mobile communications series

SOFTWARE-DEFINED RADIO for ENGINEERS

TRAVIS F. COLLINS
ROBIN GETZ
DI PU
ALEXANDER M. WYGLINSKI

Software-Defined Radio for Engineers

For a listing of recent titles in the *Artech House Mobile Communications*, turn to the back of this book.

Software-Defined Radio for Engineers

Travis F. Collins
Robin Getz
Di Pu
Alexander M. Wyglinski

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalog record for this book is available from the British Library.

ISBN-13: 978-1-63081-457-1

Cover design by John Gomes

10 9 8 7 6 5 4 3 2 1

Dedication

To my wife Lauren
—Travis Collins

To my wonderful children, Matthew, Lauren, and Isaac, and my patient wife, Michelle—sorry I have been hiding in the basement working on this book. To all my fantastic colleagues at Analog Devices: Dave, Michael, Lars-Peter, Andrei, Mihai, Travis, Wyatt and many more, without whom Pluto SDR and IIO would not exist.
—Robin Getz

To my lovely son Aidi, my husband Di, and my parents Lingzhen and Xuexun
—Di Pu

To my wife Jen
—Alexander Wyglinski

Contents

CHAPTER 5

Understanding SDR Hardware

In this chapter, we will discuss the real-world implications of using SDR hardware and fundamentals for interacting with the Pluto SDR from MATLAB. Using Pluto SDR as a template we will provide an introduction in the receive and transmit chains, discussing how analog waveforms become digital samples in MATLAB. Once we have a solid grasp on this process a common code templating will be introduced, which will be used throughout the remaining chapters when working with the radio in MATLAB. This templating will provide a simplified workflow that can help alleviate common problems faced when working with SDR's and specifically Pluto SDR. Finally, the chapter will conclude with a small example to make sure the Pluto SDR is configured correctly with MATLAB.

5.1 Components of a Communication System

The software-defined radio described in Section 5.1.1 can constitute a radio node in anything from a point-to-point link to an element in a large ad hoc network of radios. It can be used as an RFFE to a MATLAB script or Simulink model or it can be programmed and used as a complete stand-alone radio. The radio front end, in this case the Pluto SDR, is a single components in a larger communications system, which would also normally include external filters and band-specific antennas. A description of the communication systems, and the block diagram are shown in Figure 5.1(c). The major aspects of that are

- An analog RF section (atennna, RF filters, input mux, LNA, gain, attenuation, mixer);
- An analog baseband section (analog filters, ADC or DAC);
- Some signal processing units (fixed filters inside a transceiver, or user defined inside a FPGA or DSP, or general-purpose processor).

While Pluto SDR provides a great low-cost platform for STEM education and SDR experimentation, it is representive of many SDRs used in commuications systems. Although it is small and low-cost, the Pluto SDR has enough capability to tackle a wide range of SDR applications, such as GPS or weather satellite receiver or ad hoc personal area networks. The Pluto SDR plays the part of the communications systems described above as follows:

- An analog RF section (atennna, RF filters, input mux, LNA, gain, attenuation, mixer)
  - Antenna and RF filters are expected to be done outside the Pluto SDR and are the responsibility of the end user

  - The remaining portions of the first first bullet (input mux, LNA, gain, attenuation, mixer), are all implmented in the AD9363, Integrated RF Agile Transceiver
- Analog baseband section (analog filters, ADC or DAC) is implmented in the AD9363, Integrated RF Agile Transceiver
- Signal processing; this is split between
  - Parts of signal processing is implmented in the AD9363, Integrated RF Agile Transceiver. This includes the fixed halfband decimiation and interpolation filters and programmable 128-tap FIR filters.
  - Optional filtering and decimation may be done in the Xilinx Zynq' FPGA fabric.
  - The I/Q data is then passed up to the USB to a host, where MATLAB can continue the signal processing.

To understand the details of these pieces, it is necessary to peel back the plastic on the Pluto SDR and look at the devices that make up the device itself. Being a SDR platform specifically targeted for students, not only are schematics for the Pluto SDR readily available, but also the entire software and HDL stack, so we can examine in detail the makeup of the device at any level from hardware to software.

5.1.1 Components of an SDR

Most modern SDR devices typically share a similar structural design, which makes up the receive and/or transmit chains to translate data from the analog RF domain into analog baseband domain, and into IQ samples, and eventually into a software package such as MATLAB. In the very simplest sense the Pluto SDR (shown in Figure 5.1[b]) is made up of two components, as shown in Figure 5.1(a):

- An analog RF section (which specifies the receive and transmit capabilities);
- The communications mechanism (Ethernet, USB) to get IQ data back to host for processing.

Referring to Figure 5.1(c), the receive, transmit, and communication specifications of the ADALM-PLUTO consist of

- Transmit (SMA connector labeled Tx)
  - 300–3,800 GHz, 200–20,000 kHz channel bandwidth, 65.1–61,440 kSPS
  - 2.4 Hz LO step size, 5 Hz sample rate step size
  - Modulation accuracy (EVM): 40 dB (typical, not measured on every unit)
  - 12-bit DACs
- Receive (SMA connector labeled Rx)
  - 300–3,800 GHz, 200–20,000 kHz channel bandwidth, 65.1–61,440 kSPS
  - 2.4 Hz LO step size, 5 Hz sample rate step size
  - Modulation accuracy (EVM): 40 dB (typical, not measured on every unit)
  - 12-bit ADCs

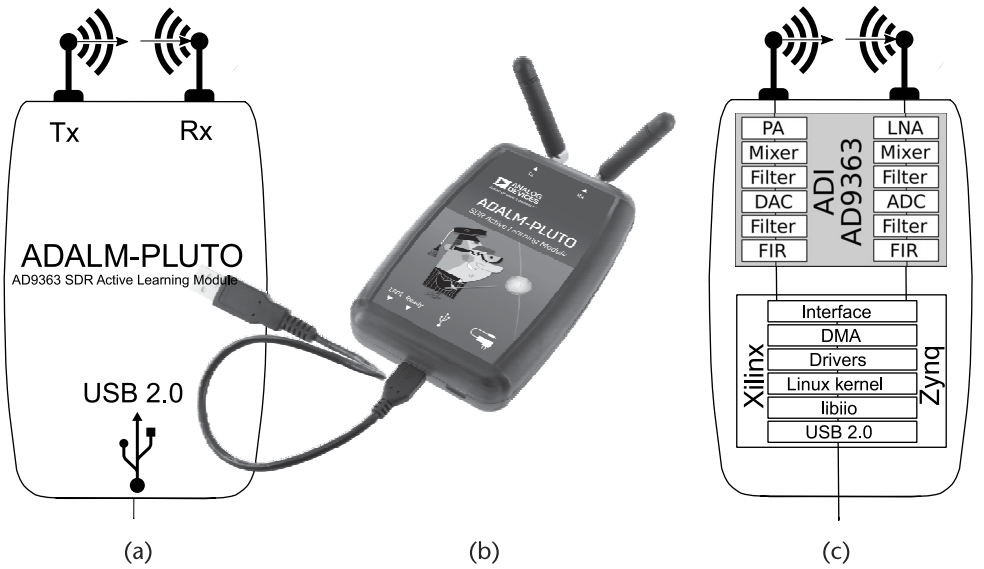

Figure 5.1 Views of the ADALM-PLUTO SDR. (a) Simplified block diagram of the ADALM-PLUTO, (b) photo of the ADALM-PLUTO [1], and (c) I/O on the ADALM-PLUTO.

