술의 기초가 됩니다.

29단계: 방향 탐지 입문 - 위상차 방식

pySDR.org의 빔포밍 및 DOA(Direction-of-Arrival) 챕터에서 다루는 것처럼, 둘 이상의 안테나를 사용하면 신호가 오는 방향을 추정할 수 있습니다.[24] 가장 기본적인 방향 탐지 기법은 두 안테나에 도달하는 신호의 위상 차이를 이용하는 위상 간섭계(phase interferometer) 방식입니다.

RX0와 RX1 포트에 연결된 두 안테나를 알려진 거리 d만큼 떨어뜨려 놓습니다. 평면파가 입사각 θ로 두 안테나에 도달하면, 두 안테나 사이의 경로 길이 차이로 인해 위상 차이 Δϕ가 발생합니다. 이 관계는 다음과 같은 수식으로 표현됩니다.

$$\Delta\phi = \frac{2\pi d \sin(\theta)}{\lambda}$$

여기서 λ는 신호의 파장입니다. 이 식을 θ에 대해 정리하면, 측정된 위상 차이 Δϕ로부터 입사각 θ를 계산할 수 있습니다.

$$\theta = \arcsin\left(\frac{\Delta\phi \cdot \lambda}{2\pi d}\right)$$

이 실습에서는 이 원리를 이용하여 신호원의 방향을 추정합니다.

```
Python
```

```
#... (27단계와 같이 듀얼 채널 데이터 수신)...
```

```
# 단, 이번에는 두 안테나를 물리적으로 분리하여 설치

# 시스템 파라미터
freq = sdr.rx0_lo  # 수신 주파수 (Hz)
c = 3e8            # 빛의 속도 (m/s)
lambda_ = c / freq # 파장 (m)
d = lambda_ / 2    # 안테나 간격 (예: 반파장)

# 평균 위상차 측정
phase_diff_rad = np.mean(np.angle(rx0_samples * np.conj(rx1_samples)))

# 위상 모호성(ambiguity) 처리
# d > lambda/2 이면 2*pi의 배수만큼 위상차가 날 수 있음
# 여기서는 d = lambda/2 이므로 모호성이 없다고 가정

# 입사각 계산
# arcsin의 입력은 -1과 1 사이여야 함
arg = (phase_diff_rad * lambda_) / (2 * np.pi * d)
if -1 <= arg <= 1:
    theta_rad = np.arcsin(arg)
    theta_deg = np.degrees(theta_rad)
    print(f"Measured Phase Difference: {np.degrees(phase_diff_rad):.2f} degrees")
    print(f"Estimated Angle of Arrival (AoA): {theta_deg:.2f} degrees")
else:
    print("Cannot determine angle, phase difference is out of bounds.")
```

실제 신호원을 안테나 배열 앞에서 움직여 보면서 계산된 AoA 값이 어떻게 변하는지 관찰할 수 있습니다. 이는 레이더, 무선 감시, 스마트 안테나 시스템의 핵심 원리를 직접 구현해보는 흥미로운 실험입니다.

30단계: 고급 이득 제어 모드 탐색

ADALM-PLUTO에서는 'manual', 'slow_attack', 'fast_attack'의 세 가지 자동 이득 제어(AGC) 모드를 사용할 수 있었습니다.[10] Jupiter SDR의 ADRV9002는 이보다 더 정교하고 다양한 이득 제어 옵션을 제공합니다.

gain_control_mode_chan0 속성을 통해 이러한 모드들을 설정할 수 있습니다.[15]

- **manual**: rx_hardwaregain_chan0 속성을 통해 이득을 수동으로 직접 제어합니다. 신호 레벨이 일정한 환경이나 정밀한 측정이 필요할 때 유용합니다.
- **automatic**: ADRV9002가 내부 알고리즘에 따라 자동으로 최적의 이득을 설정합니다. 이는

TES에서 설정한 AGC 파라미터(공격 시간, 감쇠 시간, 문턱값 등)에 따라 동작합니다.
- **pin**: 외부 제어 핀을 통해 이득을 제어하는 모드로, 하드웨어 수준의 빠른 이득 제어가 필요할 때 사용됩니다.

