Setting Up and Capturing Samples with the ADALM-Pluto

The ADALM-Pluto is a simple, but powerful software defined radio (SDR) that is excellent for teaching basic concepts in digital communications and wireless. In this first lab, you will learn to:

- Initalize and configure the ADALM-Pluto device
- Connect one or more Pluto devices to the host computer for single device loopback and two device communication links
- Transmit complex baseband samples in a repeated loop from a Pluto
- Receive a single frame of complex baseband sampes to perform offline processing
- Capture multiple frames and detect and visualize overflow (graduate students only)

Submissions: Perform the lab in pairs so that you will have two Pluto devices. You can run in either of the two device configurations described below (single or two hosts). Either way, fill in all sections labeled `TODO` . Print and submit the PDF. Do not submit the source code.

Setting Up the Pluto

Before starting this or any other labs or demos, you will need to follow the following steps:

- Purchase one or two Plutos. If you have one Pluto only, you will be limited to loopback modes. There are several vendors that sell the device. For example, it can be purchased directly from Analog devices.
- NYU students enrolled in the class will get one Pluto each. You can work in pairs for device-device experiments
- Follow the set up instructions on MATLAB set up page
- Plug in your Pluto device. You may need to run the command;

```
configurePlutoRadio('AD9364');
```

- You will need to run this command once for each device.

You are ready to go!

Set the Configurations

The labs can be run in one of three configurations:

- *One device loopback mode*: A single Pluto is connected to a single host and performs the TX and RX. This mode is the easiest to run, but the channel is only between the TX and RX antenna on the same device -- not very interesting. You should run this mode initially to make sure the system is working. To enable this mode set:

```
loopback = true;
runTx = true;
runRx = true;
```

- *Two devices / single host*: Two Pluto devices are connected to a single host. One pluto performs TX and the second performs RX. In this case, he MATLAB script below performs the functions

for both TX and RX. This mode is easier than the two host mode, but the channel is limited to the length of .the USB cables. To enable this mode set:

```
loopback = false;
runTx = true;
runRx = true;
```

- *Two devices / two hosts*: One Pluto device is connected to one host and performs the TX, a second Pluto One pluto performs TX and the second performs RX. You run two separate MATLAB scripts on each host. You set the parameters for the two hosts as follows

```
% On both hosts:
loopback = false;

% On the TX host
runTx = true;
runRx = false;

% On the RX host
runTx = false;
runRx = true;
```

TODO: Set the parameters as above depending on the desired configuration

```
% Set to true for loopback, else set to false
loopback = false;

% Select to run TX and RX
runTx = true;
runRx = true;
```

Creating TX and RX objects

Plug one or two Plutos into the USB ports of your computer. If it is correctly working, in each device the Ready light should be solid on and the LED1 light should be blinking. I have created a simple function, `plutoCreateTxRx` to detect the Pluto devices and configure them depending on the desired configuration. The following code will create the objects `tx` and `rx` that will be used for the remaining processing. The code sets other parameters for the Pluto including:

1. The baseband sample rate in Hz. In this demo, we have used a standard sample rate for 4G and 5G cellular systems.
2. The number of samples per frame. A frame is a block of samples that are transmitted or received by the SDR
3. The center frequency or carrier frequency. In this demo, we have set it to a frequency in the unlicensed band.

```
% clear previous instances
clear tx rx

% add path to the common directory where the function is found
addpath('..\..\common');
```

```
% Parameters
sampleRate = 30.72e6;
nsampsFrame = 2^12;
fc = 2.4e9;

% Run the creation function
[tx, rx] = plutoCreateTxRx(createTx = runTx, createRx = runRx, loopback = loopback, ...
    nsampsFrame = nsampsFrame, sampleRate = sampleRate, centerFrequency = fc);
```

Sending a Continous Wave

In this lab, we will send a simple complex exponential, also called a *continuous wave (CW) signal*. We can construct the signal as follows.

```
% Create a digital signal the length of one frame with a digital frequency
% of nu0 = 4/nsampFrame
nu0 = 4/nsampsFrame
x = exp(2*pi*1i*nu0*(0:nsampsFrame-1)');


% TODO:  Plot the real and imaginary components of the signal vs. time.
% Label the time axis in us (micro-seconds)
```

Now we transmit the wave. The `release()` command clears the TX buffer and the `transmitRepeat()` command sets the TX to continuously transmit the samples over and over again in a loop. This simple method is what we will use most commonly in the labs in this class. Real systems, of course, generally transmit different data in each frame.

```
if runTx
    tx.release();
    tx.transmitRepeat(x);
end

% If not running the RX, stop the live script now
if ~runRx
    return;
end
```

Capturing Data Once

We will now capture one frame of samples. To capture the samples, we use the `rx.capture()` method. This provides the samples in integer values. We scale them to floating point and plot them. If you are using two device mode, you should see a noisy version of the TX signal.

```
nbits = 12;               % number of ADC bits
fullScale = 2^(nbits-1);  % full scale

% Capture data
r = rx.capture(nsampsFrame);

% Scale to floating point
r = single(r)/fullScale;
```

```
% TODO:  Plot the real and imaginary RX samples vs. time in micro-seconds
clf;
```

Capturing Consecutive Frames (Graduate Students Only)

We can now capture multiple frame data frames in a loop. In each iteration of the loop, we capture data and store it. One can also perform any processing within the loop. The code:

```
[r, valid, over(i)] = rx();
```

returns three outputs:

- `r` : The data in the frame
- `valid` : Indicating if the data is valid
- `over`: Indicating if there was an *overflow* meaning the time between captures was too long and there was an overflow in the data buffer. In this case, the data is no longer continuous

Run the following code and print out the values of `over` to see if there was an overflow. Even though the PC is fast, for some reason, you may still get overflows.

```
nframes = 5;
data = zeros(nsampsFrame, nframes);
over = zeros(nframes,1);
for i = 1:nframes

    [r, valid, over(i)] = rx();
    if ~valid
        warning('Data is not valid');
    else
        data(:,i) = single(r)/(2^12);
    end


end

% TODO
% Print out the frame numbers of any overflows. If there were no overflows,
% print out none
```

Now create a single 1D vector, `dataConcat` with all the samples in the array `data`. This should have length `nsampsFrame*nframes`.

```
% TODO
%  dataConcat = ...
```

Plot the real component of `dataConcat` vs. time in milliseconds. If there were overflows, you will see discontinuities between the frames.

```
% TODO:  Plot the real(dataConcat) vs. time
```

```
```