- USB 2 OTG (480 Mbits/seconds), device mode
  - libiio USB class, for transfering IQ data from/to the RF device to the host
  - Network device, provides access to the Linux on the Pluto device
  - USB serial device, provides access to the Linux console on the Pluto device
  - Mass storage device
- USB 2 OTG (480 Mbits/seconds), host mode
  - Mass storage device, plug in a thumb drive, and capture or playback waveforms
  - Wifi dongle, access the Pluto SDR via WiFi
  - Wired LAN, access the Pluto SDR via wired LAN
- External power, for when using the Pluto SDR in host mode.

It is possible to run the Pluto SDR out of spec and extend the frequency range to 70–6,000 MHz to be able to capture and listen to FM broadcasts (in the 87.5–108.0 MHz bands most places, 76–95 MHz in Japan, and legacy 65.8–74.0 MHz in some Eastern European countries) at the low end, and the all the interesting things happening in 5.8-GHz ISM worldwide bands.

Because of the wide tuning range, 70–6,000 MHz, which is over three orders of magnitude, there are no band-specific receive or transmit filters in the Pluto SDR. What this means is that from a receive side, everything that is broadcasting from 70–6,000 MHz will be picked up, and could affect your signal. This is normally only an issue when you are trying to receive a very low amplitude signal. More about this in Section 5.2.6.

5.1.2 AD9363 Details

At the front of the Pluto SDR is a AD9363 5.2 transceiver from Analog Devices Inc., which is responsible for capturing and digitization of the RF data. This transceiver

provides amplification, frequency translation (mixing), digital conversion, and filtering of transmitted and receive signals. In Figure 5.2 we provide a detailed outline of the of the AD9363. While it can look complicated to the beginner, it is has nothing more than the three sections we mentioned before: an analog RF section, an analog baseband section, and some signal processing for both receive and transmit. It is important to understand the physical analog and digital hardware boundary because it will provide the necessary knowledge to configure the device from MATLAB and understand nonidealities experienced in the transmitted and received data. However, an extreme indepth understanding of the device is not required to effectively work with a SDR but some basics are invaluable.

We will discuss the AD9363 from the perspective of the receiver, but logically the same operations just apply in reverse order for the transmitter. At the very front of the AD9363 is a low-noise amplifier (LNA) providing analog gain that is a component of the automatic gain control (AGC) pipeline of the receiver. Following the LNA is the mixer, which is responsible for direct frequency translation. Unlike many traditional heterodyne transceivers, the AD9363 is a direct conversion, or ZeroIF design that does not utilize an intermediate frequency (IF) stage. For more details on the trade-offs between heterodyne and direct-conversion receivers, consider reading Razavi [2].

The mixer in the AD9363 operates from 325 MHz to 3.8 GHz within datasheet specification [2], but software modifications can be made to expand this range, which we will discuss in Section 5.2.6. Prior to this mixing process, the signal is split and fed along two different but identical paths. This process creates the in-phase and quadrature components of our signal through a simple phase rotation of the mixer's clock. Effectively this doubles the effectively bandwidth of the receiver since the in-phase and quadrature signals are orthogonal (bandwidth is $-\frac{f_s}{2}$ to $\frac{f_s}{2}$).

After mixing, the signal is filtered to remove aliasing effects of the now down-mixed signal and to reduce out of band interference and noise. The combined transimpedance amplifier (TIA) and analog filter are configured together to maintain the desired analog bandwidth, which can range from 200 kHz to 20 MHz. The TIA acts as a single pole filter and the analog programmable filter is a third-order Butterworth.

The final stage of the AD9363 is the digital conversion and decimation stage. Here the ADC will typically run at a much higher rate than the desired receive bandwidth, but the ADC itself will not provide all 12 bits defined in the specifications. The additional bits are gained in the halfband filter (HBF) stages, which will allow bit growth. The ADC itself only provides ~ 4.5 bits of resolution. This is a typical design for sigma-delta converters (Σ-Δ ADC), which inherently have low noise and run faster than the alternative successive approximation (SAR) ADCs. Refer to Section 2.5.4 for more information about Σ-Δ ADCs. However, by utilizing a very high speed ADC and associated HBFs the receive signal can be digitized at 12 bits at the desired configured sample rate. Therefore, for the best signal resolution is achieved through large oversampling of the input signal and then followed by several decimation stages.