이 실습에서는 manual 모드와 automatic 모드의 차이를 비교해 봅니다. 진폭이 시간에 따라 변하는 신호(예: AM 방송 또는 사람이 마이크에 대고 말하는 SSB 신호)를 수신하면서 두 모드의 동작을 관찰합니다.

Python

```
#... (SDR 연결 및 수신 주파수 설정)...

# 1. Manual Gain Control
sdr.gain_control_mode_chan0 = "manual"
sdr.rx_hardwaregain_chan0 = 50 # 고정된 이득
print("Receiving with Manual Gain...")
manual_samples = sdr.rx()

# 2. Automatic Gain Control
sdr.gain_control_mode_chan0 = "automatic"
print("Receiving with Automatic Gain...")
auto_samples = sdr.rx()

# 시간 영역 파형 비교
plt.figure(figsize=(15, 10))
plt.subplot(2, 1, 1)
plt.plot(np.abs(manual_samples))
plt.title("Received Signal Envelope (Manual Gain)")
plt.ylabel("Amplitude")
plt.grid(True)
plt.subplot(2, 1, 2)
plt.plot(np.abs(auto_samples))
plt.title("Received Signal Envelope (Automatic Gain)")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.grid(True)
plt.tight_layout()
plt.show()
```

결과 플롯을 보면, manual 모드에서는 신호의 원래 진폭 변화가 그대로 나타나는 반면,

automatic 모드에서는 AGC가 작동하여 신호의 진폭이 거의 일정하게 유지되는 것을 볼 수 있습니다. automatic 모드는 신호의 세기가 급격하게 변하는 환경에서도 수신기 ADC가 포화되거나 너무 약한 신호를 받는 것을 방지하여 안정적인 통신을 가능하게 합니다.

31단계**: 40 MHz** 전체 대역폭 활용

Jupiter SDR의 가장 큰 장점 중 하나는 ADRV9002가 제공하는 채널당 최대 40 MHz의 넓은 RF 대역폭입니다.[2] 이 정도의 대역폭은 최신 무선 통신 표준인 WiFi(20/40/80 MHz 채널)나 LTE(최대 20 MHz 채널)의 전체 채널을 한 번에 수신하고 분석하기에 충분합니다.

이 기능을 활용하려면 먼저 40 MHz 대역폭을 지원하도록 설정된 ADRV9002 프로파일을 로드해야 합니다 (26단계 참조). 프로파일이 로드되면, pyadi-iio를 통해 해당 프로파일에서 정의된 최대 샘플 레이트와 RF 대역폭을 설정할 수 있습니다.

아래 코드는 2.4 GHz WiFi 대역의 40 MHz 채널을 캡처하고 그 스펙트럼을 분석하는 예제입니다.

Python

```
import adi
import numpy as np
import matplotlib.pyplot as plt

#... (SDR 연결 및 40MHz 지원 프로파일 로드)...

# 최대 대역폭 설정
# 이 값들은 프로파일에 의해 결정됩니다.
sample_rate = int(sdr.rx0_sample_rate) # 예: 61.44 MSPS
rx_bandwidth = int(sdr.rx0_rf_bandwidth) # 예: 40 MHz
center_freq = int(2.437e9) # WiFi 채널 6 중심 주파수
num_samples = 2**20 # 대용량 캡처

sdr.rx_enabled_channels =
sdr.rx0_lo = center_freq
sdr.rx_buffer_size = num_samples
#... (이득 설정)...
```

```
print(f"Configured for {rx_bandwidth/1e6} MHz bandwidth capture.")
print("Receiving WiFi band data...")
samples = sdr.rx()

# 스펙트로그램 시각화
plt.figure(figsize=(15, 7))
plt.specgram(samples, NFFT=2048, Fs=sample_rate, Fc=center_freq)
plt.title(f"Spectrogram of {rx_bandwidth/1e6} MHz Wide WiFi Band")
plt.xlabel("Time (s)")
plt.ylabel("Frequency (Hz)")
plt.colorbar()
plt.show()
```

결과 스펙트로그램을 보면, 시간에 따라 변화하는 WiFi 패킷들이 여러 주파수 부채널에 걸쳐 나타나는 복잡한 스펙트럼 구조를 관찰할 수 있습니다. 이처럼 넓은 대역을 한 번에 볼 수 있는 능력은 스펙트럼 모니터링, 인지 라디오(Cognitive Radio), 그리고 통신 프로토콜 리버스 엔지니어링과 같은 고급 연구 및 개발에 필수적입니다.

