Hardware Emulations

using 5G Toolkit and SDRs: Hands-on

For academia only

Contributors:

Editor:	Simulator Design Team, GIGAYASA
Contributing Authors:	Aman Mishra, Vijaya Mareedu and Vikram Singh, GIGAYASA

Grants and Funding

Gigayasa is supported by:

- Indian Institute of Technology, Madras Incubation Center (IITM-IC).
- Startup India.
- Department of Telecommunication (DoT), India.
- Center of Excellence in Wireless Technology (CEWiT), IITM.
- Ministry of Electronics and Information Technology (MEITy), India.

Contents

1 Introduction to ADALM-Pluto SDR and its application programming interface **1**
1.1 What are Software Defined Radios (SDRs)? 1
1.2 Why do we use SDRs? 3
1.3 Specifications of some famous SDRs 3
1.4 Introducing some important Pluto SDR APIs 3
1.5 Useful Resources 7

2 References **9**

1 | Introduction to ADALM-Pluto SDR and its application programming interface

Welcome to the hand-on laboratory course on **5G wireless systems**. In this course, we will introduce the audience to some practical aspects related to 5G systems using the 5G toolkit and software defined radios (SDRs). We will be using ADALM-Pluto SDR primarily due to the availability, cost effectiveness and ease of use. However, the course can experienced using any other SDR available with you as long as it supports the Python APIs. These APIs will be used for passing the I/Q samples of 5G signals and information generated using 5G -Toolkit to radio unit of the SDRs for analog processing followed by transmission of processed signal over the wireless channel using antennas.

1.1 | What are Software Defined Radios (SDRs)?

Software Defined radios (SDRs) are radio communication systems which allows controlling one or more aspects of the radio unit's functionality using software [1]. The initial communication systems were hard coded systems hence building SDRs seemed lucrative and aspirations. However, with the advent of the technology

1.1.1 | Architecture of the SDRs

The following diagrams shows the a simplified architecture of the SDR which consists primarily for 3 parts.

- Baseband Unit
- Radio Unit
- Antennas

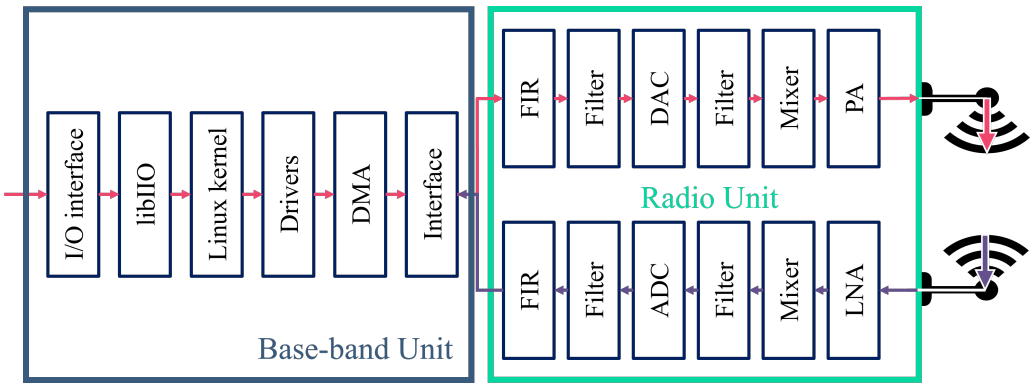


Figure 1.1: Architecture of a simple SDR (ADALM Pluto)

1.1.2 | Components of the SDRs

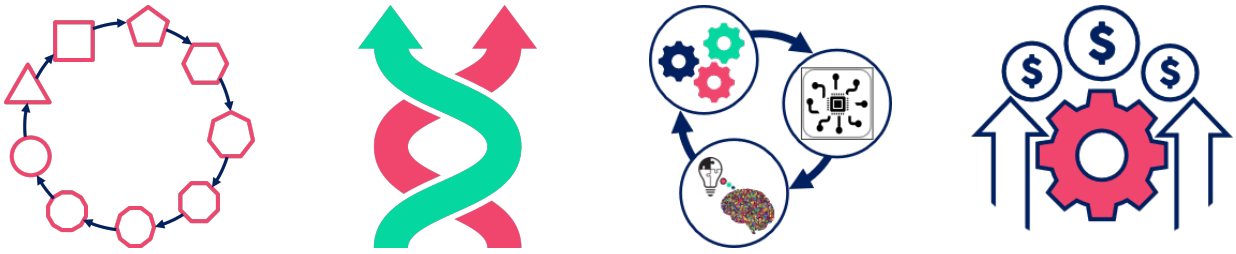
Software-Defined Radio (SDR) systems consist of various components and modules that work together to enable the flexibility and programmability characteristic of SDR. Here are some important components typically found inside an SDR:

- **RF Front-End:**
  - **Antenna**: The antenna receives radio frequency (RF) signals from the air.
  - **Low Noise Amplifier (LNA)**: Amplifies weak incoming signals from the antenna.
- **Analog-to-Digital Converter (ADC)**:
  - Converts the analog RF signal into digital form for processing by the digital components.
- **Digital Signal Processor (DSP)**:

- Converts the analog RF signal into digital form for processing by the digital components.

- **Field-Programmable Gate Array (FPGA)**:
  - A programmable logic device that provides hardware acceleration for certain processing tasks, enhancing the overall performance of the SDR.
- **Processor and General-Purpose Computer**:
  - The main computing unit responsible for executing higher-level software applications and managing the overall SDR system.
- **Software Interface**:
  - Allows users to interact with the SDR, configure settings, and control its operation. This can include graphical user interfaces (GUIs) or command-line interfaces.
- **Operating System**:
  - Provides a platform for running software applications and managing hardware resources.
- **Digital-to-Analog Converter (DAC)**:
  - Converts digital signals back to analog form before transmission through the RF front-end. Digital-to-Analog Converter (DAC):
- **Power Supply**:
  - Provides the necessary power to different components within the SDR.
- **Communication Interfaces**:
  - Interfaces for connecting the SDR to external devices or networks. This can include USB, Ethernet, or wireless interfaces.
- **Clocking System**:
  - Provides accurate timing for the various components to ensure proper synchronization.
- **Memory and Buffers**:
  - Stores digital samples, configuration settings, and other data. This can include both volatile (RAM) and non-volatile (storage) memory.
- **Software Stack**:
  - The software stack includes the SDR's firmware and higher-level software applications that define the SDR's functionality. It enables the reprogramming and customization of the radio's behavior.
- **Digital Upconverter (DUC) and Digital Downconverter (DDC)**:
  - The DUC converts baseband digital signals to the intermediate frequency (IF) or RF for transmission, while the DDC converts incoming RF signals to baseband for further processing.
- **Filtering and Amplification Stages**:
  - Digital and analog filters, as well as amplifiers, may be included in the signal path to enhance or modify the signal.

These components work together to create a flexible and programmable radio system, allowing users to implement a wide range of communication protocols and adapt to various operating conditions. The combination of hardware and software elements enables the versatility that distinguishes SDR from traditional hardware-based radio systems.



(a) Adaptability (b) Flexibility (c) Rapid Prototyping (d) Cost Efficient

Figure 1.2: Utility of SDRs

1.2 | Why do we use SDRs?

The primary advantage of SDR lies in its flexibility and adaptability, as it allows for the reconfiguration and reprogramming of the radio's functionality through software updates. Here are some utilities of SDR and how they differ from traditional hardware-based radios:

1.2.1 | Adaptability

SDRs can operate on multiple frequency bands and support various communication standards concurrently. This is advantageous for applications that require interoperability across different frequency bands or standards. Upgrades can be implemented through software updates, allowing for improvements in performance, security, or additional features without requiring hardware modifications.

1.2.2 | Flexibility and Reconfigurability

With SDR, the radio's functionality can be modified through software updates, enabling rapid adaptation to changing communication standards. This flexibility is particularly beneficial for applications like military communication, emergency services, and research.

1.2.3 | Rapid Prototyping and Validation

SDRs are popular in research and development environments because they provide a platform for experimenting with new communication techniques and protocols without the need to build new hardware for each experiment.