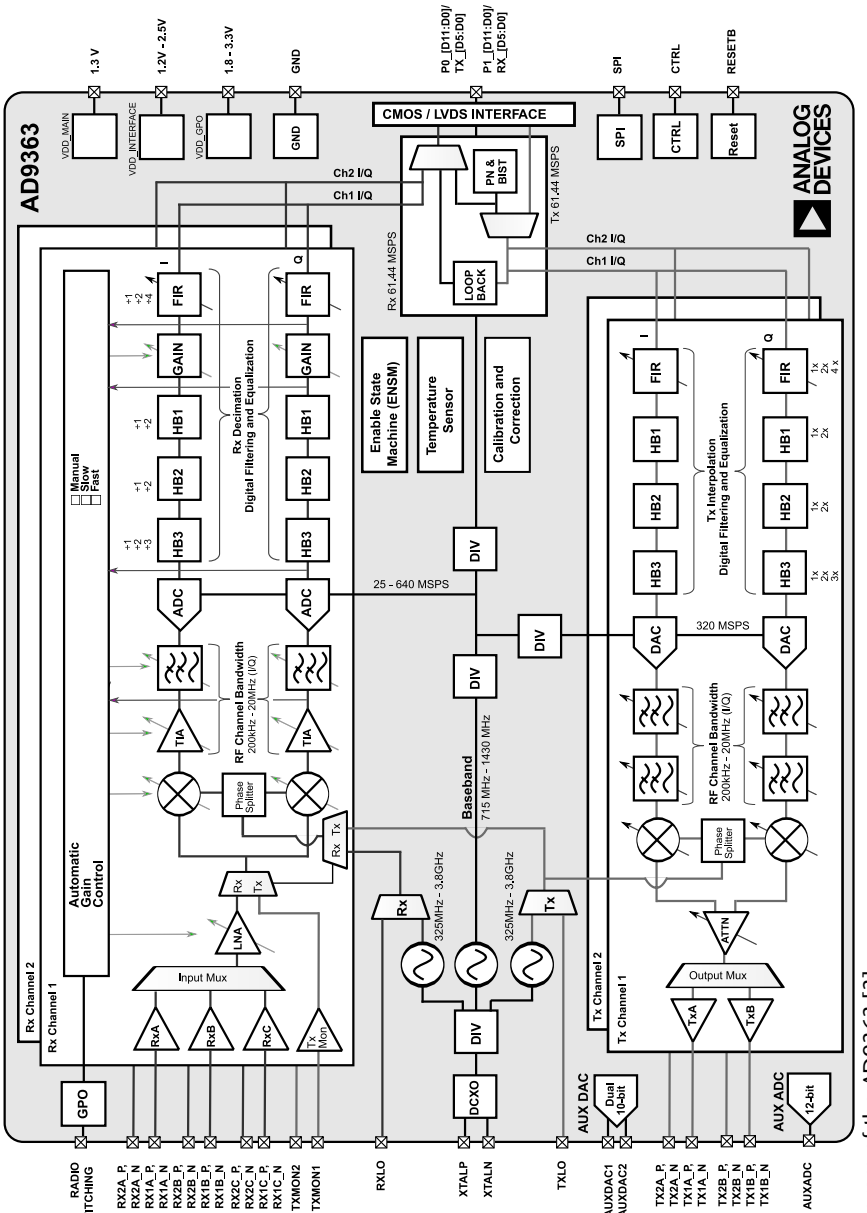

Figure 5.2 Block diagram of the AD9363 [3].

5.1.3 Zynq Details

Once the data is digitized it is passed to the Xilinx Zynq System on Chip (SoC), shown in Figure 5.3. The Zynq-7000 family offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and ASSPs. Providing integrated ARM Cortex-A9 based processing system (PS) and programmable logic (PL) in a single device, the Zynq is the used in the Pluto SDR as the main controller.

Having the combination of the programmable logic and a programming subsystem provide some unique advantages. The AD9363 ADC's lowest data conversion rate is 25 MHz. The maximum amount of decimation allows is 48. This provides a lowest sample rate of 520.833 kSPS. An additional divide by 8 decimation filter was put inside the FPGA to extend the lowest sample rate to 65.1042 kSPS. Running Linux on the ARM-A9 inside the Pluto SDR provides some unique advantages. Being able to use the Linux IIO infrastructure allows existing device drivers to be used for the AD9363. Controlling all aspects of the device, from sample rates, to FIR settings, to LO settings, to the additional decimation filters, this proven software did not have to be touched for the creation of the Pluto SDR.

Once the digital receive IQ data from the AD9363, described in Section 5.1.2 is transferred to the FPGA fabric, the AXI DMAC core writes that to the Pluto's external memory. During this step, a change is made from unique continuous samples to *n*-sample buffers.

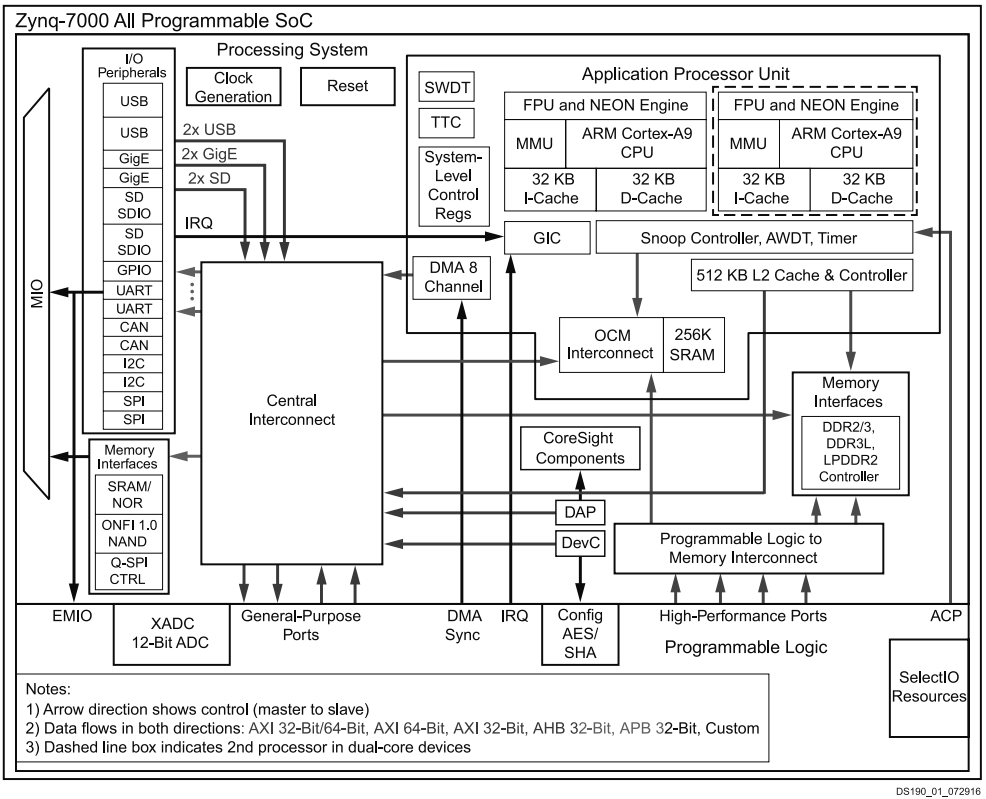

Figure 5.3 Block diagram of the Zynq [4].

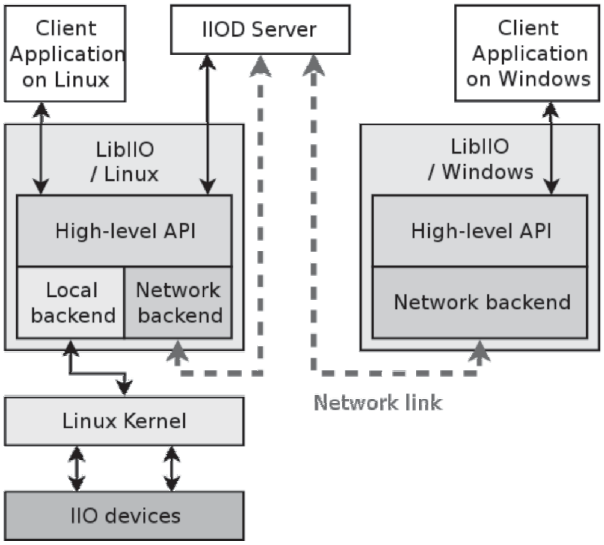

Figure 5.4 Block diagram of libiio and iiod [5].