32단계**:** 송신기 선형성을 위한 디지털 전치 왜곡**(DPD)**

고차 변조 방식(예: 64-QAM, 256-QAM)을 사용하거나 높은 출력으로 신호를 송신할 때, 송신 경로에 있는 전력 증폭기(Power Amplifier, PA)의 비선형성(non-linearity)이 문제가 됩니다. PA가 비선형적으로 동작하면 신호에 왜곡(distortion)이 발생하여 대역 외 방사(out-of-band emission)를 증가시키고 대역 내 신호 품질(EVM, Error Vector Magnitude)을 저하시킵니다.

디지털 전치 왜곡(Digital Pre-Distortion, DPD)은 이러한 PA의 비선형성을 보상하기 위한 고급 신호 처리 기술입니다.[4] DPD의 기본 원리는 PA가 발생시킬 왜곡을 미리 예측하고, 이 왜곡의 역(inverse) 특성을 가진 신호를 디지털 단에서 미리 만들어주는 것입니다. 이 "미리 왜곡된" 신호가 비선형적인 PA를 통과하면서 원래의 선형적인 신호로 복원됩니다.

ADRV9002는 이러한 DPD 기능을 내장하고 있으며, pyadi-iio를 통해 관련 캘리브레이션을 제어할 수 있습니다. 완전한 DPD 모델을 구현하는 것은 매우 복잡한 작업이지만, 내장된 추적 캘리브레이션을 활성화하여 기본적인 선형성 개선 효과를 얻을 수 있습니다.

pa_correction_tracking_en_chan0 속성은 DPD를 포함한 전력 증폭기 보정 추적 캘리브레이션을 활성화하는 역할을 합니다.[15]

Python

```
#... (송신 설정 코드)...

# DPD 관련 추적 캘리브레이션 활성화
# 이 기능을 사용하려면 특정 프로파일 및 HDL 빌드가 필요할 수 있습니다.
try:
    sdr.pa_correction_tracking_en_chan0 = True
    print("PA Correction (DPD) tracking enabled.")
except Exception as e:
    print(f"Could not enable PA correction tracking: {e}")

#... (고차 변조 신호 송신)...
```

DPD의 효과를 정량적으로 확인하려면 스펙트럼 분석기를 사용하여 인접 채널 누설비(ACLR, Adjacent Channel Leakage Ratio)나 EVM을 측정해야 합니다. DPD를 활성화했을 때 ACLR이 개선되고 EVM이 감소하는 것을 통해 송신기 선형성이 향상되었음을 확인할 수 있습니다.

33단계**:** 빠른 주파수 호핑

주파수 호핑(Frequency Hopping)은 통신 채널을 미리 정해진 패턴에 따라 빠르게 변경하는 기술로, 주로 군 통신에서 도청이나 재밍(jamming)을 회피하기 위해 사용됩니다. ADRV9002는 매우 빠른 속도로 LO 주파수를 변경할 수 있는 고속 주파수 호핑 기능을 지원합니다.[4]

pyadi-iio를 사용하여 Python 루프 내에서 sdr.rx0_lo 속성을 반복적으로 변경함으로써 간단한 주파수 호핑 수신기 또는 스펙트럼 스캐너를 구현할 수 있습니다.