1.2.4 | Cost Efficiency

While the initial cost of developing SDR may be high, the ability to adapt and upgrade through software can result in long-term cost savings, especially in applications with evolving communication standards.

1.3 | Specifications of some famous SDRs

The popularity and relevance of SDRs generally change over time. There are many good SDRs availble in the market which are being used for variety of the purposes. However, as per our expertise the SDRs listed in table 1.1 are most suitable SDRs for learning, teaching and academic research on 5G. These SDRs supports Python APIs and are within the budget for academic institutes and Wireless SDR amateurs. There are other SDRs available in the market such as RTL-SDR, Xilinx RFSoCs, NI USRPs but these SDRs are either reception only, or too expensive or doesn't support Python APIs.

1.4 | Introducing some important Pluto SDR APIs

The software functionalities of the SDRs are controlled using the APIs which are provided by the SDR vendor. These APIs are segregated into properties (variables) and methods (functions). The properties set the important parameters which are crucial to the operation of the radio units and its components. On the other hand, the methods are crucial for reading from and writing to the SDRs (I/O interfacing). We have segregated the APIs into three categories which are the SDR setup API, transmitter APIs and receiver APIs. Lets discuss these APIs in detail.

Table 1.1: Specifications of the some famous SDRs

Function	Pluto-SDR	LimeSDR mini 2.0	AntSDR E200	USRP B210/200/205mini
Chipset	AD9363	LMS7002M	AD9361/9363	AD9361/9363/9363
Frequency Range	325 MHz - 3.8 GHz	10 MHz - 3.5 GHz	70/325 MHz - 6/3.8 GHz	70 MHz - 6 GHz
Interface	USB 2.0	USB 3.0	Gb Ethernet	USB 3.0
Embedded	Yes	No	Yes	No
RF Bandwidth	20 MHz	40 MHz	56/20 MHz	61.44 MHz
Sample Depth	12 bits			
Sample Rate	61.44 MSPS	30.72 MSPS	61.44 MSPS	61.44 MSPS
Tx Channels	2	1	2	2/1/1
Rx Channels	2	1	2	2/1/1
Programmable Logic Gates	28k	44k	85k	100k/75k/150k
Memory	512 MB			
Duplex	Full			
Open Source	Full		Schematic & firmware	
Oscillator Precision	± 20 ppm	± 1 ppm initial, ± 4 ppm stable	± 2 ppm	
Transmit Power*	≤ 6 dBm	Up to 10 dBm		≥ 10 dBm
Cost†	$250	$400	$600/$350	$2300/$1300/$1600

Note-1: Lime Microsystems is currently shipping LimeSDR mini 2.0 only.
Note-2: AntSDR E200 comes in two versions which differs in the RF-chipsets:

- AntSDR E200 with AD9361
- AntSDR E200 with AD9363

* The maximum transmit power of a RF chipset depends on the carrier frequency.
† The prices may change over time and doesn't include the import/excise duty.

1.4.1 | APIs for SDR Setup

Before transmitting or receiving the data using SDRs, we have to configure some basic setup parameters crucial to SDR operation. These APIs connects the specific SDR over USB, set the sample rate for SDR operation etc.

Table 1.2: Some important application interfaces (APIs) used in Analog Devices interface (ADI)

Type	API	Function
method	Pluto	Create object of SDR setup object.
property	sample_rate	Sets the sample rate for the ADC/DAC of the SDR.

adi.Pluto(uri=ip)

This method creates an SDR object which can access the device with *uri* address of *ip*. Following code shows a few ways to do this.

```
sdr = adi.Pluto(uri="ip:192.168.2.1")
# or
sdr = adi.Pluto("ip:192.168.2.1")
# or
sdr = adi.ad9361(uri="ip:192.168.2.1")
# or
sdr = adi.ad9361("ip:192.168.2.1")
```

sdr.sample_rate

This property is used to set the sample rate (in SPS) used by SDR for analog to digital conversion (ADC) and digital to analog conversion (DAC) at the receiver and transmitter path respectively.

```
sdr = adi.Pluto(uri="ip:192.168.2.1") # Setup object connected to SDR with given IP.
Nfft              = 1024              # FFT size
subcarrierSpacing = 30*(10**3)        # subcarrier spacing
sdr.sample_rate   = int(sample_rate)  # set the sample rate
```

1.4.2 | Transmitter APIs

The APIs allows the users to control carrier frequency, signal amplification, bandwidth of the transmitted signal, transmit signal, configure the repeated transmission of the buffered I/Q samples and clear the transmit buffer.