5.1.4 Linux Industrial Input/Output Details

The industrial input/output (IIO) subsystem inside the Linux kernel is intended to provide support for devices that in some sense are ADCs or DACs, which don't have their own existing subsystems (like audio or video). This is not specific to Pluto nor specific to any SDR implmentation. It is an open-source standard adopted by many different manufactures for providing a common API to deal with a variety of different devices, This includes, but is not limited to, ADCs, accelerometers, gyros, IMUs, capacitance to digital converters (CDCs), pressure sensors, color, light and proximity sensors, temperature sensors, magnetometers, DACs, direct digital synthesis (DDS), phase-locked loops (PLLs), variable/programmable gain amplifiers (VGA, PGA), and integrated RF transceivers, like the AD9363.

There are three main aspects:

- The Linux kernel IIO driver, which runs inside the Linux kernel, in this case in the ARM in the Pluto SDR.
- libiio, the userspace library for accessing local and remote IIO devices, in this case both in the ARM, and on the host.
- iiod, the IIO Daemon, responsible for allowing remote connection to IIO clients, in this case on the ARM inside the Pluto SDR.

libiio is used to interface to the Linux industrial input/output (IIO) subsystem. libiio can be natively used on an embedded Linux target (local mode) or to communicate remotely to that same target from a host Linux, Windows, or MAC over USB, Ethernet, or Serial.

Although libiio was primarily developed by Analog Devices Inc., it is an active open-source library that many people have contributed to. It released under the GNU Lesser General Public License, version 2.1 or later, this open-source license allows anyone to use the library on any vendor's processor/FPGA/SoC that may be controlling any vendor's peripheral device (ADC, DAC, etc.) either locally or

remotely. This includes closed- or open-source, commercial or noncommercial applications (subject to the LGPL license freedoms, obligations and restrictions).

Once buffers of Rx data are in external memory, they are passed to iiod, the IIO Daemon. The iiod is responsible for managing various iio clients (normally remote on a network or USB), and translating their requests to local devices. It is able to accomplish this via the libiio library, which provides access to the AD9363 through a series of interfaces. Convinently, the libiio API access to the transceiver is identical whether working on the ARM or on a host PC which also has a libiio driver installed. Therefore, code can be implemented on the host machine connected to Pluto SDR and then deployed onto the ARM with the same code.

5.1.5 MATLAB as an IIO client

MATLAB can be used as a cross-platform IIO client to interface with the Pluto SDR. It includes a Pluto SDR system object interface. A fundamental background on system objects in MATLAB is provided in Appendix B.3. The two system objects provided in the hardware support package (HSP) for Pluto SDR are:

- comm.SDRRxPluto: Pluto SDR Receiver System object
- comm.SDRTxPluto: Pluto SDR Transmitter System object

These objects are typically constructed through the `sdrrx` or `sdrtx` function calls as in Code 5.1.

Code 5.1 Instantiating Pluto SDR System Objects: **`pluto1.m`**

```
 1 rx = sdrrx('Pluto')
14 tx = sdrtx('Pluto')
```

However, these objects can also be directly instantiated directly. The resulting object of `sdrrx` either way will have the following basic properties, which will be directly printed to the terminal when not using the semicolon as Code 5.2.

Code 5.2 Instantiating Pluto SDR System Objects: **`pluto1.m`**

```
 1 rx = sdrrx('Pluto')
 2 rx =
 3   comm.SDRRxPluto with properties:
 4   DeviceName: 'Pluto'
 5   RadioID: 'usb:0'
 6   CenterFrequency: 2.4000e+09
 7   GainSource: 'AGC Slow Attack'
 8   ChannelMapping: 1
 9   BasebandSampleRate: 1000000
10   OutputDataType: 'int16'
11   SamplesPerFrame: 3660
12   ShowAdvancedProperties: false
```

Since the Pluto SDR is part of a larger family of SDR devices, it shares the *DeviceName* attribute, which will be defined as `Pluto` for the Pluto SDR. As seen

from the usage in Code 5.2, there are various attributes for the System object.

- *RadioID* is related to the enumerate interface that the radio is using. This will either be USB:# where # is a number associated with the number of USB radios connected to MATLAB starting at zero, or ip:<ipaddress>, which utilizes the Pluto SDR over Ethernet.
- *CenterFrequency* defines the RF center frequency in hertz. Note there are separate Rx and Tx LO, on the Pluto SDR, and these are managed separately in the Rx and Tx Objects.
- *BasebandSampleRate* defines the sample rate of the in-phase/quadrature receive chains, respectively. Note, there is only one clock generator for both the ADC and DAC in the AD9363, so these must be set to the same value managing Rx and Tx on the same device.
- *GainSource* has three possible options: Manual, AGC Slow Attack, and AGC Fast Attack. When Manual is selected, another option called Gain will become available. This Gain value is associated with a gain table located within the AD9363. Therefore, when the Gain value changes multiple stages in the receive path shown in Figure 5.2 are updated based on this internal table. Custom gain tables can be used if necessary. However, such settings are considered advanced implementation options and will not be considered in this book. The other GainSource settings enable state machine based gain control within the AD9363 to adapt during operation based on receive signal strength (RSSI).
- The ChannelMapping attribute for the Pluto SDR can only be set to 1. However, on other SDRs in the Analog Devices Inc. family this is used for multichannel (multiple-input and multiple-output, or MIMO) reception.
- *OutputDataType* determines the format data is provided out of the object. Technically, from the AD9363 and libiio, MATLAB can only receive 16-bit complex integers, but we can tell MATLAB to cast them to other data types by default. Typically we will cast them to doubles since they provide the most precision, and working with trigonometric functions will require double or single precision data. As mentioned previously the receive and transmit paths only provide 12 bits of resolution, but since several of the hardware and software layers can only deal with base 8-bit types these 12 bits are provided as a 16-bit integer. For reference, on the receive side the 16-bit samples are sign extended and the transmitter will simply throw away the lowest four significant bits.
- *SamplesPerFrame* determines the number of samples in the buffer or frame that is passed to MATLAB from iiod. This will be the size of the vector provided at a given call to the object. This data will always be continguous as received from the radio unless an overflow has occurerd. However, successive calls to an instantiated object will not guarantee buffer-to-buffer continuity. Therefore, it can be useful to collect large amounts of data at a given time for processing.

The transmitter system object `comm.SDRTxPluto` has nearly identical properties except for *GainSource*, *SamplesPerFrame*, and *OutputDataType*, which do not make sense in the transmitter context.

5.1.6 Not Just for Learning

The architecture combination of the AD936x RF transceiver, the Zynq SoC, and external memory, which is found on the Pluto SDR should not just be thought of as just a learning platform. There are many commercial systems built on a similar architectures that can use the same software (HDL, Linux kernel, and IIO) to communicate with the IIO clients (like MATLAB). For example, Epiq Solutions, located in Schaumburg, Illinois, builds an industrial-grade, commercial solution, shown in Figure 5.5 known as Sidekiq Z2.