Python

```
import adi
import time

#... (SDR 연결)...
sdr = adi.adrv9002(uri="ip:192.168.2.1")

# 스캔할 주파수 목록 (Hz)
hop_frequencies = [int(f*1e6) for f in range(902, 928)] # 902-928 MHz ISM 대역
```

```
# 수신 설정
sdr.rx_enabled_channels =
sdr.rx0_sample_rate = int(1e6)
sdr.rx_buffer_size = 1024
#... (이득 설정)...

print("Starting fast frequency hopping scan...")
try:
    while True:
        for freq in hop_frequencies:
            # LO 주파수 변경
            sdr.rx0_lo = freq

            # 해당 주파수에서 짧게 데이터 수신
            samples = sdr.rx()

            # 수신된 데이터의 평균 전력 계산
            power = 10 * np.log10(np.mean(np.abs(samples)**2))

            print(f"Freq: {freq/1e6:.1f} MHz, Power: {power:.2f} dB")

            time.sleep(0.01) # 호핑 간격 (dwell time)

except KeyboardInterrupt:
    print("\nScan stopped.")
```

이 스크립트는 지정된 주파수 목록을 빠르게 순회하면서 각 주파수에서의 신호 전력을 측정하고 출력합니다. 이를 통해 특정 대역의 채널 점유 상태를 실시간으로 모니터링할 수 있습니다. ADRV9002의 빠른 튜닝 속도 덕분에 수십, 수백 개의 채널을 초당 여러 번 스캔하는 것이 가능합니다.

34단계**:** 온디바이스 프로세싱 **- MPSoC**의 장점

ADALM-PLUTO를 사용할 때, 수신된 IQ 데이터는 USB를 통해 호스트 PC로 전송된 후 모든 신호 처리가 PC의 CPU에서 이루어집니다. 이는 대역폭이 넓어지거나 처리 알고리즘이 복잡해지면 USB 대역폭이나 PC의 CPU 성능이 병목이 될 수 있음을 의미합니다.

Jupiter SDR은 이러한 패러다임을 근본적으로 바꿉니다. 5단계에서 SSH로 접속했던 것처럼, Jupiter SDR은 그 자체가 강력한 쿼드 코어 ARM 프로세서를 탑재한 임베디드 리눅스

컴퓨터입니다.[5] 따라서 우리가 작성한 Python 스크립트를 호스트 PC가 아닌 Jupiter SDR 내부에서 직접 실행할 수 있습니다.

이것이 바로 "온디바이스 프로세싱"의 개념입니다. IQ 데이터를 네트워크로 전송하는 대신, Jupiter SDR 내부에서 직접 FFT, 복조, 디코딩 등의 모든 신호 처리를 수행하고, 최종 결과(예: 텍스트 메시지, 측정값)만을 네트워크를 통해 호스트 PC로 전송합니다.

이 접근 방식의 장점은 다음과 같습니다.

- 네트워크 대역폭 감소: 수십 MSPS의 원시 IQ 데이터 대신 처리된 결과만 전송하므로 네트워크 부하가 극적으로 줄어듭니다.
- 저지연 처리: 데이터가 장치 내부에서 바로 처리되므로 호스트 PC까지의 전송 지연이 없습니다.
- 자율 시스템 구현: 호스트 PC 없이 전원만 연결된 상태로 Jupiter SDR이 독립적으로 임무를 수행하는 자율적인(autonomous) 시스템을 구축할 수 있습니다. 이는 현장 배치형 센서나 통신 중계기에 이상적입니다.

사용자는 SSH를 통해 Jupiter SDR에 접속한 후, 평소처럼 Python 스크립트를 실행하기만 하면 됩니다. pyadi-iio는 로컬에서 실행될 때 자동으로 local: 백엔드를 사용하여 FPGA 및 ADRV9002와 직접 통신합니다.

35단계: 실시간 프로세싱과 Cortex-R5F

Zynq MPSoC의 또 다른 강력한 기능은 실시간 처리를 위해 설계된 듀얼 코어 ARM Cortex-R5F 프로세서입니다.[5] 우리가 Python 스크립트를 실행하는 Cortex-A53 코어는 리눅스와 같은 범용 운영체제(OS)를 실행하므로, 스케줄링 등으로 인해 작업 실행 시간에 약간의 지연 변동(jitter)이 있을 수 있습니다.

반면, Cortex-R5F는 실시간 운영체제(RTOS)나 베어메탈(bare-metal) 환경에서 동작하며, 매우 낮은 지연 시간과 결정성(determinism)을 보장합니다. 이는 모터 제어, 빠른 프로토콜 처리, 기능 안전(functional safety)이 요구되는 애플리케이션에 필수적입니다.