Table 1.3: Some important APIs related to transmitter used in ADI

Type	API	Function
method	tx	Transmits the samples input to the method.
	tx_destroy_buffer()	Clear the buffer and stop the transmission of data.
property	tx_rf_bandwidth	Set the bandwidth of the transmit filter.
	tx_lo	Sets the transmitter local oscillator frequency.
	tx_hardwaregain_chan0	Sets the gain of the transmitter power amplifier.
	tx_cyclic_buffer	Enable/disable cyclic transmission of samples in buffer.
	tx_enabled_channels	Integer indicating the indices of the tx-channels.

- **sdr.tx(samples)**

  This method is used to *transmit* the *samples* input to it. It is important to note that the input samples is expected to stay within the interval -2^{14} to $2^{14}-1$. Otherwise, the input samples will be a victim of saturation while digital to analog conversion. Its always good to scale the input to these values to relax the need of amplification using power amplifier which is a non linear system. The scale and range of inputs is different for different SDRs.

- **sdr.tx_destroy_buffer()**

  This property is a boolean flag to clear/destroy the transmission buffer and stop the transmission of samples from the tx buffer. Default value is False.

- **sdr.tx_rf_bandwidth**

  This is a property to set the bandwidth (in Hz) of the transmission filter. The variable should comply with the specifications of the SDR. For Pluto SDR, the transmission bandwidth should be between 200 kHz and 56 MHz.

- **sdr.tx_lo**

  This property sets the carrier frequency (in Hz) used by local oscillator to generate the carrier/sinusoid waveform. The carrier is used to modulate/up-convert the analog information signal. For Pluto SDR, this variable can take a value between 325 MHz to 3.8 GHz.

- **sdr.tx_hardwaregain_chan0**

  This property sets the amplification gain (dB) provided by the power amplifier to the modulated signal. The higher the value stronger the output signal and longer a signal can propagate over the wireless medium. For Pluto SDR, this variable can take values between -90 to 0.

- **sdr.tx_cyclic_buffer**

  This property is a boolean flag to enable/disable the cyclic transmission of samples stored in the transmit buffer. This property is very useful for non realtime simulations/emulations. Default value is False.

- **sdr.tx_enabled_channels**

  This is Python list type property used to configure the number of tx channels and their channel indices. The number of entries or length of the property defines the number of tx channel. For Pluto SDR, this property can either have length 1 or 2 (if set up for dual channel transmission).

```
# Import the library
import adi

# Set carrier frequency
carrierFrequency           = 10**9

# Generate 1000 samples
x_time                     = np.random.rand(1000)

# Config Tx
sdr.tx_rf_bandwidth        = int(60*10**6) # filter cutoff, just set it to the same as
    sample rate
sdr.tx_lo                  = int(carrierFrequency)
sdr.tx_hardwaregain_chan0 = -10 # Increase to increase tx power, valid range is -90 to 0
    dB

# Initiate transmission
sdr.tx_cyclic_buffer       = True  # Enable cyclic buffers
sdr.tx(x_time)                # start transmitting
```

1.4.3 | Receiver APIs

The receiver APIs provides the users with methods and properties to sample the wireless channel and store the received samples, control the reception filter bandwidth, strength the received signal and, set the local oscillator frequency. The APIs are discussed briefly in table 1.4 and in detail in this sections.

Table 1.4: Some important APIs related to receiver used in ADI

Type	API	Function
method	rx	Sample the wireless channel and load them into buffer.
property	rx_rf_bandwidth	Set the bandwidth of the receive filter.
	rx_lo	Sets the receiver local oscillator frequency.
	rx_hardwaregain_chan0	Sets the gain of the receive low noise amplifier.
	gain_control_mode_chan0	Defines the mode of receiver AGC.
	rx_buffer_size	Number of samples to read and load into SDR buffer.
	rx_enabled_channels	Integer indicating the indices of the rx-channels.