Although the Sidekiq Z2 utilizes a similar architecture as Pluto SDR, it does it in a standards-compliant Mini PCIe card form factor measuring $\sim 51 \times 30$ mm. In addition, the Sidekiq Z2 incorporates RF filtering, a high-precision reference clock, a more powerful dual-core Zynq, an extended RF tuning range from 70 – 6,000 MHz using the AD9364, doing so with industrial temperature range (-40° – +70° C) rated components. This allows the Sidekiq Z2 to serve as the basis for real-world flexible RF solutions, even in harsh industrial environments.

By building on the same architecture and using the same software infrastructure, this allows algorithm developers to prototype using a device like the Pluto SDR, and when they are ready for an industrial form factor, or need something embeddable,

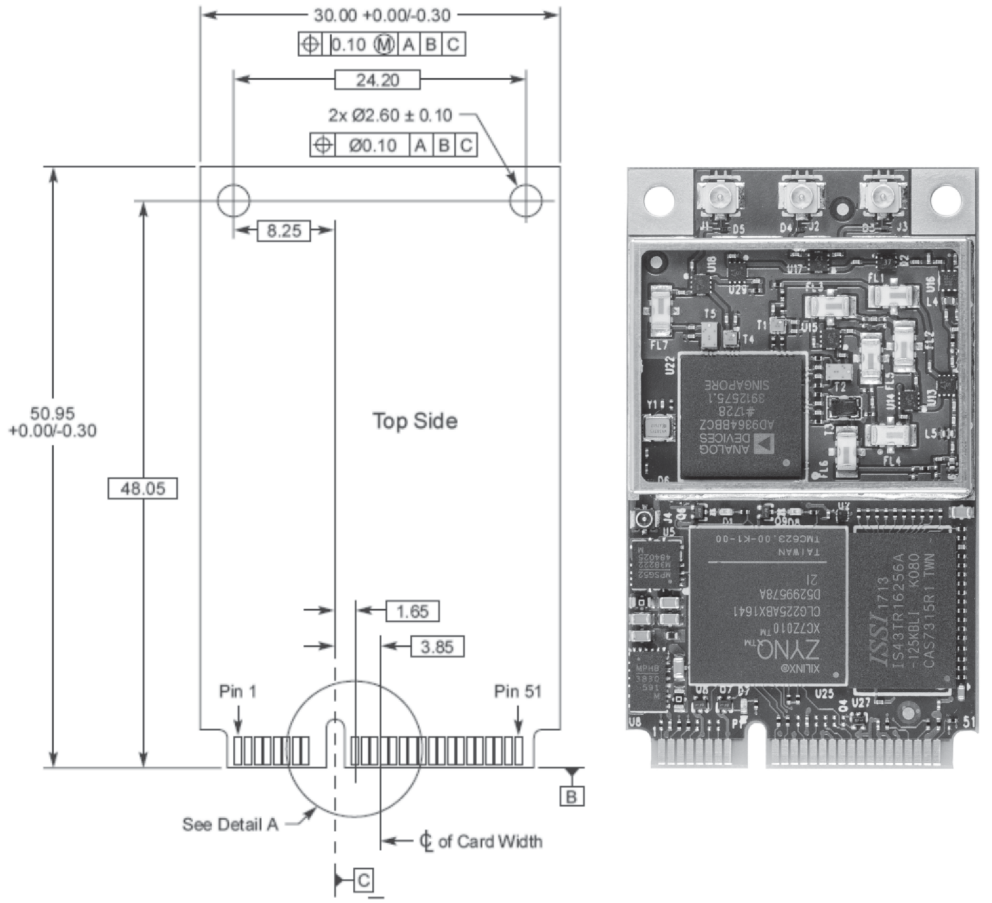

Figure 5.5 *Sidekiq Z2* from Epiq [6]. Not to scale.

they can easily transition over to the Sidekiq Z2. This also allows developers using the Sidekiq Z2 to communicate directly to MATLAB or any of the other IIO client software packages for testing in their specific end product. There are many things about RF that just cannot be simulated, and unless you have a final product with the ability to save and playback waveforms, or to remotely connect to the device, making a final release to production can be difficult to impossible. There is nothing more frustrating for an engineer than to have something that works on the bench fail in the field and have to go there for local troubleshooting. Using the provided open-source, industry-standard frameworks can reduce that.

5.2 Strategies For Development in MATLAB

As we discussed in Section 5.1.5, controlling how data enters MATLAB is very important for consistent operation. In this section we will discuss some strategies for structuring MATLAB code to effectively develop an algorithm. To help guide the development process we have provided templates to show how to appropriately form source around the radio's API. These templates can help progression of designs to real-time or offline work without the radio attached.

5.2.1 Radio I/O Basics

In each of these templates we will assume that a radio has been instantiated as the object `rx`, as in running the code in Code 5.1. Additionally, we assume that the SamplesPerFrame parameter of the object is set to some variable *frameSize*. In the first template presented in Code 5.3 we first collect *framesToCollect* frames of data, where each frame is of *frameSize* samples. The code in Code 5.1 tries to guarantee that we have collect *framesToCollect* × *frameSize* samples of continguous data from the radio with gaps. This is a good technique if more data than 2^{20} samples need to be collected, which is the maximum value you can make the SamplesPerFrame parameter of the Pluto SDR System object. After this data is collected we perform some processing, which in this case is a visualization with `dsp.SpectrumAnalyzer` scope.

Alternatively, if we don't require fresh samples for every run it can be useful to save data to a file so we don't have to worry about clearing data from the workspace. A useful tool for this work is the `comm.BasebandFileWriter`, which saves complex received data with additional metadata like sample rate to a file for off-line processing. We show usage of the `comm.BasebandFileWriter` system object in Code 5.4 with the collected data from Code 5.3.

Utilizing data from a filesource can make testing much more repeatable when debugging issues during algorithm development. It can also be much faster to address a file than to go out to the radio and pull in new data, especially when setting up a transmitter is also required. In Code 5.5 we show use of the complementary System object to the`comm.BasebandFileWriter` called `comm.BasebandFileRead`. The `comm.BasebandFileRead` System object can be configured to provide a specific amount of samples for each call to the object through the SamplesPerFrame parameters to emulate using the Pluto SDR. This is a useful strategy when a radio is not available.