Python 튜토리얼의 범위를 넘어서지만, 사용자는 Xilinx/AMD의 Vitis 및 PetaLinux 개발 도구를 사용하여 Cortex-R5F용 C/C++ 코드를 작성하고, 이를 Cortex-A53에서 실행되는 리눅스 애플리케이션과 연동할 수 있습니다. 예를 들어, R5F 코어는 RF 프론트엔드를 정밀하게 제어하거나 고속의 MAC 계층 프로토콜을 처리하고, A53 코어는 상위 계층의 네트워크 스택이나 사용자 인터페이스를 처리하는 역할 분담이 가능합니다. 이는 Jupiter SDR 플랫폼이 가진 잠재력의 깊이를 보여주며, 향후 더 복잡한 시스템을 설계할 때 고려해볼 수 있는 고급 아키텍처입니다.

Part IV: 시스템 통합 및 향후 방향 **(36-40**단계**)**

마지막 파트에서는 지금까지 배운 기술들을 종합하여 더 복잡한 시스템을 구축하는 프로젝트를 수행하고, Python 스크립팅을 넘어선 학습 경로를 제시합니다.

36단계**:** 프로젝트 **-** 광대역 스펙트럼 모니터 구축

이 관석 프로젝트는 여러 이전 단계에서 배운 기술들을 통합하여 실용적인 애플리케이션을 만듭니다. 목표는 지정된 주파수 대역을 지속적으로 모니터링하고, 특정 전력 임계값을 초과하는 신호를 자동으로 감지하여 해당 신호의 중심 주파수, 대역폭, 전력과 같은 정보를 기록하는 광대역 스펙트럼 모니터를 구축하는 것입니다.

이 프로젝트는 광대역 수신(31단계), FFT 기반 신호 처리(9, 11단계), 그리고 온디바이스 프로세싱(34단계) 개념을 결합합니다. 스크립트는 Jupiter SDR 내부에서 실행되어야 효율적입니다.

구현 단계**:**

1. 연속 수신 루프: while True: 루프를 사용하여 지속적으로 광대역 IQ 데이터를 수신합니다.
2. **PSD** 계산: 각 데이터 블록에 대해 FFT를 수행하여 PSD를 계산합니다.
3. 신호 검출: PSD 데이터에서 노이즈 레벨보다 충분히 높은 피크들을 찾습니다. scipy.signal.find_peaks 함수가 유용합니다.
4. 파라미터 추정: 검출된 각 피크에 대해 중심 주파수, 6dB 대역폭, 피크 전력 등을 추정합니다.
5. 로깅: 검출된 신호의 정보(타임스탬프, 주파수, 대역폭, 전력)를 화면에 출력하거나 파일에 기록합니다.

이 프로젝트는 인지 라디오의 '스펙트럼 센싱' 기능을 구현한 것으로, 동적으로 비어있는 채널을 찾거나, 불법적인 무선 신호를 감지하는 등 다양한 실제 시나리오에 적용될 수 있습니다.

37단계**:** 프로젝트 **-** 간단한 **2x1 SIMO** 송수신기

이 프로젝트는 다중 안테나 기술을 활용하여 통신 링크의 신뢰성을 향상시키는 시스템을 구축합니다. 2x1 SIMO(Single Input, Multiple Output - 여기서는 수신 측이 다중 안테나이므로

반대 개념)는 하나의 송신 안테나와 두 개의 수신 안테나를 사용하는 시스템을 의미합니다.

이 시스템은 28단계에서 구현한 수신 다이버시티 기술을 실제 통신 링크에 적용합니다.

시스템 구성**:**

- 송신기 **(SISO)**: ADALM-PLUTO 또는 다른 SDR이 하나의 안테나를 사용하여 23단계에서 만든 QPSK 패킷을 송신합니다.
- 수신기 **(1x2 SIMO)**: Jupiter SDR이 두 개의 수신 안테나(RX0, RX1)를 사용하여 신호를 동시에 수신합니다.
- 신호 처리: 수신된 두 신호에 MRC(최대비 결합)를 적용하여 결합된 신호를 생성합니다.
- 복조 및 **BER** 계산: 결합된 신호를 사용하여 복조를 수행하고 BER을 계산합니다.