- **sdr.rx()**

  This method samples the wireless channel using the RF chipset, stores the samples in the receive buffer, and return the content of the receive buffer. The number of samples stored in the buffer is configured using *sdr.rx_buffer_size*.

- **sdr.rx_rf_bandwidth**

  This is a property to set the bandwidth (in Hz) of the reception filter. The variable should comply with the specifications of the SDR. For Pluto SDR, the transmission bandwidth should be between 200 kHz and 56 MHz.

- **sdr.rx_lo**

  This property sets the carrier frequency (in Hz) used by local oscillator to generate the carrier/sinusoid waveform. The carrier is used to demodulate/down-convert the analog information signal. For Pluto SDR, this variable can take a value between 325 MHz to 3.8 GHz.

- **sdr.rx_hardwaregain_chan0**

This property sets the amplification gain (dB) provided by the low noise amplifier to the received signal. The higher the value stronger the output signal. The variable is relevant if and only if the *sdr.gain_control_mode_chan0* is set to ***manual***. For Pluto SDR, this variable can take values between 0 to 74.5 dB. When setting the gain manually, one must ensure that output of low noise amplifier doesn't saturate the ADC.

- **sdr.gain_control_mode_chan0**

  This property is sets the gain control mode for the low noise amplifier (LNA). Pluto SDR supports 3 modes of gain control

  - manual
  - slow_attack
  - fast_attack

  For the manual gain contol, the gain is set using the variable *sdr.rx_hardwaregain_chan0*. The gain can be configured automatically using either *"slow_attack"* or *"fast_attack"* mode. The AGC ensure that the ADC doesn't saturate and loose waveform information. The *"fast_attack"* mode estimates the optimal gain aggressive initially and may results in sharp transients before stabilizing.

- **sdr.rx_buffer_size**

  This property set the receiver buffer size. When the *sdr.rx* is invoked, the SDR starts to sniff the wireless channel and stores the recieved samples in the reciever buffer. The upper limit on the size of this buffer is defined by the DRAM size.

- **sdr.rx_enabled_channels**

  This is Python list type property used to configure the number of rx channels and their channel indices. The number of entries or length of the property defines the number of rx channel. For Pluto SDR, this property can either have length 1 or 2 (if set up for dual channel reception).

```
# Import the library
import adi

# Config Rx
sdr.gain_control_mode_chan0 = 'manual'
sdr.rx_hardwaregain_chan0   = 40.0      # dB
# The receive gain on the Pluto has a range from 0 to 74.5 dB.

# or

sdr.gain_control_mode_chan0 = 'slow_attack'
# # AGC modes:
#     # 1. "manual"
#     # 2. "slow_attack"
#     # 3. "fast_attack"

sdr.rx_lo           = int(carrierFrequency)
sdr.rx_rf_bandwidth = int(60*10**6) # filter width, just set it to the same as sample
    rate for now
sdr.rx_buffer_size  = int(4*buffer_size)

# Clear buffer just to be safe
for i in range (0, 10):
    raw_data = sdr.rx()

# Receive samples
rx_samples = sdr.rx()

# # Stop transmitting
sdr.tx_destroy_buffer()
```

1.5 | Useful Resources

We request all the readers to please explore the following resources as well. These blogs, YouTube channels, and web-pages will be useful for firmware/libraries installation, and utilities of the SDRs beyond wireless communication.

- ADI installation.
  - Installation for Windows
  - Installation for Linux
- Jon Kraft Youtube Channel.
  - Upgrading Pluto to 2×2 MIMO SDR.
  - Simplest way to Upgrade Pluto Firmware.
- Mark Litchman's PySDR.
  - Setup and Getting Started with Pluto SDR
  - Setup and Getting Started with NI USRP B2x0
- Pluto-ADALM SDR official resources.
- Gigayasa's resources: SDR integration with 5G-Toolkit.

2 | References

[1] L.H. Crockett, D. Northcote, and R.W. Stewart. *Software Defined Radio with Zynq Ultrascale+ RFSoC*. Strathclyde Academic Media, 2023.