Code 5.3 Template Example: **`template1.m`**

```
%% Template 1
% Perform data collection then offline processing
data = zeros(frameSize, framesToCollect);
% Collect all frames in continuity
for frame = 1:framesToCollect
    [d,valid,of] = rx();
    % Collect data without overflow and is valid
    if ~valid
        warning('Data invalid')
    elseif of
        warning('Overflow occurred')
    else
        data(:,frame) = d;
    end
end

% Process new live data
sa1 = dsp.SpectrumAnalyzer;
for frame = 1:framesToCollect
    sa1(data(:,frame)); % Algorithm processing
end
```

Code 5.4 Template Example for Saving Data: **`template1.m`**

```
% Save data for processing
bfw = comm.BasebandFileWriter('PlutoData.bb',...
    rx.BasebandSampleRate,rx.CenterFrequency);
% Save data as a column
bfw(data(:));
bfw.release();
```

Code 5.5 Template Example for Saving Data: **`template2.m`**

```
%% Template 2
% Load data and perform processing
bfr = comm.BasebandFileReader(bfw.Filename, 'SamplesPerFrame',frameSize);
sa2 = dsp.SpectrumAnalyzer;
% Process each frame from the saved file
for frame = 1:framesToCollect
    sa2(bfr()); % Algorithm processing
end
```

Once an algorithm has been tuned we can place the processing sections within the main loop with the Pluto SDR's System object like in Code 5.6. This type of processing is defined as stream processing in MATLAB [7], where we immediately work on new data. This will limit the amount of information required to be collected and can be useful if logical actions, such as changing channels, need to be applied. As long as the algorithm placed within the loop is able to keep up with the data streaming in from the radio, no overflow warning should occur. This is known as

operating in real time. However, since there is some elasticity with the buffers to and from the radio overflows will not always happen immediately. For example, if the algorithm is only slightly slower than the radio's data rate then it may take many loop iterations for the internal radio and operating system buffers to fill to a point of overflow. Therefore, when testing for real-time operation it is useful to run an algorithm for several minutes to check if an overflow will occur.

Code 5.6 Template Example for Saving Data: **`template3.m`**

```
%% Template 3
% Perform stream processing
sa3 = dsp.SpectrumAnalyzer;
% Process each frame immediately
for frame = 1:framesToCollect
    [d,valid,of] = rx();
    % Process data without overflow and is valid
    if ~valid
        warning('Data invalid')
    else
        if of
            warning('Overflow occurred')
        end
        sa3(d); % Algorithm processing
    end
end
```

5.2.2 Continuous Transmit

Anytime the Pluto SDR is powered on, the transceiver is activated and begins to operate even if the user did not intend to. When powered on Pluto SDR will transmit data; this is just how the transceiver was designed. Therefore, when using just the receiver System object (`comm.SDRRxPluto`) data will be transmitted by the actual device. Normally, the transceiver will transmit the last buffer available in the DMA continuously until powered down. If the Tx LO is accedentily tuned to the same value as the RX LO, when communicating between multiple radios or just receiving, this continuous transmit operation can cause significant interference.

Code 5.7 Template Example Transmit Repeat: **`transmitRepeat.m`**

```
% Transmit all zeros
tx = sdrtx('Pluto');
fs = 1e6; fc = 1e4; s = 2*pi*fs*fc*(1:2^14).';
wave = complex(cos(s),sin(s));
tx.transmitRepeat(wave);
```

There are two options to reduce or even remove this interference. The first option is to instantiate a transmitter System object (`comm.SDRTxPluto`) and write a vector of zeros to the object as shown in Code 5.8. This will reduce the transmit energy of the device to a minimal level. However, there will still be leakage into the receiver's data due to Tx LO leakage.

Code 5.8 Template Example Transmit Zeros: **`transmitzeros.m`**

```
% Transmit all zeros
tx = sdrtx('Pluto');
tx(zeros(1024,1));
```

Alternatively, we can simply shift the LO of the transmitter to a frequency beyond the receive bandwith. We demonstrate this configuration in Code 5.9 where we offset the *CenterFrequency* of the transmitter's System object. This is a better alternative since there LO leakage from the transmitter should not appear in the received data.

Code 5.9 Template Example Transmit Zeros: **`transmitoffset.m`**

```
% Move transmitter out of receive spectrum
tx = sdrtx('Pluto');
rx = sdrrx('Pluto');
tx.CenterFrequency = rx.CenterFrequency + 100e6;
```

5.2.3 Latency and Data Delays

When working with the Pluto SDR from a host it will soon become obvious that there will delays associated with transmitted and received data, especially when performing loop-back operations from transmitter to receiver. These delays are a function of the internal buffers and FIFOs of the host PC and the Pluto SDR itself. However, there exists both deterministic and elastic/random delay across the various layers illustrated in Figure 5.6. The reason why there is a nondeterministic delay in the transport stacks is due to contention in the kernels of the ARM and host PC. In the worst case the round-trip time should be on the order of tens of milliseconds. In order to guarantee certain delays or minimal delay would require operating directly on the ARM or on the FPGA. Nevertheless, development in these processing regions becomes much more time consuming. However, any radio platform that passes data into MATLAB will have to deal with these delays, but they may have different sources depending on the radio architecture.

One of the complications of looking at Figure 5.6 is that in many places, the transport is defined by bytes, while in other places it convenient to discuss samples. A single I/Q sample (complex) in this circumstance (singe radio channel) is two 16-bit samples, or 4 bytes.

To review Figure 5.6 in detail on the receive side, when an IIO client like MATLAB requests a buffer of 32768 samples at a sample rate of 1 MSPS:

- `iiod` will capture 32768 continuous samples via libiio. This will take 32.768 milliseconds, and consume 131,072 bytes. While `iiod` ensures the packet is contiguous by not sending any data until it has all been captured, it does increase a fixed latency of the sample rate $\times$ the number of samples being captured. `iiod` was designed to ensure there are no memory copies after the data has been captured, and simply passes pointers around from the AD9363 driver to `libiio` to `iiod` and then to the USB stack.

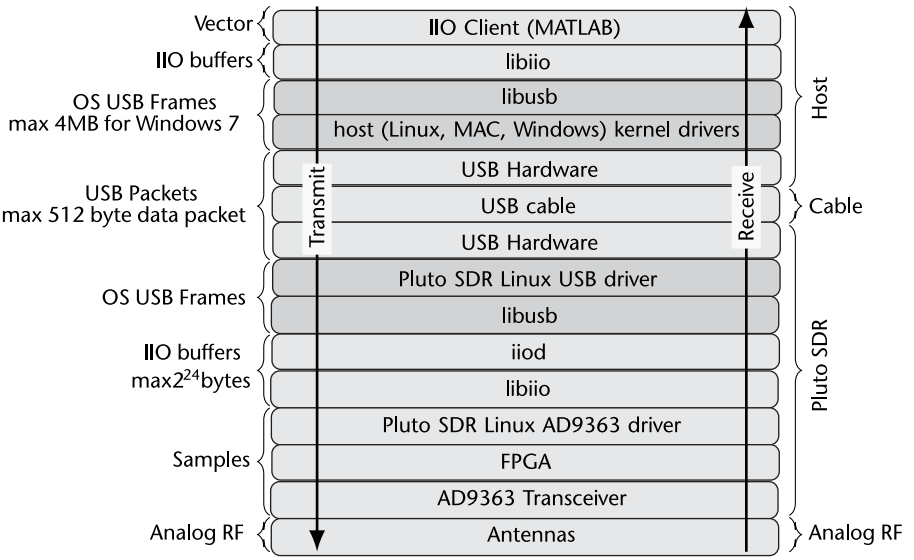

Figure 5.6 Hardware and software stacks transmitted and received packets must traverse from MATLAB to the transceiver throught the various layers in the system.