한 개의 안테나만 사용했을 때(SISO)의 BER과 두 개의 안테나와 MRC를 사용했을 때(SIMO)의 BER을 동일한 환경에서 비교 측정합니다. 페이딩이 심한 실내 환경 등에서 테스트하면, SIMO 시스템이 훨씬 더 낮고 안정적인 BER을 제공하는 것을 확인할 수 있습니다. 이는 다중 안테나 기술의 실질적인 이점을 명확하게 보여주는 프로젝트입니다.

38단계**: GNU Radio**와의 통합

pyadi-iio는 스크립트 기반의 유연한 제어를 제공하지만, 많은 SDR 사용자는 시각적인 플로우그래프 환경인 GNU Radio Companion에 익숙합니다. Jupiter SDR은 gr-iio 라이브러리를 통해 GNU Radio와 완벽하게 호환됩니다. gr-iio는 IIO 장치를 위한 소스 및 싱크 블록을 제공하여, Jupiter SDR을 GNU Radio 플로우그래프에 쉽게 통합할 수 있게 해줍니다.[13]

통합 방법**:**

1. **gr-iio** 설치: 호스트 PC 또는 Jupiter SDR에 gr-iio가 설치되어 있는지 확인합니다. ADI가 제공하는 SD 카드 이미지에는 일반적으로 사전 설치되어 있습니다.
2. **IIO Attribute Sink** 블록: 이 블록을 사용하여 플로우그래프 내에서 Jupiter SDR의 속성(예: rx0_lo, rx_hardwaregain_chan0)을 설정합니다. GUI 슬라이더와 같은 위젯을 연결하여 실시간으로 주파수나 이득을 변경할 수 있습니다.
3. **IIO Device Source** 블록: 이 블록은 Jupiter SDR에서 IQ 데이터를 수신하여 플로우그래프로 가져오는 역할을 합니다. 장치 URI(ip:192.168.2.1)와 수신할 채널 수, 버퍼 크기 등을 설정합니다.
4. **IIO Device Sink** 블록: 이 블록은 플로우그래프에서 처리된 IQ 데이터를 Jupiter SDR로 보내 송신하는 역할을 합니다.

GNU Radio를 사용하면 FFT, 필터, 복조기 등 다양한 신호 처리 블록들을 시각적으로 연결하여 복잡한 수신기를 빠르게 프로토타이핑할 수 있습니다. Python 스크립팅과 GNU Radio를 함께 사용하면, 각각의 장점을 활용하여 더욱 효율적인 개발이 가능합니다.

39단계**: HDL** 커스터마이제이션으로의 길

지금까지의 모든 실습은 Jupiter SDR의 Zynq MPSoC에 내장된 ARM 프로세서, 즉 소프트웨어 영역에서 이루어졌습니다. 그러나 Zynq 플랫폼의 진정한 힘은 프로세싱 시스템(PS)과 프로그래머블 로직(PL), 즉 FPGA를 긴밀하게 통합한 데 있습니다.

매우 높은 처리량이 요구되거나 나노초 단위의 지연 시간이 중요한 알고리즘(예: 대규모 FFT, 고속 상관기, 실시간 암호화)은 Python이나 C 코드로 실행하기에 너무 느릴 수 있습니다. 이러한 경우, 해당 기능을 FPGA 패브릭에 하드웨어 회로로 직접 구현(HDL 커스터마이제이션)하여 소프트웨어 대비 수십, 수백 배의 성능 향상을 얻을 수 있습니다.

ADI는 Jupiter SDR을 포함한 모든 SDR 플랫폼의 FPGA 설계(HDL 코드)를 오픈소스로 제공합니다.[3]

- **ADI HDL** 저장소: GitHub(github.com/analogdevicesinc/hdl)에서 Verilog HDL 소스 파일과 프로젝트 빌드를 위한 Tcl 스크립트를 찾을 수 있습니다.
- 개발 도구: AMD/Xilinx의 Vivado Design Suite를 사용하여 HDL 코드를 수정하고, 합성(synthesis) 및 구현(implementation)을 거쳐 새로운 FPGA 비트스트림(bitstream) 파일을 생성할 수 있습니다.
- 통합: AXI4(Advanced eXtensible Interface 4) 버스를 사용하여 커스텀 HDL 블록을 MPSoC의 ARM 프로세서와 연결할 수 있습니다. 이를 통해 리눅스에서 실행되는 소프트웨어가 메모리 맵 레지스터를 통해 커스텀 하드웨어 가속기를 직접 제어할 수 있습니다.

HDL 개발은 상당한 학습 곡선을 요구하지만, 이는 Jupiter SDR 플랫폼의 성능을 극한까지 활용할 수 있는 궁극적인 경로입니다.

40단계**:** 결론 및 추가 학습을 위한 로드맵

이 40단계의 튜토리얼을 통해 사용자는 ADALM-PLUTO와 pySDR.org에 대한 기존 지식을 바탕으로, 전문가급 SDR 플랫폼인 Jupiter SDR을 효과적으로 활용하는 시스템 통합 전문가로 거듭났습니다.

주요 학습 성과 요약**:**

- 하드웨어 아키텍처 이해: AD9363/Zynq-7010과 ADRV9002/Zynq MPSoC의 핵심적인 차이점과 그 성능적 의미를 이해했습니다.
- **API** 마이그레이션: pyadi-iio 라이브러리 내에서 adi.Pluto와 adi.adrv9002 클래스 간의 API

차이를 극복하고, 명시적인 다중 채널 제어 방식을 습득했습니다.
- 핵심 **SDR** 개념 구현: pySDR.org의 핵심 커리큘럼(변조, 필터링, 동기화 등)을 Jupiter SDR의 우수한 하드웨어 위에서 성공적으로 구현했습니다.
- 고급 기능 활용: 듀얼 채널 동기 수신, 다이버시티, 방향 탐지, 광대역 스펙트럼 분석 등 Pluto에서는 불가능했던 고급 애플리케이션을 구현했습니다.
- 시스템 패러다임 전환: SDR을 단순한 PC 주변기기가 아닌, 온디바이스 프로세싱이 가능한 독립적인 임베디드 라디오 컴퓨터로 인식하게 되었습니다.

이 튜토리얼은 Jupiter SDR의 방대한 가능성을 탐험하기 위한 시작점입니다. 더 깊이 있는 학습을 위해 다음 자료들을 참고할 것을 권장합니다.

- **ADI EngineerZone** 포럼: Jupiter SDR 및 pyadi-iio와 관련된 기술적인 질문을 하고 다른 개발자들과 지식을 공유할 수 있는 최고의 커뮤니티입니다.[16]
- **ADI Wiki**: ADI의 모든 하드웨어 및 소프트웨어에 대한 공식 문서와 사용자 가이드가 있는 곳입니다. Jupiter SDR 및 HDL/Linux 프로젝트에 대한 최신 정보를 얻을 수 있습니다.
- **AMD/Xilinx** 문서: Zynq UltraScale+ MPSoC의 아키텍처, Vitis 통합 개발 환경, PetaLinux 임베디드 리눅스 구축 도구에 대한 심층적인 정보를 제공합니다.
- 고급 통신 이론: "Digital Communications" by John G. Proakis와 같은 교재를 통해 MIMO, OFDM, 고급 채널 코딩 등 현대 통신 시스템의 기반이 되는 이론을 학습할 수 있습니다.

Jupiter SDR이라는 강력한 도구를 손에 넣은 지금, 여러분은 상상하는 거의 모든 무선 시스템을 현실로 만들 수 있는 잠재력을 갖게 되었습니다. 지속적인 학습과 실험을 통해 SDR 기술의 최전선에서 혁신을 이끌어 나가기를 바랍니다.