- This data will then be passed to the Linux kernel on the Pluto where it will be segmented into 512 byte USB packets (per the USB 2.0 spec), were it will be reassembled into a 131,072-byte buffer on the host side. The USB stack will introduce an unknown, and unbounded latency, which will be determined by the host operating system and how much free memory it has available.
- Once the entire buffer is assembled, it is passed to libiio, where it is then passed (untouched) to the iio client, in this case MATLAB, which may do further processing.
- In this case, MATLAB may cast the data from fixed point to floating point or double ± 1.0 data, which also takes some time.

Due to these delays we must design our algorithms and overall system with an understanding of these conditions. For example, if a single frame wanted to be transmitted and received from the same Pluto SDR we can observe a large gap of samples before the frame is observed at the receiver output vectors due to these delays.

It is for these reasons that many systems want to put as much processing as possible as close to the radio as possible. With this architecture, that would be on the FPGA or the ARM processor. However, on the Pluto SDR, the FPGA is small, and there is only a single core ARM, limiting its targeting capabilities.

5.2.4 Receive Spectrum

The receive signals we observe will always will contain noise from the environment and from the radios themselves. To provide perspective on this noise we demonstrate a simple example using the script shown in Code 5.10.

If one employs Code 5.10, we will notice the spectrum is not perfectly flat, containing peaks or spurs in the spectrum. These are actually artifacts of the

Code 5.10 Template Example View Spectrum: **`template_rt.m`**

```
% View some spectrum
rx = sdrrx('Pluto');
rx.SamplesPerFrame = 2^15;
sa = dsp.SpectrumAnalyzer;
sa.SampleRate = rx.BasebandSampleRate;
for k=1:1e3
  sa(rx());
end
```

radio itself, which naturally come about through the complex receive chain, which provides signal gain enhancement through the AGC stages. Depending on the bandwidths we choose in the receive chain and the AGC configuration itself, we can modify these spurs. However, due to the complexity of the AD9363 transceiver this can be a challenging task since it contain thousands of registers for the AGC itself for configuration. Therefore, since these spurs can be minor relatively to the signal of interest itself we can simply ignore them or perform filtering in software to reduce their affects. Nonetheless, when working with Pluto SDR we should always be aware of these nonidealities in the hardware.

Fortunately, Pluto SDR does maintain many built-in calibrations to help reduce self-induced problems. These include RF DC tracking, quadrature tracking, and baseband DC tracking. To access these features, Pluto SDR enabled the parameter *ShowAdvancedProperties*, which will then display these features. Since the AD9363 is a direction conversion receiver, a common problem with such devices is the presence of a tone or energy at DC or near the zeroith frequencies. This is due to the radio itself. The DC tracking components, RF and baseband, both work to reduce these effects.

The last advanced feature is quadrature tracking. The quadrature tracking reduces and in-phase and quadrature (IQ) imbalance that may occur, which may be difficult to spot in a frequency domain plot. An imbalance would typically appear as a spur reflection in the frequency domain. When enabling quadrature tracking, these image spurs should be reduced significantly. However, when working with a constellation plot IQ imbalances become more noticable. There will always be some residual imbalance, but corrections are performed during initialization so it will not be improved over time necessarily.

5.2.5 Automatic Gain Control

One of the most complex pieces of the AD9363 is the AGC. The AGC is actually spread out through the receive chain apply gain at different stages and sensing the amplitude of the received signals. From the simplistic API of MATLAB we have three options: Manual, AGC Slow Attack, and AGC Fast Attack. Under Manual the receiver will simply fix the input gain to a specific value based on an internal gain table. Essentially the manual gain acts as a single index into the internal table. This Manual setting is useful when using cabling or when the transmitter is at a fixed known distance. Under the Manual setting it can make receiver algorithms easier to debug since the AD9363's state will remain the same.

In practice the amplitude of the receive signal will be unknown and even change across multiple transmissions. Therefore, we can simply utilize the AGC Slow Attack and AGC Fast Attack modes. If the received transmission is very short or rapidly changes in amplitude AGC Fast Attack will be the best option. However, AGC Slow Attack will be prefered when received data has a relatively maintained amplitude. An obvious question would be, why not always used AGC Fast Attack? The main disadvantage of AGC Fast Attack is that it can create large discontinuities in the amplitude, which may distort the received signal. For illustration we provide a comparison of a system setup in loopback with a single Pluto SDR transmitted a QPSK signal. We provide a time plot of the received signal to demonstrate both the delay of the received signal and the changes in amplitude when using the different gain modes. As we can observe there are rapid changes in gain for the AGC Fast Attack mode, but the gain is more gradual over time for the AGC Slow Attack mode. The determination of the correct mode is not always obvious. Therefore, time series testing and observation can be useful during algorithm development.

5.2.6 Common Issues

The way that various signals mix can also be an issue. As described in Section 2.3.1, the mixer accepts a single-ended local oscillator (LO).

5.3 Example: Loopback with Real Data

Now that we have a solid understanding of the system object that controls the Pluto SDR and some coding structures for interacting with Pluto, we can explore a simple loopback example. Loopback means that the waveform is both transmitted and received by the same Pluto SDR, which is a common strategy for algorithm debugging and general hardware debugging.

Starting from the transmitter (tx), in this example you will first notice we have set the gain to -30, which is 20 dB down from the default. The reasoning behind reducing the gain is to prevent clipping or saturation at the receiver. Since the transmit and receive antennae are about 12 mm from one another the received signal will be rather loud. The sinewave used here was simply generated by the `dsp.SineWave` system object for convenience, which will provide a complex output for the Pluto SDR. A special method called `transmitRepeat` was used, which will continuously transmit the passed vector. This will prevent any gaps in the transmission.

In the received waveform in Figure 5.7 we can observe both the complex and real components of the signal over time, which are $\frac{\pi}{2}$ radians out of phase with one another as expected. At the start of the signal we can observe a larger amplitude than future samples. This is a direct result of the AGC settling, and since the AGC starts from a high gain by default at configuration or setup time. In MATLAB this setup time only occurs on the first call to the receiver object (rx), not the construction time. This is also known as the first Step method call, which will call an internal method of the system object called setupImpl.