참고 자료

1. Jupiter: Software-Defined Reference Design - Analog Devices, 9월 21, 2025에 액세스, https://www.analog.com/en/resources/media-center/videos/6351261749112.html
2. AD-JUPITER-EBZ Evaluation Board - Analog Devices, 9월 21, 2025에 액세스, https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/ad-jupiter-ebz.html
3. SOFTWARE COMPONENTS FOR - GNU Radio Events, 9월 21, 2025에 액세스, https://events.gnuradio.org/event/24/contributions/660/attachments/211/521/GRCON2024_OSS-HDL-IP_HENNERICH.pdf
4. ADRV9002 Dual Narrow/Wideband RF Transceiver - ADI | Mouser, 9월 21, 2025에 액세스, https://www.mouser.com/new/analog-devices/adi-adrv9002-rf-transciever/
5. Zynq UltraScale+ MPSoCs - AMD, 9월 21, 2025에 액세스, https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html
6. Characteristics of FPGAs, Zynq, and Zynq MPSoC devices - Vemeko FPGA, 9월 21, 2025에 액세스, https://www.vemeko.com/blog/67143.html
7. Zynq Migration Guide: Zynq-7000 SoC to Zynq UltraScale+ MPSoC Devices, 9월 21, 2025에 액세스,

https://static.eetrend.com/files/2020-01/wen_zhang_/100047000-88375-ug1213-zynq-migration-guide.pdf
8. Analog Devices ADRV9002 | Demoboard Jupiter-SDR | Würth Elektronik Reference Design Search, 9월 21, 2025에 액세스, https://www.we-online.com/en/components/icref/analog-devices/ADRV9002-Jupiter-SDR-RF-signals-communications
9. Zynq® UltraScale+™ MPSoC and Road to Versal - CERN Indico, 9월 21, 2025에 액세스, https://indico.cern.ch/event/799275/contributions/3413723/attachments/1860639/3057774/Zynq_UltraScale_MPSoC_Product_Overview_-_CERN_-_INDICO.pdf
10. PlutoSDR in Python | PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/content/pluto.html
11. Quick Start — Analog Devices Hardware Python Interfaces 0.0.19 documentation, 9월 21, 2025에 액세스, https://analogdevicesinc.github.io/pyadi-iio/guides/quick.html
12. Analog Devices Hardware Python Interfaces 0.0.19 documentation, 9월 21, 2025에 액세스, https://analogdevicesinc.github.io/pyadi-iio/
13. gr-iio: Nuances, Hidden Features, and New Stuff - GNU Radio, 9월 21, 2025에 액세스, https://www.gnuradio.org/grcon/grcon19/presentations/gr-iio_Nuances_Hidden_Features_and_New_Stuff/Travis%20Collins%20-%20gr_iio.pdf
14. analogdevicesinc/pyadi-iio: Python interfaces for ADI hardware with IIO drivers (aka peyote), 9월 21, 2025에 액세스, https://github.com/analogdevicesinc/pyadi-iio
15. adrv9002 — Analog Devices Hardware Python Interfaces 0.0.19 documentation, 9월 21, 2025에 액세스, https://analogdevicesinc.github.io/pyadi-iio/devices/adi.adrv9002.html
16. ADRV9002 + ZCU102 + Pyadi iio + RX Testing - Q&A - Software Interface Tools, 9월 21, 2025에 액세스, https://ez.analog.com/sw-interface-tools/f/q-a/539809/adrv9002-zcu102-pyadi-iio-rx-testing
17. Introduction | PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/content/intro.html
18. PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/
19. Digital Modulation | PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/content/digital_modulation.html
20. Link Budgets | PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/content/link_budgets.html
21. Channel Coding | PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/content/channel_coding.html
22. Synchronization | PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/content/sync.html
23. Multi-channel transmission/reception using pyadi-iio - Q&A - Software Interface Tools, 9월 21, 2025에 액세스, https://ez.analog.com/sw-interface-tools/f/q-a/595794/multi-channel-transmissio

n-reception-using-pyadi-iio
24. Beamforming & DOA | PySDR: A Guide to SDR and DSP using Python, 9월 21, 2025에 액세스, https://pysdr.org/content/doa.html
25. ADALM-PLUTO and IIO Workshop - YouTube, 9월 21, 2025에 액세스, https://www.youtube.com/watch?v=Yx3RPOtv7x8
26. HDL Reference Designs - Analog Devices, Inc., 9월 21, 2025에 액세스, http://analogdevicesinc.github.io/hdl/
27. AD9361 and ADRV9002 - Q&A - EngineerZone - Analog Devices, 9월 21, 2025에 액세스, https://ez.analog.com/rf/wide-band-rf-transceivers/design-support/f/q-a/590448/ad9361-and-adrv9002