Code 5.11 Loopback Example: **`loopback.m`**

```
% Setup Receiver
rx=sdrrx('Pluto','OutputDataType','double','SamplesPerFrame',2^15);
% Setup Transmitter
tx = sdrtx('Pluto','Gain',-30);
% Transmit sinewave
sine = dsp.SineWave('Frequency',300,...
                    'SampleRate',rx.BasebandSampleRate,...
                    'SamplesPerFrame', 2^12,...
                    'ComplexOutput', true);
tx.transmitRepeat(sine()); % Transmit continuously
% Setup Scope
samplesPerStep = rx.SamplesPerFrame/rx.BasebandSampleRate;
steps = 3;
ts = dsp.TimeScope('SampleRate', rx.BasebandSampleRate,...
                    'TimeSpan', samplesPerStep*steps,...
                    'BufferLength', rx.SamplesPerFrame*steps);
% Receive and view sine
for k=1:steps
  ts(rx());
end
```

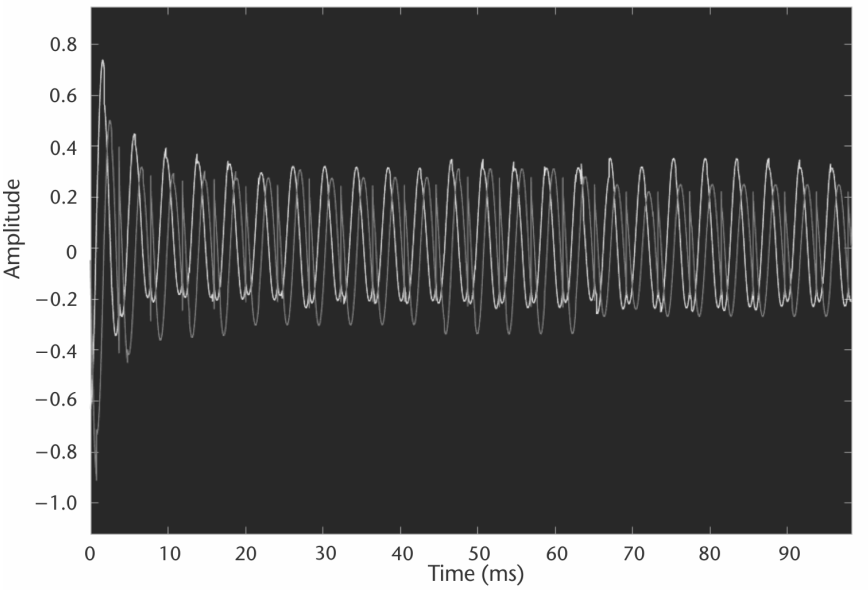

Figure 5.7 Loopback sinewave of 300 Hz from Pluto SDR generated by Code 5.11.

Looking closer on the sine wave, discontinuities can be observed that result from small gaps between received frames. This is actually a result of the transmitter that is repeating the same waveform over and over. Since the transmitted waveform is not cyclic when relating the start and end of the data passed to the transmitter we observe these discontinuities. To avoid these we would need to make sure a period ends at the end of the passed frame and started exactly at the beginning of the passed frame.

5.4 Noise Figure

With the receiver pipeline discussed in the previous section, the AD9363 is able to achieve a noise figure (NF) of only 2 dB at 800 MHz. NF is a common metric to compare receivers, which is a measure of the internal or thermal noise of the electrical components of the device. NF is calculated based on the SNR ratio of output to input in dB as

$$NF = 10\,log_{10}\frac{SNR_{INPUT}}{SNR_{OUTPUT}}, \tag{5.1}$$

where NF is in dB, and both SNR_{INPUT} and SNR_{OUTPUT} are in linear scale [8]. For comparison, another popular SDR the RTL-SDR has a NF of 4.5 dB, which is almost double the NF of the Pluto SDR. NF is important because it will affect the eventual sensitivity of the receiver, and the lower the better. The easiest way to measure NF is with a noise figure analyzer that generates noise into the receive path of a system, which is then fed back out. The output noise is then measured and compared with the input noise to create an NF measurement. Figure 5.8 demonstrates an example set up to evaluate NF using a noise figure analyzer where the device under test (DUT) will operate in loopback mode transmitted out the received noise. When measuring NF it can be useful to use a very stable LO to drive the device to limit internal noise, since generally the noise of a DUT needs to be measured, not the source oscillator driving it. Furthermore, NF is always based on frequency and the measured NF of a DUT such as an SDR will typically be based on the noisiest part of the device.

NF is a common metric that hardware manufacturers typically use but it can be difficult to relate to a communications problem, since communications engineers tend to measure further down the receive chain after the filter stages. NF also requires specific and usually expensive instruments to measure. However, it is important to understand its meaning when specified on a datasheet, since it can give a rough estimate on the low bound for error vector magnitude measurements.

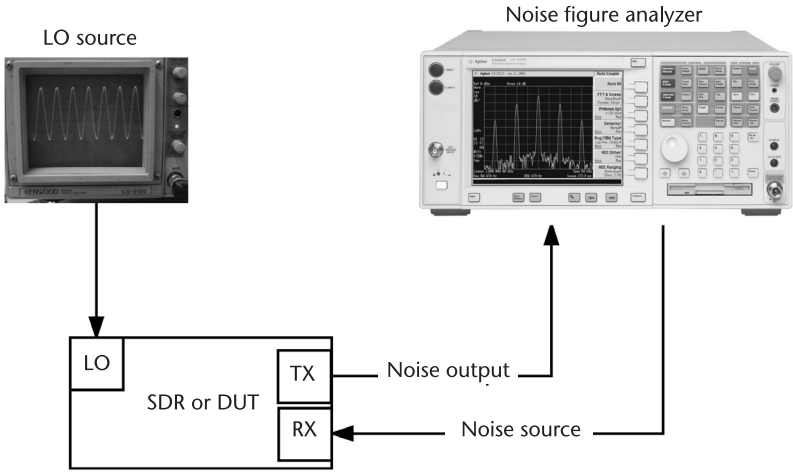

Figure 5.8 Example noise figure evaluation of SDR or device under test using a noise figure analyzer and external stable LO source.

References

[1] Analog Devices ADALM-PLUTO Software-Defined Radio Active Learning Module, http://www.analog.com/plutosdr 2017.

[2] Razavi, B., "Design considerations for direct-conversion receivers," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Vol. 44, No. 6, June 1997, pp. 428–435.

[3] Analog Devices AD9363 Datasheet, http://www.analog.com/AD9363 2015.

[4] Xilinx, *Zynq-7000 All Programmable SoC Overview*, [Online], 2017, https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[5] Analog Devices, What is libiio?, [Online], 2018, https://wiki.analog.com/resources/tools-software/linux-software/libiio.

[6] Epiq Solutions, *Sidekiq Embeddable Software Defined Radio 70MHz–6GHz*, [Online], https://epiqsolutions.com/sidekiq/.

[7] The Math Works, Inc., Stream Processing in MATLAB: Process Streaming Signals and Large Data with System Objects, https://www.mathworks.com/discovery/stream-processing.html.

[8] Friis, H. T., "Noise Figures of Radio Receivers," *Proceedings of the IRE*, Vol. 32, No. 7, July 1944.